

GPU Architectures and Programming

Prof. Soumyajit Dey
Computer Science and
Engineering
IIT Kharagpur



INDEX

S. No	Topic	Page No
	<i>Week 1</i>	
1	Review of basic COA w.r.t. performance	1
2	Review of basic COA w.r.t. performance	15
3	Review of basic COA w.r.t. performance	36
4	Review of basic COA w.r.t. performance	50
	<i>Week 2</i>	
5	Intro to GPU architectures	62
6	Intro to GPU architectures	76
7	Intro to GPU architectures	96
8	Intro to GPU architectures	112
	<i>Week 3</i>	
9	Intro to CUDA programming	124
10	Intro to CUDA programming (Contd.)	140
11	Intro to CUDA programming (Contd.)	150
12	Intro to CUDA programming (Contd.)	159
	<i>Week 4</i>	
13	Multi-dimensional mapping of dataspace; Synchronization	175
14	Multi-dimensional mapping of dataspace; Synchronization (Contd.)	190
15	Multi-dimensional mapping of dataspace; Synchronization (Contd.)	203
	<i>Week 5</i>	
16	Warp Scheduling and Divergence	221
17	Warp Scheduling and Divergence (Contd.)	235
18	Warp Scheduling and Divergence (Contd.)	249
	<i>Week 6</i>	
19	Memory Access Coalescing	265
20	Memory Access Coalescing (Contd.)	278
21	Memory Access Coalescing (Contd.)	291
22	Memory Access Coalescing (Contd.)	301
23	Memory Access Coalescing (Contd.)	317
24	Memory Access Coalescing (Contd.)	328

25	Memory Access Coalescing (Contd.)	340
26	Memory Access Coalescing (Contd.)	354
27	Memory Access Coalescing (Contd.)	369

Week 7

28	Optimizing Reduction Kernels	388
29	Optimizing Reduction Kernels (Contd.)	406
30	Optimizing Reduction Kernels (Contd.)	431
31	Optimizing Reduction Kernels (Contd.)	447
32	Optimizing Reduction Kernels (Contd.)	462
33	Optimizing Reduction Kernels (Contd.)	478
34	Optimizing Reduction Kernels (Contd.)	493

Week 8

35	Kernel Fusion, Thread and Block Coarsening	510
36	Kernel Fusion, Thread and Block Coarsening (Contd.)	526
37	Kernel Fusion, Thread and Block Coarsening (Contd.)	540
38	Kernel Fusion, Thread and Block Coarsening (Contd.)	559
39	Kernel Fusion, Thread and Block Coarsening (Contd.)	574
40	Kernel Fusion, Thread and Block Coarsening (Contd.)	589

Week 9

41	OpenCL - Runtime System	612
42	OpenCL - Runtime System (Contd.)	628
43	OpenCL - Runtime System (Contd.)	646
44	OpenCL - Runtime System (Contd.)	658
45	OpenCL - Runtime System (Contd.)	675
46	OpenCL - Runtime System (Contd.)	690
47	OpenCL - Runtime System (Contd.)	707

Week 10

48	OpenCL - Heterogeneous Computing	727
49	OpenCL - Heterogeneous Computing (Contd.)	742
50	OpenCL - Heterogeneous Computing (Contd.)	758
51	OpenCL - Heterogeneous Computing (Contd.)	772
52	OpenCL - Heterogeneous Computing (Contd.)	782

53	OpenCL - Heterogeneous Computing (Contd.)	798
----	---	-----

Week 11

54	Efficient Neural Network Training/Inferencing	814
55	Efficient Neural Network Training/Inferencing (Contd.)	830
56	Efficient Neural Network Training/Inferencing (Contd.)	847
57	Efficient Neural Network Training/Inferencing (Contd.)	865
58	Efficient Neural Network Training/Inferencing (Contd.)	885
59	Efficient Neural Network Training/Inferencing (Contd.)	904

Week 12

60	Efficient Neural Network Training/Inferencing (Contd.)	919
61	Efficient Neural Network Training/Inferencing (Contd.)	934
62	Efficient Neural Network Training/Inferencing (Contd.)	950
63	Efficient Neural Network Training/Inferencing (Contd.)	969

GPU Architectures and Programming
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No 1
Review of Basic COA w.r.t. Performance

Hi everybody. So, this will be our first lecture for the course on GPU architectures and programming. So from our introductory video, we have provided a brief outline of the different topics that we are going to cover in this course. And just going through some of the related slides of that video once again for our purpose.

(Refer Slide Time: 00:49)



So this is just going back into the history as we explained earlier. In personal computers, we had this notion of graphics pipeline, which was functioning in the form of Video Graphics Array, which eventually got replaced by a more programmable version which is the Graphics Processing Unit or GPU. Eventually people found that since it is a programmable parallel processor, it can be used for accelerating lot of general purpose workloads. Which is basically giving rise to the term GPGPU or General Purpose Graphics Processing Unit.

(Refer Slide Time: 01:27)

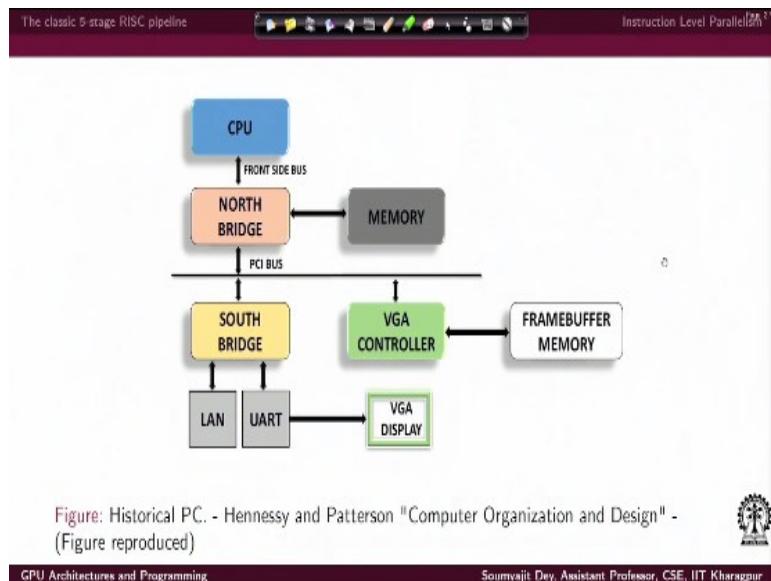


Figure: Historical PC. - Hennessy and Patterson "Computer Organization and Design" - (Figure reproduced)



GPU Architectures and Programming

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And we also provided this snapshot of how our standard CPU connects with a VGA and other related peripherals.

(Refer Slide Time: 01:38)

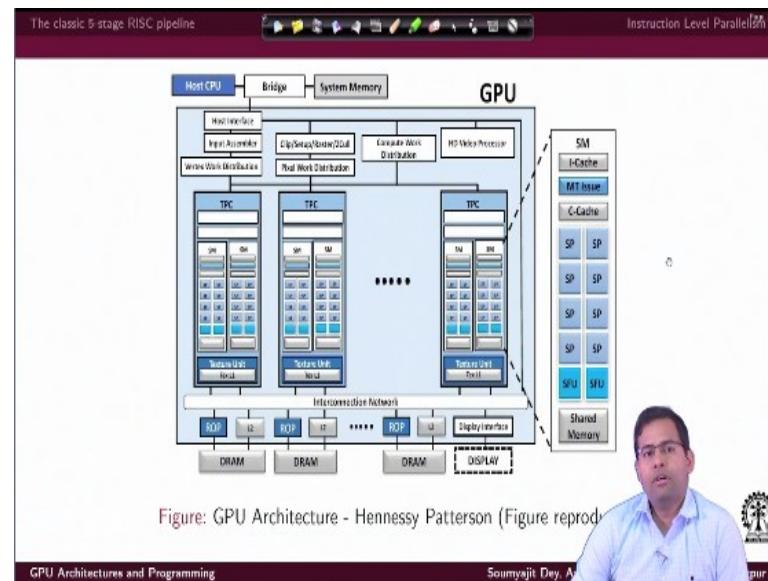


Figure: GPU Architecture - Hennessy Patterson (Figure reproduced)

Soumyajit Dey, A

We saw how really the architecture of a GPU looks like with a lot of processing cores inside it, which are the SPs or the Scalar Processors. This is just to give you an idea that we can view a CPU, as a multi core platform containing a small number of compute codes, whereas a GPU, essentially contains thousands of compute codes, those simplistic in terms of functionality, but lots of them.

(Refer Slide Time: 02:07)

The slide title is "Course Organization". It contains a table with the following data:

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

At the bottom left is the text "GPU Architectures and Programming". At the bottom right is the name "Soumyajit Dey, Asst. Prof., IIT-B".

With this basic background we have already introduced the course organization, Out of which. Now we will get into the review of the basic computer organization architecture, which is the topic one. So let us get started with that.

(Refer Slide Time: 02:23)

The slide title is "The classic 5-stage RISC pipeline". Below the title, the text "Section 1" is visible. At the bottom left is the text "GPU Architectures and Programming". At the bottom right is the name "Soumyajit Dey, Asst. Prof., IIT-B".

So we will start with a brief overview of classic five stage RISC pipeline.

(Refer Slide Time: 02:33)

The classic 5 stage RISC pipeline

Instruction Level Parallelism

Basic RISC architecture

- ▶ The operation of a processor is characterized by a fetch⇒ decode⇒execute cycle.
- ▶ RISC n CISC ⇒ two different philosophies of computing hardware design
- ▶ RISC/CISC - Reduced/Complex Instruction Set Computing
- ▶ CISC approach - complete a task with as few instructions (instrs) as possible
- ▶ A CISC instruction : `MUL addr1 addr2 addr3`
- ▶ Equivalent RISC : `LOAD R2 addr2; LOAD R3 addr3; MUL R1 R2 R3; STORE addr1 R1`

GPU Architectures and Programming

Soummyit Dey, Asst. Prof., IIT Kharagpur

So, what is the RISC by the way. Now, RISC is more of a computing philosophy which evolved as separate philosophy. Then practice philosophy of CISC, which essentially means complex instruction set computing while RISC mean, reduced instruction set computing. So what really are these things. To get into that, let us just browse up on what does the processor really do.

A processor is a digital system, which is executing a few specific operations in the loop continuously. A processor crunches instructions, which we have been knowing from our basic background as assembly instructions which are getting assembled by assembler to a machine instructions. A processor processes machine instructions through a standard loop of fetching the instruction, decoding the meaning of the instruction and finally executing the instruction. And then again jumping to the next instruction. How a processor really goes about this business is where the computing philosophies differ. So fundamentally, this style of whether, to follow RISC style of computing or CISC style of computing is based on the basic notion of instruction set of a processor.

That is, what are the different types of machine level instructions, a processor is going to support. Are those instructions going to be very simplistic activities? Or are the instructions individually powerful enough to do a lot of things by just one specific instruction itself? So this is where this philosophies differ. That is why RISC is reduced instruction set computing and CISC means complex instruction set computing.

So coming to the CISC approach, the philosophy of CISC is essentially, that a processor should try and complete a processing task, using as few instructions as possible. That means, when a program is compiled to generate code for a CISC processor, one of the basic objectives of the code generation process would be to represent the program in terms of assembly instructions and thereafter in terms of machine instructions using as few instructions as possible. That means each of the instructions are going to do a whole lot of work. For example, let us take a CISC instruction. As we have given here that we have a multiplication operation MUL with three operators, address 1, address 2 and address 3. Essentially we are meaning that while executing this instruction, the processor shall fetch data from the memory address 2 and memory address 3, multiply them, and store the result back to memory address 1. Now of course, as we can see that this is quite a lot of jobs. It is going to load data from the main memory, then multiply those data points, and is going to store the result back to the main memory. All these are done by executing a single instruction. So in terms of hardware design that means we are thinking of a hardware which is complex enough to do all those things by executing a single instruction.

The RISC philosophy is diametrically opposite to this. So in an equivalent RISC world. We do not have this kind of a multiply instruction. In fact that does not get replaced by a sequence of these instructions. So there will be a load type instruction, using a load command. One should be able to load the data from memory address 2 to some register R2. This will be followed by another load command through which one should be able to load the data from address 3 to the register R3. And then we have a multiply instruction. So as you can see, this multiply instruction takes as arguments the contents of registers R2 and R3, multiplies them and stores the result in the register R1. Now this will be succeeded by the store instruction, which will store the result from R1 back to the memory address, address 1.

So this essentially summarizes the RISC philosophy that all kinds of operations specifically arithmetic logic operations will have to happen over data values that are present in the register, the result of such operations shall also be stored in registers. There is no operation that is going to happen on value stored in the memory. So, the only instructions which are going to write or read data from memory are the load and store instructions. So, this is one way to represent the

difference between a complex instruction, and the equivalent case of a sequence of simple instructions. So when we say that if the RISC processor is a processor where the instruction set comprises only this kind of very simple instructions. When we say it is a CISC processor, the instruction set is arbitrarily complex. There are instructions which are very powerful by themselves doing a lot of stuff.

(Refer Slide Time: 08:28)

The classic 5-stage RISC pipeline

CISC vs RISC

CISC features

- ▶ Older ISA
- ▶ Multi-cycle instructions, HW intensive design
- ▶ Efficient RAM usage
- ▶ Instructions - complex and variable length, lots of them
- ▶ Micro-code support
- ▶ Compound addressing modes

RISC features

- ▶ Ideas emerged in 1980s
- ▶ Single-cycle instructions, SW intensive design
- ▶ Heavy RAM usage, Large Register file
- ▶ Small no. of simple fixed length instructions
- ▶ Less no. of addressing modes

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor

Overall, to summarize the difference. The CISC features, the features of a CISC kind of system. The instruction set architecture of a CISC processor is older, the philosophy is quite old. So CISC processor came much earlier. RISC was an idea which emerged in the 80s.

Earlier, it was mostly CISC. Instructions used to be multi cycle. That means, the processor is driven by a master clock, and each instruction execution shall consume multiple clock cycles. Of course, because each instruction is going to do a lot of work. So it is a hardware intensive design. In a sense, it may be the case that for some powerful instructions, you need some specific hardware built into the system.

RISC feature is that all the instructions of single cycle. Every instruction will do something very simple, but at the same time it will consume a single clock cycle and get the job done. Which would mean for the same functionality the number of instructions in CISC is less, the number of instructions in RISC or more, but each of them, execute faster. So, overall, the timing for the final task may not differ much. This is how the system goes about achieving the job.

That is why I say, CISC is a hardware intensive design because there are maybe more specialized hardware for specific kind of instructions. And RISC is more of a software intensive design because I am breaking every task into small atomic instructions, which would mean for every task, I will generate a lot of instructions in software terms. Now since CISC I have much less number of instructions to execute, the RAM usage is much more efficient, whereas for the equivalent functionality since I have a big code to execute the RAM usage is much heavier. And also, since all RISC operations are to be carried over operands from the register file, specifically the arithmetic and logical operations. I need the support of a much larger register file, when compared with CISC.

Other important thing in CISC is the instructions are complex, they are multi cycle, which also means the number of cycles required for executing each of the instructions may vary. Whereas as we have discussed in RISC, all the instructions are single cycle. Now another very important thing is, since there is no regularity in terms of the intensiveness of CISC instructions, some instructions may be very complex, take multiple operands, take multiple cycles to execute. And due to its inherent complexity in terms of specification as well as number of operands, the instructions encoding length may be very big. Whereas for some other CISC instructions, the instructions encoding length may be small. Overall, the point is the CISC instructions may vary in length. Whereas for RISC, the instructions, all of them are very regular, they are going to be over the same fixed length. Because they are quite generic. They are all single cycle. They are intensiveness in terms of hardware research will be kind of similar. So they will all be of exactly the same opcode length.

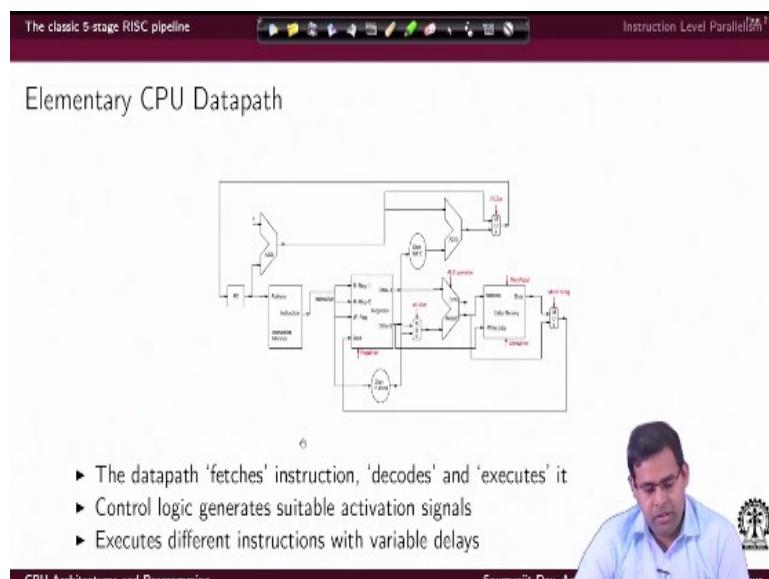
Another very important CISC feature is compound addressing modes. There will be a lot of support for different multiple addressing modes in a CISC ISA. ISA are instruction set architecture. Whereas in RISC, there will be much less number of addressing modes that will be supported.

To get into more details about these features is best that one should consult any well known undergraduate book on basic computer architecture. But these are the important points, which we wanted to summarize, as the difference between these two very well known computing styles

and their adoption in basic processor design. In modern days, possibly apart from x86 architecture, all other processors and more specifically in the embedded world, the processors used in mobile platforms they are mostly following their RISC computing philosophy.

In other terms, that means in the x86 world into Intel has still been able to hold on to the CISC philosophy, but internally, there is also a question of translation of these instructions. But that are advanced topics, which we may touch upon later on.

(Refer Slide Time: 13:40)



Just an example of a CPU data path, so what's the data path? This is basically a drawing, which is trying to summarize how the different functional units inside the processor, are going to be connected. As we can see, there is a block, which represents the instructions and their storage in memory. Now of course in a general processor instructions and data will be resident in the same memory for a Von Neumann architecture. This block highlights the register file, which will be containing all the CPU registers. And these are block, which is more representative of the ALU. This is kind of very simplistic CPU data path, which is trying to convey the basic operations that a CPU is going to execute. Like we have said, the CPUs data path is going to execute three basic operations for now- fetching the instruction, decoding the instruction and executing it. Now these are the functional operations, which means it will take one instruction, understand the semantic meaning of the instruction and do exactly what the instruction wants to do in terms of

functionality. But thereby there is also the question of reading and writing operands from the memory, as well as updating values and registers.

Now, these will also give rise to some more functions which we will see soon. Now, coming back to the question of data path. So, the data path, essentially represents the flow of data among different functional units as captured in this picture. Like how exactly the data flows from each of the functional units, what are their dependencies, and the more important thing is, how is the flow of data control.

Now, you can see that there are certain signal lines here, which are marked in red. Now these are the lines which are expecting some digital comments, based on which they decide how this digital system would operate. Fundamentally speaking that data flowing in from a hardware unit should be used for what kind of functionality inside the hardware unit, and should be routed as an output to which hardware unit. This is what is dictated by these signals marked in red, and they are known as the control signals. The control signals get generated through a separate logic called the control logic.

(Refer Slide Time: 16:31)

The slide has a dark header bar with the text 'The classic 5-stage RISC pipeline' on the left and 'Instruction Level Parallelism' on the right. Below the header, the title 'Single cycle implementation of datapath' is centered. To the right of the title, there is a small portrait of a man with glasses and a blue shirt. At the bottom of the slide, there is a footer bar with the text 'GPU Architectures and Programming' on the left and 'Soumyajit Dey, Asst. Prof.' on the right, along with the IIT-B logo.

- ▶ The choice of clock rate is limited by the instruction with maximum delay
- ▶ Options : choose the clock period more than latency of 'slowest' instruction or,
 - ▶ choose variable periods for diff instructions – not practical !
- ▶ Alternate possibility - break the instruction execution cycle into a series of basic steps
 - ▶ Basic steps have less delay, choose a fast clock and use it to execute one basic step at a time

Now as we have discussed earlier, that in RISC, the philosophies that every instruction executes in a single cycle. Now, why is that a good thing. Now let us note the important thing that even for RISC style instructions, each instruction may have different timing delay when the

functionality is implemented in terms of hardware. That is why we can say that instructions execute with variable delays in a CPU data path.

Now, the question is, when I have implemented such a data path in hardware. These are digital data path. It needs to be clocked by a suitable clock signal. And I have to choose a suitable frequency or period of the clock. Now, as in any standard digital design, the clock period is chosen by looking at the critical path of the circuit, ie.what is the situation in which the data path, or this specific digital circuit will incur the maximum delay.

As we have said that even for instructions which are all lightweight and atomic as in RISC. Their executions may create different possible delays in data path . So then does it mean, the clock signals frequency should be different for different execution of instructions? That is definitely not possible. That means we have to find out execution of which instruction takes the most amount of time and choose the clock period to be equal to or greater than the latency of that slowest instruction. Now, even if we do that, we still have a problem. Because then we may have a system where I have multiple possible instructions to execute. Some of them are very slow, in terms of hardware latency, some of their executions is very fast in terms of hardware latency. But since I have chosen the clock period, so that it can execute the slowest instruction. Even for the faster instructions and their execution, I suffered the delay of the slowest instruction. So that leads to a very conservative design. So how do I get out of this problem and increase a processors', or CPU data path throughput. The way to go about it to be that look at the instruction execution path. What are the basic functionalities that each instruction is supposedly going to do, and break them into a set of basic operations or basic steps.

Now why do we do that, because if I can break this execution of an instruction into certain basic steps. With respect to the flow of data and control and processing in this data path. Then I can now clock each of these basic steps by a fast clock. How does it help the basic steps are simple, so they can be clocked by a fast clock. For some instructions I may need the sequence of all the basic steps, for some instructions I may not need them. Which means, now you have this system, driven by a fast clock with some instructions completing fast and some instructions completing slow, based on what is the requirement of the instruction in terms of these basic steps.

(Refer Slide Time: 20:32)

The classic 5 stage RISC pipeline

Instruction Level Parallel Processing

Multi-cycle instructions

A basic stage represents one of the following states in the execution of an instruction

- ▶ Fetch (IF): $IR \leftarrow \text{Memory}[PC]$; $PC=PC+4$
- ▶ Decode (ID): Understand instruction semantics
- ▶ Execute (EX): based on instruction type
 - ▶ Arithmetic/logical operation, Mem address / Branch condition computation
- ▶ Memory (MEM): For load/store Instr, read/write data from/to memory
- ▶ Writeback (WB): Update register file

Soumyajit Dey, Associate Professor, IIT Kharagpur

GPU Architectures and Programming

Now, if we get into this issue of identifying the basic steps required for executing an instruction. Earlier we have been telling like an instruction needs to be brought from the memory and then decoded by the decoder and finally executing the instruction.

At the same time we also said, Okay, this is fine, but there is also this issue of bringing data from the memory, updating the data in the register file and all that. So bringing all these issues together, we can say that the execution of any instruction in a RISC kind of system can be broken down into these five basic steps.

The first stage would be, the instruction fetch stage where we check the content in the memory for the location that is loaded into the program counter mark here by this PC. And load the content into this specific register, which is called IR, the instruction register. So, I hope everybody will be familiar with this notion of problem counter which is basically a specific register in your CPU which loads specific memory address from which you are supposed to fetch the instructions and execute them. So this is noted down here in terms of the standard mnemonics that you can find, that the content of the memory location that is pointed to by the program counter is getting loaded into the instruction register, followed by the program counter getting implemented by 4, Why 4? Essentially we are saying thinking that memory is byte addressable, and the memory word length are 32 bits. Assuming that we should get the next program counter value by incrementing the current program counter value by 4.

Now what is the next stage. So the next stage would be this decode stage. That means an instruction has been fetched. An instruction is nothing but a sequence of 1s and 0s. In our assumption here it is a 32 bit instruction and this sequence of 1s and 0s need to be understood by the CPU. So there is this decode stage, where based on the instructions encoding and its operands, the CPU or the data paths should understand what does this instruction, want to do. Is it a load instruction, or stored instruction or an add instruction, what does it really want to do. Based on this understanding, it should activate suitable functional units in the next stage ie. the execute stage, and it should perform the required operation, depending on whether it is an arithmetic or logical operation, whether it is the computation of a memory address for a branch, or whether it is the computation for a branch condition itself. So these are the computational requirements, which, if there is any will be computed in the execute stage.

Now this execute stage has to be followed by another stage called the memory stage. What is this for? In case the instruction is a load store type instruction. It needs to fetch data from a memory location, and that data has to be loaded into the register file. So this fetching of data from the memory will be performed in this memory stage. So, basically this is the stage, which is going to handle all read, writes from and to the memory.

Now we have the last stage in the pipeline, which is called the write-back stage. So, any kind of update that is going to happen on the register file has to happen here. For example, if I take the example of a load instruction. So suppose I am loading the value from some memory location x to some register R1, the access of memory location x and fetching the data from that location to some specific register called memory data register should happen in the MEM stage. And then writing this data from the memory data register to the specific register R1 in the register file, this update of the register file shall happen in the write-back stage. So these are the five stages, which, more or less characterize the execution sequence to be followed in a basic RISC processor. That is why we call it the classic five stages RISC pipeline.

Now of course, in a real processor. There are much, much more complex functionalities present in a basic RISC processor. There may be many more pipeline stages present there. But, for our

purpose, we stick to this discussion of a pipeline with our understanding of this basic five stages of execution.

(Refer Slide Time: 26:01)

The classic 5-stage RISC pipeline

Pipelining

- ▶ Operate IF→ID→EX→MEM→WB in parallel for a sequence of instructions
- ▶ Every basic stage is always processing some instruction
- ▶ In every clock cycle, one instruction completes - ideal scenario
- ▶ Practical issues - pipeline hazards

GPU Architectures and Programming

Soumyajit Dey, Asst. Prof.

So, overall when this pipeline operates, every instruction has to go through the sequence of five pipeline stages. From instruction fetch, instruction decode, execute, memory and write-back. It depends whether some of the instructions will really need something significant to be done in that stage or not. But all these stages will be active. Since the functionality of each of these stages, is very basic, there can be a faster clock clocking all these stages. We will have a fast execution of all the instructions, whether or not it really needs executing in that stage or not.

Now, once we pipeline the operations in this kind of a processor I can always have these basic stages, always processing some instructions, which means, suppose I have instruction 1 entering the pipeline that is it has been fetched. And it has been decoded, so when the instruction 1 is going to the decode stage, I can have instruction 2 being fetched. And when instruction 1 is getting executed instruction 2 can get decoded and instruction 3 can be fetched. So that would mean, if I look at the pipeline from its endpoint. I always see one instruction, getting completed. In that way, I can say that, in this pipeline one instruction getting executed in each clock cycle ie. one instruction is getting finished wrt execution in each of the clock cycles. So this is where the optimization really helps.

To summarize, from the earlier points. Had we been sticking with a single cycle implementation of our CPU data path. The single cycle implementation would have required different amount of time for executing each of the instructions depending on instructions functionality. In that case the choice of clock will be based on what is the slowest instruction. And then what would happen is the lighter instructions will suffer the same latency, as the slowest instructions. In order to alleviate that issue we broke the execution of instructions into these kind of basic stages. Since each of the basic stages are very simple, we are able to clock the five pipeline, with a very first clock.

Since that is the case, and I can keep each of these stages always active, I can keep on filling instructions at the front of the pipeline. And I can see instructions getting completed at the end of the pipeline, one in every clock cycle. So overall, using this idea of pipelining, starting from multi cycle execution of instructions, we are able to see, the pipeline is able to complete one instruction per clock cycle.

So in that way I can say that every instruction requires 1 clock cycle. But that is not really the case right every instruction is requesting 5 clock cycles with respect to the basic stages. But since the pipeline is always active, it is interleaving the insert, execution of instructions. I can say that finally from a user's point of view, every RISC instruction when it executes in pipeline, I have one instruction completing in one clock cycle.

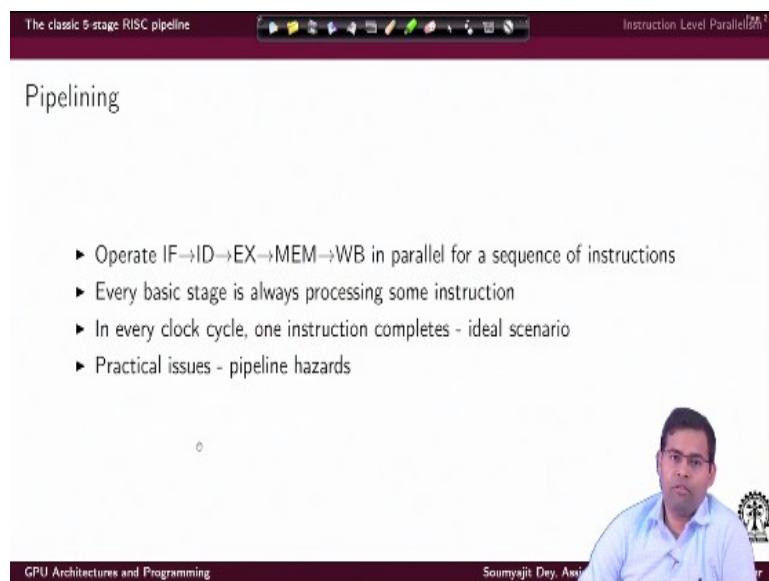
Now, this would really be the case in an ideal scenario. So what is a non ideal scenario? Non ideal scenario means that while executing an instruction in a stage of the pipeline, it may so happen that immediately the instruction is not able to execute in the next clock cycle in the next stage of the pipeline. These are the issues known as pipeline hazards, which we will be covering.

GPU Architectures and Programming
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 2
Review of Basic COA w.r.t. Performance (contd.)

Hi, so we have been discussing about the different stages, present in a basic RISC pipeline. Why pipelining really helps? And when I have a pipeline implementation. Although each of the instructions are taking multiple cycles to complete. Based on the cycle being defined with respect to each of the pipeline stages, but by looking at the end of the pipeline, it will seem that in every cycle have one instruction completing. That means for each instruction the execution latency is one cycle.

(Refer Slide Time: 01:01)



The slide has a dark header bar with the text 'The classic 5-stage RISC pipeline' on the left and 'Instruction Level Parallelism' on the right. Below the header is a title 'Pipelining'. The main content area contains a bulleted list of points about pipelining:

- ▶ Operate IF→ID→EX→MEM→WB in parallel for a sequence of instructions
- ▶ Every basic stage is always processing some instruction
- ▶ In every clock cycle, one instruction completes - ideal scenario
- ▶ Practical issues - pipeline hazards

At the bottom of the slide, there is a footer bar with the text 'GPU Architectures and Programming' on the left and 'Soumyajit Dey, Asst. Prof.' on the right, along with the IIT Kharagpur logo.

Now, as we also discussed that, this is an ideal scenario where it is being assumed that the instruction when it's moving from one stage of the pipeline to the succeeding stage, the movement is seamless ie. there is no issue of delay or anything, but that really doesn't happen. And these are the issues known as pipeline hazards.

(Refer Slide Time: 01:32)

The slide title is "Structural hazard". The content discusses a sequence of 4 lw (load-word) instructions where the fourth instruction depends on the first, creating a structural hazard due to lack of resources if hardware cannot support multiple reads in parallel. A video player interface at the bottom shows the speaker's name is Soumyajit Dey, and the course is GPU Architectures and Programming.

- ▶ Consider a sequence of 4 lw (load-word) instructions
- ▶ When the first instruction fetches data from memory, the fourth instruction itself is to be fetched from memory
- ▶ This is *structural hazard* as the pipeline needs to stall due to lack of resources, if the hardware cannot support multiple reads in parallel

So let us look at a few possible, such hazards. And in a practical pipeline why such hazards actually happened. Such hazards can be classified into several types. One of them is structural hazard. So let us consider a sequence of four load instructions in MIPS. We have them as lw load- word instructions. Let us say I am loading data from memory for four consecutive loads. So when the first instruction fetches data from memory. That means, it is in which stage of the execution?

(Refer Slide Time: 02:06)

The slide title is "Pipelining". The content explains that operations like IF→ID→EX→MEM→WB are performed in parallel for a sequence of instructions, with each stage processing one instruction per clock cycle, leading to practical issues like pipeline hazards. A video player interface at the bottom shows the speaker's name is Soumyajit Dey, and the course is GPU Architectures and Programming.

- ▶ Operate IF→ID→EX→MEM→WB in parallel for a sequence of instructions
- ▶ Every basic stage is always processing some instruction
- ▶ In every clock cycle, one instruction completes - ideal scenario
- ▶ Practical issues - pipeline hazards

Going back, the first instruction is fetching data from memory in the fourth stage of its execution. That means the second instruction is in EX stage, third instruction is an ID stage,

fourth instruction is in the instruction fetch stage. So that would mean, when the first instruction is fetching data from memory, the fourth instruction itself is to be fetched from memory.

Considering a memory element, where I do not have the facility of parallel reads, even from different locations. This is a situation where the hardware does not have support in terms of resources, because both the instruction, and the data may be loaded into the same memory element and they cannot be read together. So this is what we call a structural hazard.

It's a hazard where the pipeline needs to stall before fetching the fourth instruction, and it needs to complete that data read for the first instruction in its memory stage. And the reason for the stall is lack of resources, because I have only one memory element as a resource, and I am assuming that although the data is located at different locations in the memory as there is no multi port support.

(Refer Slide Time: 03:30)

The classic 5-stage RISC pipeline

Instruction Level Parallelism

Data Hazard : MIPS example

- ▶ sub \$2, \$1, \$3; and \$12, \$2, \$5 Read after Write (RAW)
- ▶ if 'sub' is in IF stage in $i + 1$ -th clock cycle, \$2 is updated in $(i + 5)$ -th cycle
- ▶ 'and' is in EX stage in $i + 4$ -th cycle, updated value of \$2 is not yet ready
- ▶ Solution : 'sub' computes the value for \$2 in $(i + 3)$ -th stage,
- ▶ this may be *forwarded* directly to execution of 'and'
- ▶ need suitable logic to detect hazard and forwarding requirement

GPU Architectures and Programming

Soumyajit Dey, Aw

Consider another example of another kind of hazard, which is data hazard. So what happens in a data hazard? Let us have this example of two instructions, instruction for subtraction and 'and' operation. So, I have the different registers enumerated as 1, 2, 3, and the instruction encoding is such that here representing the registers by \$1 \$2 \$3 like a standard for MIPS instructions. So, this is a subtraction instruction.

The idea is that you want to subtract the content of 3rd register from the content the 1st register and store the result in register 2. This instruction is followed by and operation, where the content of register, 2 and 5 will be ‘and’-ed. And the result is to be stored in register 12. Now, this is called data hazard, because we have what we call popularly as a read after write (RAW) dependency, let us look into it in, with more attention now.

So let us consider that this ‘sub’ instruction has entered the pipeline, that means it is in the IF stage of the pipeline in some $i+1$ -th clock cycle. So, if it's in the IF stage in the $i+1$ th cycle, then it moves to the decode stage in $i+2$ -th clock cycle, it moves to the execute stage in the $i+3$ -th clock cycle, and that would mean. After the execution of the instruction, the content of this register 2 is getting updated in the $i+5$ -th clock cycle.

Because $i+1$ instruction fetch, $i+2$ instruction decode, $i+3$ subtraction instruction executed. Then $i+4$ is a memory stage, and then I have right back happening in the $i+5$ -th cycle. And that is when \$2 is getting updated. But what is the situation with the ‘and’ instruction. So when the ‘sub’ instruction is in its $i+5$ -th cycle. The ‘and’ instruction in an ideal scenario will just be following it. That means, in the $i+4$ -th cycle ‘sub’ was in the memory stage. And in the same $i+4$ -th cycle, ‘and’ is in the execute stage. That means, ‘and’ demands a value, a proper value in the register 2 in $i+4$ -th cycle. But when is register 2 getting updated by ‘sub’? It is getting updated in the $i+5$ -th cycle, which means if ‘sub’ and ‘and’ are executing in the pipeline one after another, and following sub exactly with no intermediate lag among them in terms of execution stages. Then, ‘and’ requires an updated value in $i+4$ -th cycle, and the value is not ready, it is supposed to get ready in the $i+5$ -th cycle. So, that is a hazard because in that case, I should not be able to execute, and without any delay, getting inserted in the pipeline. Unless I add in some extra hardware support.

What can be the solution in this case? The solution is, observe the scenario such that although the instruction ‘sub’ updates the content of register 2 in the $i+5$ -th cycle, when does it get computed. It gets computed in the execute stage for instruction ‘sub’ which is the $i+3$ -rd cycle. So, the value is actually ready, it is just not updated.

So what if we have some extra hardware support such that, whenever the value is ready. It can immediately be forwarded wherever it is required for execution. Without the value getting transmitted to the register file, and then getting used from the register fight. So these are called forwarding units, which may be present inside a pipeline for resolving these kinds of dependencies. But then again there is the issue that hardware ie. the CPU data path should have support in terms of detecting these kind of hazards that means while executing these instructions sequence. If I am using a CPU data path, the simplistic one I showed earlier, it will not be able to do this. The hardware should be able to look into this instruction sequence and identify that this instruction sequence has a read after write(RAW) dependency. So this can be modeled by a formal condition and that condition needs to be checked by the hardware. And when the chip details his dependency, the forwarding units should be activated accordingly. As we got to understand, these are the ways in which data hazards can be detected and suitably handled inside the pipeline.

(Refer Slide Time: 09:01)

The slide title is 'The classic 5-stage RISC pipeline' and 'Instruction Level Parallelism'. The main heading is 'Control hazards'. The bullet points are:

- ▶ Branch decisions : the branch condition needs evaluation (beq \$1, \$2, offset)
- ▶ The branch decision is inferred only in MEM stage
- ▶ Optimization : assume branch not taken, operate pipeline *normally*,
- ▶ Execute branch when decision is evaluated as true (taken) and flush intermediate instructions from pipeline
- ▶ Sophisticated schemes : use branch prediction HW (predict a branch decision based on branch history table content)

A video player at the bottom shows a speaker named Soumyajit Dey, A.I.T.B.S., with the text 'GPU Architectures and Programming'.

An example of other kinds of hazards, for example control hazards. So what's that? Take the execution example of a branch decision. We have branch instructions, and they need to be supported by any processor Why? Because, think of the situation that you have written a problem in C language, definitely there are changes in execution flows, which you model by writing code using IF ELSE blocks.

Now, those will be translated to branch conditions or branch instructions when it gets translated to machine code. So, definitely, any CPU has to support branches instructions. So, if we take the example of this branch instruction `beq $1, $2, offset` (`beq` is branch if equal). So, in case the content of the registers is found to be equal, then the branch has to be taken. So, this condition of branching has to be evaluated.

So when this branch instruction is getting executed where will this condition be evaluated, it will be evaluated in the execute unit. So the result of this branch conditions evaluation is inferred only when the instruction is in the MEM stage. So, up to this point, what shall be going on for the instructions, after this branch instruction. Because as we remember, our standard pipelining approach has been you execute one instruction in every cycle. You provide the pipeline with one instruction in every cycle. Following that philosophy. When the branch instruction goes from fetch to decode that means when the meaning of branch is getting decoded the instruction fetch has to happen, the IF stage should be fetching the next instruction. When the branch condition is getting evaluated the IF stage should be fetching another instruction. And when the branch conditions meaning has been inferred and it's available in the MEM stage, the IF stage should be fetching MLF the third instruction. But then only after the branch decision has been inferred it is known whether to follow the current sequence of instructions. or whether to abort it and move the PC content to some updated address and accordingly fetch instructions from the new branch of execution.

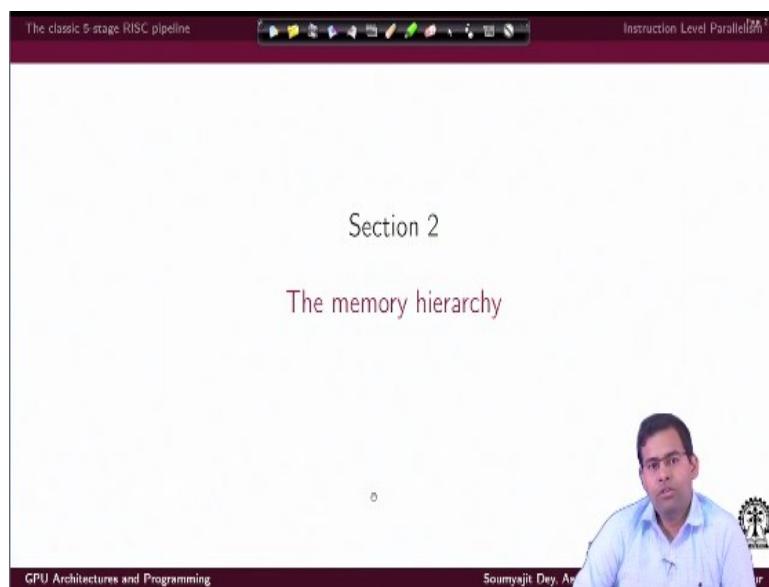
Now, why we call it a hazard? Because just after fetching the instruction is really not alone. Now what can be an obvious solution in this case? You execute the branch when the decision is evaluated as true and flush intermediate instructions from the pipeline. That means you simply keep on feeding the pipeline with instructions. However, when the decision is found to be true that means the branch has to be taken. That means whatever further instructions you have sent into the pipeline, their execution doesn't have any role. Then you flush the pipeline, and then you move to the new address, and then start executing instructions from that address- the branch address.

But is that a good way to go about handling branch? Maybe not. Why? Because statistically speaking there's a 50-50 chance, and I am not considering facts like for what kind of program we are talking about and I am not modeling the branch decision, and just thinking it is fair enough to keep on executing instructions one after another. And only when the decision is known, then I flush the pipeline and go to the branch address.

That is why there are sophisticated statistical schemes implemented in hardware in modern CPUs. So they have this specific kind of hardware called branch predictors. So branch predictor is a lightweight hardware, which could contain something like a branch history table. That means, what were the branch decisions in a few of the earlier branches. And based on that, it will try to predict whether to take or not take a current branch.

So then, it's not as naive as the previous approach. Its banking on statistics of previous branch history and trying to take an informed decision based on that. However, again if it's found that when the branch condition is evaluated and is something different from whatever was predicted, then the pipeline flushing operation is definitely required.

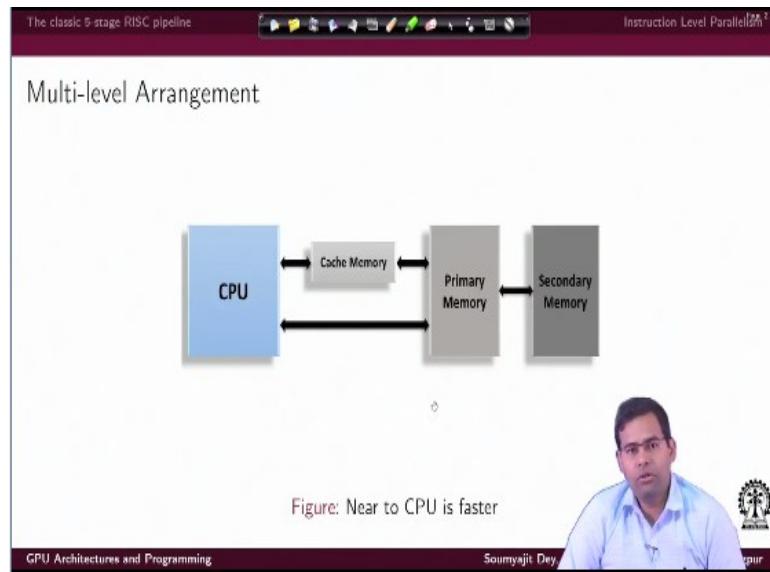
(Refer Slide Time: 14:09)



So that is all about very brief overview of pipelining, and how the pipelining deviates in several of the non-ideal scenarios. Some of those scenarios can be handled. We feel that this is small but

necessary overview, which will be helpful in going further into the GPU part but before that there is something we need to discuss about how memory is organized, in a standard CPU.

(Refer Slide Time: 14:42)



So, the instructions that the CPU will be crunching they are fetch from a primary memory or main memory that is known to us, more like a RAM. And then this primary memory will be connected to a secondary memory like disk drive. In between the CPU and the primary memory, we have something called a cache memory. So these are the three primary levels of memory, which defines the storage from which instructions and data are fetched and executed by the CPU as and when required. Now, the memory segment which is near to the CPU is faster. Cache memory is not a discrete thing, it is something which is on chip with the CPU.

(Refer Slide Time: 15:33)

The classic 5 stage RISC pipeline

Instruction Level Parallelism

Principle of locality

- ▶ Temporal locality : If an item is referenced, it will tend to be referenced again soon
- ▶ Spatial locality : If an item is referenced, items at nearby addresses will be referenced soon
- ▶ Hence, computer memory is hierarchically organized
- ▶ Register file provides fastest access,
- ▶ Cache memory uses (fast) SRAM (static random access memory)
- ▶ Main memory uses (slow) DRAM (dynamic random access memory) : is less costly per bit than SRAM

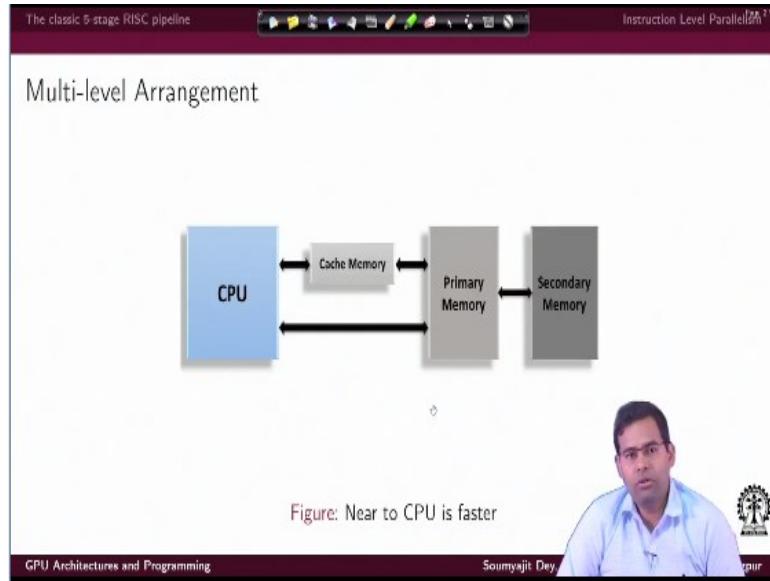
GPU Architectures and Programming

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, why really is cache memory used? The specific reason between them comes from the principle of locality. So what is this principle of locality? There can be two kinds of localities - temporal locality as well as spatial locality. So what is temporal locality? If a program references an item, let us say a variable or an array location. Then, there is a high chance that that item will be referenced again soon by the program. This is temporal locality with respect to time.

Similarly, we have the concept of spatial locality. Which would mean that if an item is referenced, let us say an array content, A[1], items which are located at nearby addresses in the memory are likely to be referenced soon. So if a program P refers that means loads the content of an array A, let us say a1 is highly likely the content of A[2], A[3], those will also be referenced in very near future. So this notion of spatial locality and they hold true for most of our programs. Based on that, computer memory gets hierarchically organized as we have told this. Let's go back to the picture.

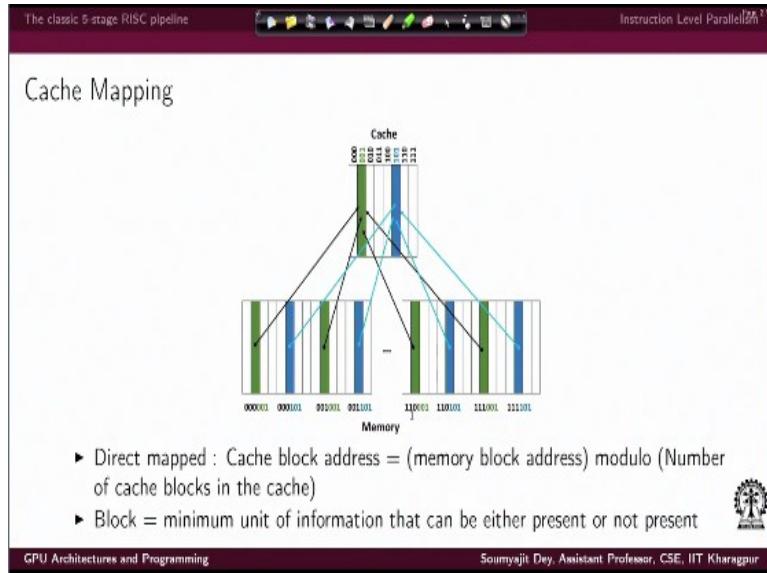
(Refer Slide Time: 17:01)



The memory element line nearest to the CPU is going to be the fastest. The one that is going to be further down on the right hand side will be slower. So cache memory will be some kind of memory technology which is faster with respect to primary or main memory, and the secondary memory would be slower with respect to this primary memory. But also, as we go right hand side, the speed decreases while the size increases. The fastest memory element, to be more specific would be the CPU registers. Because they are sitting really near to the arithmetic logic units ie. the main execution units. So in that way the register file provides the fastest access to the data, you just load from the register file to the ALU, and do some computation. The cache memory uses the SRAM or static random access memory technology. So SRAM technology is fast, but costly also.

The main memory uses comparatively slower DRAM technology, which is less costly per bit than SRAM. That is why I can have a big main memory, but I cannot have a big cache memory. It is fast, but costly. The main memory is slow, but I can have a much bigger memory.

(Refer Slide Time: 18:22)



Now the question is, how really do we decide that okay, I am bringing in data from main memory location, I am going to load it in the cache memory, and then I will load it in the corresponding register, do some computation. And all that, but where really should I load the data in the cache memory? That gives rise to the problem of what we call us cache mapping.

So what's cache mapping? It is basically a set of rules that will tell you that okay if you bring data from some memory location, m_i , what should be the location in the cache, where that data has to be stored. Now of course, there can be different policies or different functions which dictate the mapping. We will discuss the popular ones, and their relative advantages and disadvantages. So, let us start with one of the strategies. So, there can be a direct map cache mapping.

So as we discussed the the cache is a much smaller memory segment with respect to the main memory. So here in this picture. We have the main memory, these kind of again representative from the well known book on Harrison Patterson. So this is the main memory, we are trying to show the locations of the main memory. And we have a cache which is a much smaller memory, the addresses in the cache are kind of marked here as consecutive addresses, starting from 000 to 111. Every location in this cache we are calling it as a cache block. So what's the block? A block is defined as a minimum unit of information that can be either present or not present.

Now, what does this really mean? Earlier we have talked about the concept of memory word. A word means, the amount of data that may be stored together in a register or the amount of data that can be passed on the CPUs bus.

For example, if I am talking about a 32 bit CPU that means the memory word length is 32. The register file is containing registers of 32 bit size. Memory becomes byte addressable that is why when we increment the program counter I do a PC to PC+4. I increment by 4 bytes, I go to the next memory word, which is the next consecutive 32 bits in the memory. Now when we talk about cache it's not really the case that when I read data from memory, I will read 1 memory word and put 1 memory word in the cache, I may read more than that, I may read, let us say 4 consecutive memory words. And put them in the cache. So this is what is defined as the cache block size. So then I would start saying that okay, this is the minimum amount of information that I would read from the memory put in the cache. Or I will, I will update that cache location with some other locational data from the memory.

So then what is the addressing scheme of the cache. So the address of a cache block is given by, suppose I am trying to load data from the memory. I define a memory blocks address. So that is from where I am starting to load data. Let us say I am loading four consecutive cache memory words, the block size is four. So the address of the memory block modulo(%) the number of cache blocks.

So just as an example here in this picture, how many cache blocks are there? So I start from 000 to 111, so I have 8 cache blocks. That means, from every memory location, you do modulo 8 operation, you take the address of every memory any local memory location, you do a modulo 8 operation. Whatever you get is the location in the cache, where the data from this memory block will get loaded, so that is the idea of a direct mapping. For every block in memory, you have a corresponding unique mapping to some location of the cache, some unique block in the cache.

(Refer Slide Time: 22:57)

The classic 5-stage RISC pipeline

Instruction Level Parallelism

Cache Blocks

- ▶ With larger blocks we have lower miss rates due to spatial locality, large blocks lead to large miss penalty
- ▶ Nothing is free : with very big block sizes, we have too small no of blocks in cache, eventually the miss rate goes ~~down~~ up
- ▶ Handling Cache Miss:
 - ▶ Send the PC value (current PC - 4) to the memory
 - ▶ Read access from main memory, write updated cache entry

0

Somnajit Dey, Asst. Prof.

Now, coming to this idea of cache blocks, how to decide what should be the size of a cache block. As we say that it may not make sense that you load only one word from the memory, because you may have a wider access to the memory. So, you may load multiple consecutive words and that divides the cache block size or the memory block size.

Now then the question is, how large should be the block? And what is the impact of the block size? If I have large blocks in the cache. That means whenever I read data from the memory to the cache, I read more amount of consecutive data, more amount of consecutive memory words and store it in a cache block. That is what I mean by a large block size. Then we have lower misread due to special locality. Why is that? Because, as we said the principle of special locality sets. Let us go back.

(Refer Slide Time: 24:02)

The classic 5 stage RISC pipeline

Instruction Level Parallelism

Principle of locality

- ▶ Temporal locality : If an item is referenced, it will tend to be referenced again soon
- ▶ Spatial locality : If an item is referenced, items at nearby addresses will be referenced soon
- ▶ Hence, computer memory is hierarchically organized
- ▶ Register file provides fastest access,
- ▶ Cache memory uses (fast) SRAM (static random access memory)
- ▶ Main memory uses (slow) DRAM (dynamic random access memory) : is less costly per bit than SRAM

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

If an item is referenced items that nearby addresses will be referenced soon. Right. So then, in case I have loaded a specific data element from memory. The block size is large. There is a good chance that many of the consecutive memory locations, which are getting loaded. Since the block size is large, I have loaded many such consecutive locations. And some of them would get referenced soon. And they are already in the cache.

So, I will have a lower miss rate due to this special locality. But then what is the overhead. The overhead is, since I am loading data in large blocks I am reading a lot of data together. So, if there is a miss, the miss penalty is large, in case of a miss, I have to do a memory read and memory read will take a lot of time. However, I cannot keep on increasing the block size, because then at some point of time, the miss read will not keep on going, it will not be the case that the miss read is low, but I will have the situation that the miss read goes up. So with very large block size, we have two small number of blocks, and eventually the miss rate goes up,

Sorry, here. maybe we should do a correction. Yeah. So that is it. This should be.

Now, how are things taken care of, when there is a cache miss. So of course, in case there is a cache miss that means of value has been referenced by the CPU, the first place to search for that memory address should be in the cache. If found that's okay content of that address is not loaded. So then the location of the PC has to be updated by current PC -4. Why because they

instruct by the time this is in getting done, the instruction count has already gone up. So that is why you do a current PC -4, and you send that value to the memory right?

So that is how you handle a cache miss. Basically you send the PC value, the program counter value to the memory, and you do a read access from the main memory, and after doing the read access using the corresponding mapping scheme. You find out what is the cache block location, and, accordingly you update the cache.

(Refer Slide Time: 26:57)

The classic 5-stage RISC pipeline Instruction Level Parallelism

Cache write policy

- ▶ Handling consistency : always write the data into both the memory and the cache (write-through)
- ▶ Conservative policy, slows things down
- ▶ Use *write buffer* to perform writes only when buffer is full. Buffer size can be decided by memory speed
- ▶ Alternative policy *write-back* : Writes are updated only in cache. Main memory is updated only during cache block replacement
- ▶ Write-back offers better performance in case of frequent writes, is more complex to implement

GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So an important issue, apart from deciding the mapping of data to the cache is how really the cache should handle variable updates. That is what should be the policy based on which cache data should be updated back to the memory. Now why is this important, because you bring data from the main memory to the cache, you load the data to the registers. You do some operation you update the register content, which now need to be return back to the cache. And further back to corresponding memory location.

So when do we update the memory location, after updating the cache. Now one simple policy can be write-through, that means, always write the data into both the memory and the cache, so whenever I update our cache location, I also update the corresponding location in the main memory.

Now, is it good? Not really, because it's the conservative policy. Because then every time I do a right on the cache. It's accompanied by a right on the main memory it's going to slow things down. Alternatively, you have a separate write buffer, so that in the write buffer you sequence pending memory updates. So by default you have update cache. And in the write buffer you sequence these pending memory updates for the main memory, and when the buffer is full. Then you carry on the updates.

What should be the buffer size can be decided by the memory speed, right? Because if the memory has a high latency The buffer needs to be big to hide the latency. An alternate more clever policy will be write-back. That means, by default, whenever a program is updating variables you update in the cache. Now you update in the main memory. Only when that specific cache block is replaced.

Why is this good? It's good because whenever the variable will be referred in the future. You will anyway refer from the cache first, where the variable value is already updated.

The issue will come only when some other cache load is going to override this data, that means this current updated cache block is going to be replaced. Then this update needs to be transmitted back to the main memory.

So this is like doing a lazy right. Doing it as and where it's needed, and the need actually comes with the cache blocks gets replaced. This gives definitely better performance because it's not conservative, more so in case of frequent writes, because then you do not write back to the memory that often, but the bad thing is, it's more complex to implement. The hardware needs to do a lot of stuff so it's more complex to implement.

(Refer Slide Time: 29:44)

The classic 5 stage RISC pipeline

Instruction Level Parallelism

Memory System

- ▶ Memory chips are designed to read/write more than one word in parallel (hiding latency)
- ▶ Use a wide bus - allow parallel access to all words in a block
- ▶ OR - keep bus of standard width (= memory word length = register size) and connect bus with multiple memory units in parallel (memory banks)
- ▶ WHY ? bus transmission is fast, memory read/write is slow

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor

A photograph of Soumyajit Dey, an Assistant Professor, speaking. He is wearing a light blue shirt and glasses. The background shows a university logo.

Now, with respect to the memory system. How is this issue of reading and writing data from the memory, arranged. Now, memory chips are fundamentally designed to read write, more than one word in parallel. Why? Of course you want to hide the latency. Cache is fast. The main memory is slow. So that is why whenever you transact with the main memory, you like to transact more number of things in parallel.

So that whenever you do an access to the main memory, you bring more things, or you write-back more things. Now, how can this really be done in the physical hardware? One option is , you use a wide bus. That means you are doing parallel access to multiple words in a memory. let us say you are accessing multiple words, all of the words in a block in parallel, but then you need the bus to be wider. Maybe you have a 32 bit system. But the bus is not 32 bit in this scheme you require a wider bus.

Alternatively, you can keep the bus of standard width, which is equal to the memory word length or equivalent to the register size. But then, although the bus is of standard width, you connect the bus with multiple memory units in parallel. That is the memory bank, the memory chip is not one single chip. There are multiple small memory chips together.

And read and write operations can be done in parallel, from all these different memory banks. Why is this a good optimization? The reason is, when we talk about memory access, we

talked about the total time. That means accessing the data from the memory, reading or writing, as well as transmission through the bus. The bus is fast. The issue is more with the memories read and write, which is slow. That is why it may make sense to keep the bus of standard width and creating banks in the memory so that I can read or write in parallel.

(Refer Slide Time: 31:46)

The classic 5 stage RISC pipeline

Instruction Level Parallelism

Cache Mapping: alternate schemes

- ▶ Fully associative: a block can be placed in any location in the cache. (Large HW requirement for fast parallel search)
- ▶ Practical only for cache with small number of blocks
- ▶ Optimizing in the middle : set associative cache
- ▶ An n -way set-associative cache consists of a number of sets, each of which consists of n blocks.
- ▶ Set number = (Memory Block number) modulo (Number of sets in the cache)
- ▶ Inside a set, all the tags of all the elements must be searched
- ▶ Increasing associativity decreases miss rate up to a point, but increases

Soumyajit Dey, A

Now coming back to the cache mapping. We discussed how cache mapping can be done in a direct map way. There can be alternate schemes, for example, instead of doing a direct map. I can do a fully associative mapping of a cache. So what's that? A fully associative mapping would mean any block of data from the main memory can be placed in any location in the cache. If you remember in direct map. We were doing a modulo number of cache blocks operation and deciding what should be the location in the cache, where a memory block gets mapped. So in that way for every memory block there was a unique location of the cache block.

Coming to fully associative, we simply say that no you can load anything anywhere in terms of blocks. But why is that a bad thing because then when you were referencing data from a cache, you have to search all the cache locations. Because there is no mapping, there is no mapping that looks at this memory address. If it is at all present in the cache. It should be located exactly in this location, it's not so. So you need a large hardware for doing the parallel search. that is why it

may not be practical. It is practical only for caches with very small number of blocks. If you consider a big cache pool is it is a bad scheme.

So, we can optimize in the middle - between these two extremities of direct map and fully associative, we go for an in the middle approach we decide decide that let us say that mapping scheme can be what we call a set associative cache. Lets define a set of associative cache. We say a cache is in with set associative. That means, cache has got a number of sets, and each of them consists of n blocks. So now instead of talking about cache as a thing, containing a set of cache blocks. We say that it's a hierarchical hardware containing things that we call sets, and each set contains n blocks. That means every memory block can get mapped. Earlier in a direct map scheme, it was mapping a memory block to a cache block. I do not do that right now. I map a memory block to our cache set. By the formula, the number of the set is the index of the set is, the memory block number modulo number of sets in the cache, why do I do that? Because now given a memory block number, I know in which set of the cache, it should be in case it is present at all. And inside that set, I have, n different locations to search for. So instead of searching the full cache, I searched, only a corresponding set. So in that way, I have an in the middle approach.

The scheme is not fully associative, I do not have a complex search by hardware. I have to only search inside a set. At the same time, we are alleviating one of the issues with direct map cache. That means, in direct map, if there were multiple memory locations, which were getting mapped due to the modulo operation with the same cache block number, one getting loaded would need to replace the other. But now I can have due to this idea of sets and one set containing n blocks, I can have multiple memory locations mapping into the same set, staying together in the cache.

So this is the advantage we get by doing the trade off. And in that way, there is a related question that how do you decide the value of this n ? Now it's more of a design question. You have to choose a suitable associated value, based on what is your target design criteria.

Look at this important things like if you increase the associativity, it decreases miss rate up to a point. Why is that so? Because if you increase set associativity then coming back to the earlier point. I have the possibility of storing more number of memory blocks mapping to the same set

together. So, maybe it will reduce the miss rate up to a point. But what is the disadvantage? It increases the hit time. Why, because again I have to do some amount of search inside the set. So, if the set size goes on becoming big, the hit time increases.

(Refer Slide Time: 36:41)

The classic 5 stage RISC pipeline

Instruction Level Parallelism

Cache replacement policy

- ▶ In direct mapped cache, a new block can go exactly one location
- ▶ In fully associative cache, a new block can potentially replace any existing block - how to resolve ?
- ▶ In set associative cache, a new block can potentially replace any existing block inside a matching set - how to resolve ?
- ▶ Least Recently Used (LRU) policy - The block replaced is the one that has been unused for the longest time.

GPU Architectures and Programming

Soumyajit Dey, Asst. Prof., IIT Kharagpur

Now, another important thing is the replacement policy. So what is that? Now, as we have discussed that suppose in a direct map scheme, when new block comes and this block is going to get mapped to some cache block position where there is already some data. That means that data has to be replaced. So, in this case there is no problem, right, because every memory location, every memory block has a unique mapping with a block in the cache. So whenever it is being referenced but the corresponding position is filled up, you have to replace it with this thing. So there is no issue. But how about fully associative? Then a memory block can potentially replace any existing block . Because anything can go anywhere.

So then comes the question that how to resolve this. Suppose I have loaded a memory location. And then I have to load, another data. I have to find out whom to evict now. Because there is no rule, any data point, any block from the memory can be mapped to any block location in the cache, there is no rule. So how do I resolve whom to evict. The same question also comes in case of set associative cache, because now I suppose I am trying to load a specific memory block it can potentially replace any existing block inside a matching set.

Because the operation if you remember, was this. If you are given a memory block number you identify the corresponding set number. Now inside the set you have n possible blocks to replace. Which one to replace? The most standard policy that is used is Least Recently Used policy, which means the block replaces something which is being unused for the longest time. That means in every cache block, we have an idea that okay for how long it is lying there without being used. So there is a least recently used block. So you will replace that one.

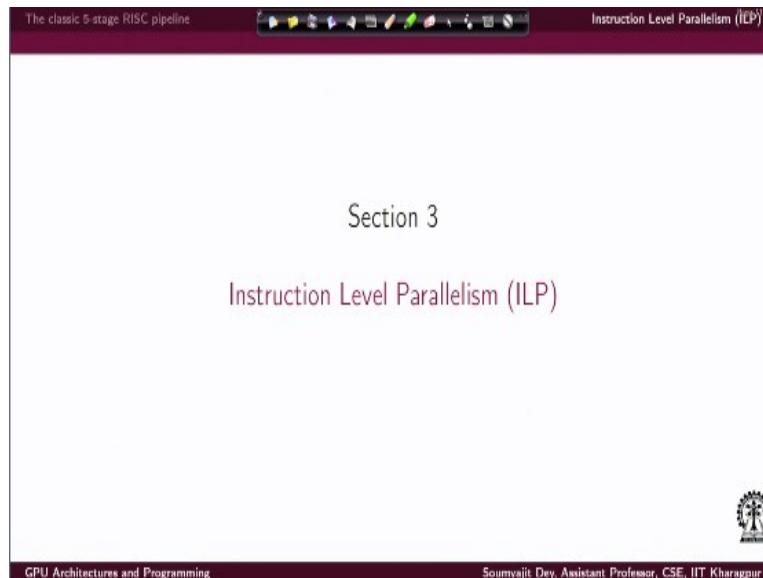
So this is kind of a summary of the different pieces of information. Which is really important with respect to memory system design its hierarchy, its access mechanisms. Its mapping schemes and replacement policy, which are pretty much used, and well known concepts in RISC processor. So with this will complete the present lecture. Thank you.

GPU Architectures and Programming
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 3
Review of basic COA w.r.t. Performance (contd.)

Hi, so let us get on with the section three of our first part of our course, where we are kind of reviewing the basic background on RISC processing of basic computer architecture.

(Refer Slide Time: 00:30)



Now, if you recall, we have discussed about the five stage RISC pipeline. Next we have discussed about the memory hierarchy that is prevalently, popularly used. Now, we have just covered the basics here.

And after that, we are trying to figure out what are the different techniques that are popularly employed inside microarchitecture in order to exploit the parallelism that is present while instruction processing. Because, although we started with the basic background that I have a basic five stage RISC pipeline in modern microprocessors the pipelines are far deeper. Moreover, pipelines are hardly parallelized.

That means there are multiple parallel functional units, you can just simply view it that in a modern microprocessor there are a collection of pipelines which are processing instructions in

parallel, although that arrangement is not that simple. So, this whole idea of figuring out how a microprocessor can function in parallel through pipelining and parallel processing kind of techniques. This comes into his purview of the title, that we have here for section three which is instruction level parallelism.

(Refer Slide Time: 02:05)

The classic 5 stage RISC pipeline

Instruction Level Parallelism (ILP)¹⁰

Actual Pipeline CPI

Pipeline Cycles per instruction (CPI) = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

- ▶ Handling hazards require both architectural and compiler techniques
- ▶ Data hazard types while executing instruction i followed by j in a pipeline
 - ▶ RAW — j tries to read a source before i writes it, so j incorrectly gets the old value
 - ▶ WAW — j tries to write an operand before it is written by i . Will not happen in simple RISC, but in pipelines that write in more than one basic stage or allow an instruction to proceed even when a previous instruction is stalled
 - ▶ WAR - j tries to write a destination before it is read by i , can happen in case instructions are reordered
 - ▶ RAR - not a hazard

instr i → instr j ⇒ IF ID ...

Soumyajit Dey Assistant Professor

Now, to discuss about how I can make what are what are the architectural and compiler techniques using which I can make microprocessor compute on instructions very fast. Let us go back to the basic formula that we had. Which was that the basic index of how how fast the pipeline is executing is CPI that is cycles per instruction. So I have a pipeline, through which instructions are flowing starting from the fetch states, up to its completion in terms of register write-backs and memory updates.

So, we are thinking that let us say a single instruction take some m number of cycles on the average so I would say that the pipeline's performance is m . ie. the CPI is m . It takes m cycles per instruction. Now, if I am trying to compute the CPI inside the pipeline execution. What is the actual CPI that depends on several factors, as we have discussed earlier, considering a multi stage pipeline where an instructions operation is broken down into several basic stages. And every basic stage is actually engaged in processing some instructions whenever it is executing without any kind of stall that gives me an ideal pipeline scenario. And inside an ideal pipeline, at

the end of the pipeline I can see one instruction getting completed by every clock cycle. So that gives me an ideal pipeline CPI for a single pipeline as 1.

That means one instruction gets completed in every clock cycle. However, it doesn't really happen because as we have seen, there are several kinds of hazards structural, data as well as control hazards. Which introduced what we call as pipeline stalls. So due to a structural hazard that can there can be a stall in the pipeline, that means some instruction inside the stage is not progressing in the next stage in the consecutive clock cycle.

So that induces a structural stall and similarly there can be data hazard induced stalls as well as control hazard induced stalls Now the question is, how are such hazards detected and handled? There are various techniques for that. Now as we can understand that as a collection of these all different phenomena, we have the actual pipelines CPI deviating from the ideal pipeline CPI. Now of course the calculation becomes more complex when I have more complex pipelines, which can execute multiple instant which can actually get multiple instructions issued in parallel, instead of just having one fetch unit issuing, I mean, executing and then starting to fetch one instruction in one clock cycle. So, starting from this perspective. Let us first discuss what are the classifications of hazards and what kind of techniques are practically used for getting around them by the architecture.

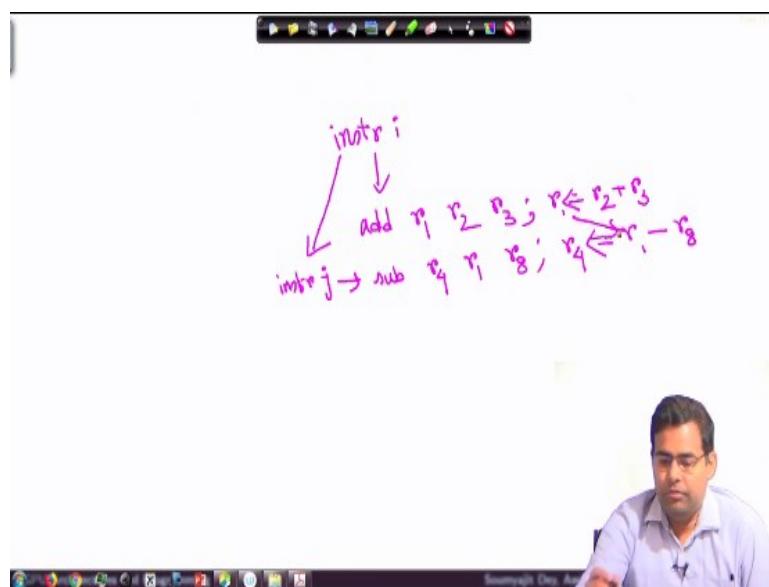
Of course for handling hazards, there are well known techniques based on compilers, which can also help us to alleviate certain types of hazards now this is also something we will look into. So, I mean the overall point here is that these hazards are kind of impeding the overall performance. Hence, there has to be certain techniques. Some of the things which can actually help you to approach the ideal CPI.

Some of these techniques can be implemented in hardware, so they are architectural techniques. While some of the techniques can be implemented by the compiler so that it can generate the assembly and corresponding machine instructions smartly enough so that the architecture can process them with in a more parallel fashion, providing us with higher throughput and automatically able to exploit higher level of parallelism.

So, starting with a classification of different kinds of hazards. So consider the different scenarios for which we have some data hazards. Now for that. Let us start with a scenario that I have an instruction i executing and is followed by the execution of an instruction j. So, let us consider instruction i is followed by an instruction j. Now, what are the different problems that can happen when this sequence is entering the instruction fetch unit, followed by instruction decode unit, and so on so forth. So, the first kind of hazard that may happen is, what is known as read after write (RAW) dependencies.

Or read after write class of hazards. So suppose instruction j is trying to read a source before instruction i writes it. So, let us try and understand how that can happen. So let me pick up a simple example here.

(Refer Slide Time: 07:35)



Suppose I have this instruction i. It is going to perform some operation. So let this instruction be an addition. And it is going to add values from registers r2 and r3, and write it to r1. So, overall, I have r1 is getting updated with the content of r2 and r3. Now what if I have an instruction j which is going to use the content of r1 for some purpose so let it be subtract instruction. It is going to subtract from the content of r1, some content of some register let say r8 and put it to r4. So, in terms of pseudo code if we write I am trying to update r4 with r1 minus r8. Now the thing

is, I want these instructions to be scheduled. Like, i is to be followed by j. Which means, when I am executing instruction j, r4 deserves an updated value of r1.

That means r1 must have been updated by the previous i-th instructions so here. I have this dependency. Now, coming back to our description of the hazard. So j is trying to read from some source. and that is some place where i is supposed to write. Now as we know in the pipeline is going to write, execute the write part in the write-back stage. That means, this destination register for i will be updated in the write-back stage.

Whereas the read has to wait until this update happens. Otherwise, what can happen is, j will incorrectly read the old value, this is something we have discussed earlier also. So, this is kind of a hazard, known as, read after write hazard. So unless I put in the stall in the execution of instruction j, it is going to read the older value of source, instead of reading the updated value, of the source register. So this is a read after write kind of dependency.

Now, there are several other kinds of dependencies that may happen. So read after write is the most simple one, and also the most frequent one of them. You have to understand that unless I am going to delay the execution of j it is bound to read from the registers, an old value. And it is not going to consider the updated value, that is supposed to be written by the preceding instruction that is instruction i.

So this is your example of read after write and to preserve the semantics of the instructions ie. in order to force that the read happens only after the update that is done by the write we have to insert the stall. So this is a data hazard of type read after write. Similarly, there can be write after write(WAW). So I am always considering that i execute followed by j execution. Earlier instance I had that j was going to read from a source. Before i write the source.

Now, let's consider this is execution, that is going to write something. And j also went to write something. Now, j tries to write an operand and before it is written by, i. Now why it can happen in a general simple pipeline of where register updates will only happen in the writeback stage.

This problem cannot happen right because instruction i is being followed by instruction j in the pipeline.

So only after i will update it's write. Only after that, j will be updating its write, it cannot happen in such a simpler pipeline but it may happen in a, in a more complex pipeline. Where you can have the facility to do write in multiple stages, rather than in one stage. So, in summary, this will not happen in simple RISC. But in pipelines that write in more than one basic stage or allow an instruction to proceed even when a previous instruction is stalled. These are the situations where the situation may arise.

Now, the other type of dependencies, write after read (WAR). Now what is really that? j is the succeeding instruction and it tries to write a destination, before it has been read by i. Now as we have discussed already that i is being followed by j. So how can really j write before i reading the instruction. So, before proceeding with this again, let me go back to WAW type and repeat because it is related to WAR. In WAW the problem can only happen when in the pipeline there are multiple possible states where write can actually be executed. And if the previous instruction is doing some write at the later stage of the pipeline, whereas the following instruction is trying to do the write at some earlier stage of the pipeline then this problem can come.

But what about WAR? Here we are talking about the write after read, where the write is an event of the succeeding instruction and read is the event of the preceding instruction. So read is by i. And j is the succeeding instruction trying to write. Now. In an inorder pipeline this cannot be a problem. So we have a new word here what is an inorder pipeline. Whatever we have discussed till now is extremely in order that means I am given an instruction sequence that is exactly the sequence in which I am feeding the instructions into the pipeline. But the point is, it is not always necessary.

There are situations we will explore them soon, where we will see that instructions may get reordered or instructions execute out of order. When that happens, So suppose there is a smart mechanism inside your micro architecture which is deciding that okay I will do a reordering of instructions. It has to do the reordering in such a way that this issue doesn't arise that if it is the

case that i is going to read some location and j is going to write some location and the original ordering is i followed by j. If i switch the order then, j, will be writing a destination and after that, i will be reading it, which is not the actual order as it is indeed there is kind of indicated by the original program. So that is again a violation so WAR is a violation.

So, we see that write after write and write after read are violations that may happen in certain scenarios. For write after read, instructions have to execute out of order. For write after write, the pipeline has to have the facilitated from multiple stages write are possible. So, apart from this, the read after write is a scenario, which can occur even in a basic pipeline. But that can be elevated with suitable stalls.

Now what about the other combination that can happen, combining reads and writes, so we have read after read that is never a hazard, because the reads, do not update state any register.

(Refer Slide Time: 16:12)

The slide is titled "Compiler Techniques for ILP". It features a purple header bar with the text "The classic 5-stage RISC pipeline" and "Instruction Level Parallelism (ILP)!!". Below the title, there is a text block: "To keep a pipeline full, a compiler can find sequences of unrelated instructions that can be overlapped". A C-like code snippet follows:

```
for (i=100; i>=0; i=i-1)
x[i] = x[i] + s;
```

 The text "Unoptimized MIPS" is present. Below it, a "Loop:" section contains assembly-like code:

```
L.D F0,0(R1) ;F0=array element
ADD.D F4,F0,F2 ;add scalar in F2
S.D F4,0(R1) ;store result
DADDUI R1,R1,#-8 ;decrement pointer //loop overhead
;8 bytes (per DW)
BNE R1,R2,Loop ;branch R1!=R2 //branch decision
```

 On the right side of the slide, there is a video frame showing a man with glasses and a blue shirt, identified as Soumyajit Dey. At the bottom, the text "GPU Architectures and Programming" and "Soumyajit Dey, Asst. Prof." are visible.

Now, as we have discussed earlier, that there are various possible techniques in which the parallelism of instructions can be exploited, some of the techniques are architectural and some of the techniques are also compiler based. So, let us kind of review what are the different optimizations that a compiler can do to extract instruction level parallelism. I mean, of course there are many possible optimizations. We will just talk about one or two of them for our basic motivation.

So consider at this point, the simple C code segment that we have for a loop. So we have a loop with the loop index down counting from hundred to zero. And all we are trying to do is we are doing a scalar addition we are adding the scalar value s to all elements of the vector x. Now, consider the corresponding MIPS assembly, which is unoptimized.

So, what do we really have in the MIPS assembly, you have this loop. (17:20) So, we are assuming that the address of this array, x, which is storing in the vector x, ie, the base addresses is located in r1 and r2 is the end address essentially x[100] at the start. That address with offset zero gets loaded to the register is F0. F2 is a register containing the scalar value, that is, s right. So, you execute ADD.D basically ADD for double type values.

And you store the content in register F4. So, F4 essentially has the content of x[100] + s. And then you store the result. Where do you store it? You are supposed to store it exactly in the same place from which you read it right? So, you store it by this instruction, exactly at that location. What do you do after that? You are supposed to browse back in the array. How do you do that. So, you decrement the pointer by 8 bytes, there is a width of the double type words. So you decrement the pointer. That is the content at this content of R1/8. And then you branch again into the loop. In case you still have data to be processed, that means the base address is inside R2, as long as you are not done. Essentially, this is going to contain that and as long as you are not done, you will be continuously going inside the loop and doing the addition. and in that way you do the scalar addition here.

Now, what is the issue here. Why do I say that it is unoptimized. It can be considered unoptimized, because the reason we can have an alternate version of the loop where we may have, we may end up with the less number of instructions to execute.

(Refer Slide Time: 19:32)

The classic 5-stage RISC pipeline

Instruction Level Parallelism (ILP)

Unrolling: eliminated three branches and decrements of R1 (Hen Pat et al. al.)

```

Loop: L.D F0,0(R1)
      ADD.D F4,F0,F2
      S.D F4,0(R1)
      L.D F6,-8(R1)
      ADD.D F8,F6,F2
      S.D F8,-8(R1) //Code size increase - more instr cache miss
      L.D F10,-16(R1) //more no. of live values - increased register pressure
      ADD.D F12,F10,F2
      S.D F12,-16(R1)
      L.D F14,-24(R1)
      ADD.D F16,F14,F2
      S.D F16,-24(R1)
      DADDUI R1,R1,#-32
      BNE R1,R2,Loop
  
```

GPU Architectures and Programming Soumyajit Dey, Asst. Prof.

So let us see what such a solution. So, suppose the compiler is provided with this kind of an unoptimized code in the initial part of the code generation process. And then the compiler does some architecture aware optimization, which is this kind of loop unrolling. So what it does is the compiler will instead of executing these many hundred iterations of the loop, it thinks that Okay, I will execute these many divided by four iterations of the loop. And inside each of the iterations, I will do for others additions. Let us see how the code looks like in that case. So now I have the loop, containing a sequence of 4 scalar additions of constitutive locations in the array. So these are load, followed by add, and then store. Again load of the previous byte, followed by add, and then stored. Again the load of the previous to previous byte. that is why of set is -16, followed by add followed by stored. So at this point I am expecting that from your basic UG background on computer architecture. You should familiarize yourself with MIPS assembly, so that this kind of a program is readable for you. Just a reminder here. So we do a store here again. And then again, again we do a load at the offset of the fourth position.

And then again we do the add and store it back right? So I am inside one iteration of the loop instead of doing one scalar addition, I am doing 4 scalar addition. And then, I am shifting back the pointer by eight times four ie. 32 bytes, and again executing the loop with the branches branch, if not equal instruction right? So what does this really mean I am doing the same functionality. Not a problem.

All I am doing is inside one iteration of the loop i am doing more number of additions, how does it tell. It tells because I am executing the branch if not equal instruction, less number of times, the previous number of times /4 that many number of times. So, in every iteration, I am eliminating three branches, and I am executing the 4th branch. Also I am saving on another instruction which is the pointer shifting ie. the decrementing of R1. So, I have eliminated so many decrements operations and so many branch operations. Why is that a good thing, because the pipeline that takes the equal amount of time for decrementing by eight or 32, but it just has to execute that many instructions list. It also has to execute the branch if not equal instruction that many times lists right? So the for every branch we know there can be associated stalls and all that. So all those problems get elevated.

In that way, I have lesser number of instructions executing. And also, if we look you look at the code here.I mean, with lesser number of instructions executing, I have an overall speed up for this scalar Addition that is going on. And we have more overlapped execution in the pipeline, because the instructions are unrelated. I do not have branches. So, I do not have sequence of branches for each ADD. So that makes this execute faster.

However, what is the problem with this. In general, there is no problem but of course, you can always argue that if we follow this process that again, I can unroll any loop up to any bound. Definitely that is not a good solution because programs are considered good for the loopy behavior right? That means there has to be a sweet spot.

That means what is a good unrolling factor. Now let us consider the side effects, or bad things due to unrolling. Your code size is increasing right? Earlier your assembly was this. now you have a bigger assembly to be executed, you have loopy behavior But inside each iteration of the loop you have more work to do. So your code size increases that will create more number of instruction cache miss.

Because you are now fetching more instructions from the cache. I mean since your code size has increased. So there will be more number of misses. And you are fetching more instructions of

basically which are the same instructions, but they are duplicated and stored in separate memory addresses. So there's definitely a problem.

And the other problem is when this code is executing, you have more number of live values. So more registers may get engaged due to renaming and other things which will cover letter, and that creates a problem called register pressure. So these issues are there, which will be clear, of course, as we progress through a bit more of content, but just to make a point that one should not think that okay unrolling is a smart thing to do but that means I should completely unroll any program and forget loops. There's also not a good thing, because of these problems.

Now, coming to some other techniques that also help with respect to exploiting the instruction level parallelism. Now, in unrolling the help came because we have less number of branch computations we also elevated some arithmetic instructions in terms of pointers increment and decrement. So with all this that helped in executing the pipeline faster with more overlapping structures and eliminating certain specific assembly instructions. This is more of a compiler optimization. Not something done by the architecture so it is not a hardware optimization. There are, as we have discussed earlier, that optimizations can be done both by the hardware, as well as the software. So what can we have possible software optimization.

Let us speak of that. Sorry, what can be a possible good hardware optimization, let us look into that.

(Refer Slide Time: 25:58)

The classic 5-stage RISC pipeline

Instruction Level Parallelism (ILP)¹¹

Branch Prediction assisted ILP

General single level predictor with 2-bit saturating counter

- ▶ conditional jump has to deviate twice from past before the prediction
- ▶ Consider a sequence of altering decisions in a loop and calculate performance improvement over 1-bit saturating counter !!!!

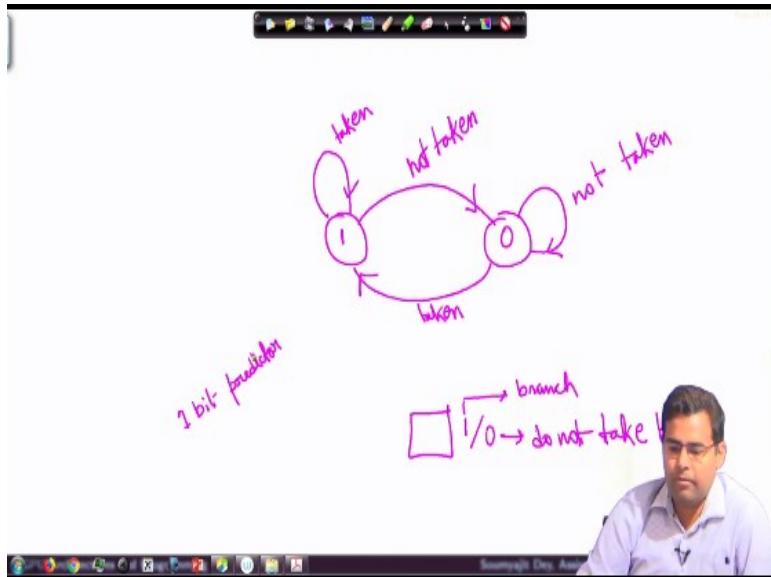
GPU Architectures and Programming

Soumyajit Dey, Assistant Professor

So, in modern microprocessors there is a specific hardware unit called a branch predictor. What it tries to do is to decide whether a branch should be taken or not taken, and we have discussed a bit about this earlier also, that I can, I can say that a branch will be always taken, or I can say that the branch will be never taken that can be a static kind of prediction. So, because in general for a uniform statistics, there is a 50-50 chance about that.

However, things can be done with a bit more statistical sense. Like, we can look at the previous history of branches and decide. Okay, whether to take the branch or not. Now, by looking at the previous history, I mean if I think that I will, I will implement predictor which is much more sophisticated. Let us think of the behavior of such a predictor in terms of a finite automata. So what can be a very simple automata.

(Refer Slide Time: 27:08)



So, consider a simple situation that I am going to decide whether to take a branch or not based on a one bit content flip flop containing either a 1 or a 0. If it is 1 that wouldn't mean are you want to take the branch. And if it is 0 you do not take the branch. Now, I want to implement finest admission logic to decide on this, and I have the previous history of branches, based on only 1 bit. So, suppose the value is 1, which is indicating that the previous branch was taken and right now, if I get a branch instruction, I will take the branch. And if this is a correct decision. Then again, the state gets upgraded with one only because the branch was really taken.

Now let those value being 0 denote that. Okay, although I thought that the branch was taken. But in reality, it was not taken. The I shift to another state. That means I update this one with a 0, meaning that this last branch I did not take.

So, next comes another branch instruction, I will also decide to not take it. And that will be kind of a self loop for the state, denoting that I remain in this not taking state that means I do not update the content zero in this flip flop. Now, following the similar logic and completing this automata. While my decision is that if any instruction comes I do not take the branch and now if finally, it is found that the branch has to be taken then I switch back to 1. So, this completes the design of a very simple one bit I mean a branch predictor with 1 bit history. It is just storing the information about the last branch. So to summarize, this is 1 bit predictor.

Things can be much more complex. As we can see, so this is nothing but just an extension of that same concept. And we are now thinking that okay I have predictor with 2bit and the formal term that you use is 2bit saturating counter. Saturating counter for obvious reason that okay if I take a decision and that decision is reinforced twice. I keep on taking the decision.

So that can be a nice summary. I took the decision that the branch is taken. And then I find that Okay, again it is taken. So now I switch to a state that I call as strongly taken. So, if I find that in a next sequence. There is a branch instruction and is not taken. I come to a state that is called a weakly taken that means Still, if an instruction comes, I will consider to take the branch, but then again I see that this also wrong. So then, again, the branch is not taken. Then I come to a state that is weakly not taken all that is happening is, if I go, if I think of the original construction that I just did. I was simply remembering the history of the previous branch and trying to do the same. Now, I am trying to do something based on the history of last two branches. So, if last two branches were taken I will definitely take the branch.

If the last word one branch was taken, but the last branch was not taken. Still, I will take the branch provided, I have switched to the weakly taken state from the strongly taken state, and will continue like that. So, you can just encode this behavior, as we can see in the form of a two bit saturating counter. And with this kind of a switching logic, which will actually tell you when or not to take a branch.

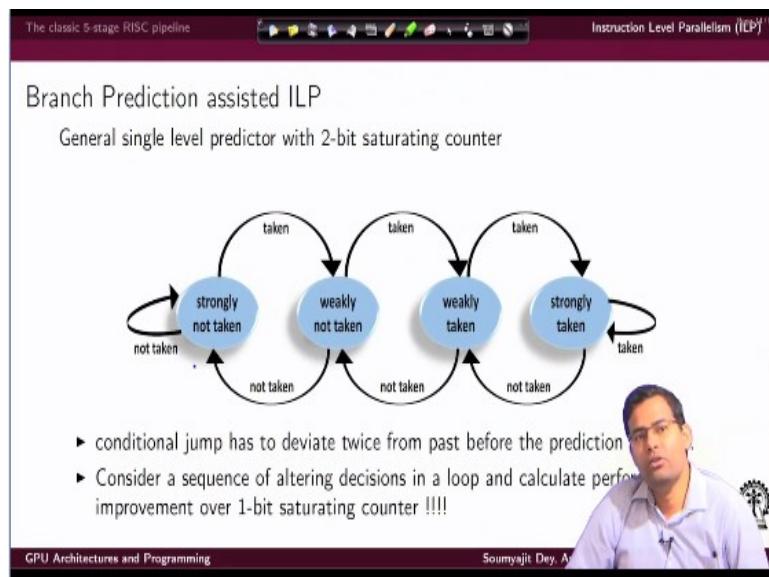
All again I will just repeat that the weird things have started different now is, in my simpler design, it was a one bit predictor. In these design, you said 2 bit predictor. I am trying to decide on branches, based on a history of size 2. Earlier, I was trying to decide on branches, based on a history of size 1. So maybe we will take this up again. Yeah. Thank you.

GPU Architectures and Programming
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 4
Review of Basic COA w.r.t. Performance (contd.)

Hi. So, we were discussing about different possibilities of branch prediction in order to increase the available parallelism in a system or reduce pipeline stalls. That can happen due to control hazards that is branch hazards. So we were discussing about this 1 bit branch predictors, and we have thought that okay there can be generalized right? I can also have a 2 bit branch predictor, essentially a branch predictor which will try to predict a branch decision based on more amount of branch history.

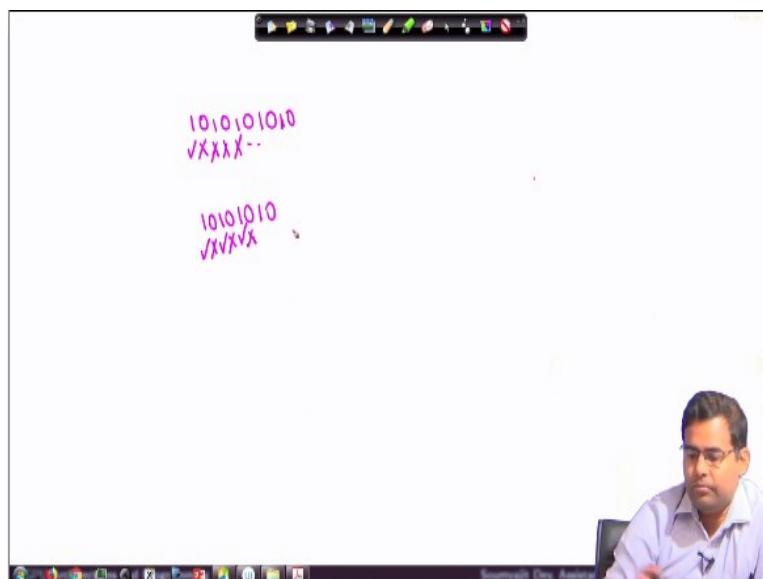
(Refer Slide Time: 01:01)



So all that 1 bit branch predictors does is, it just looks at the previous decision and repeats it. A 2 bit branch predictor looks at the last two decisions and tries to do something, like that. So this is an example of a branch predictor designed with a 2 bit saturating counter. So, I mean, just to motivate why is that a good thing. First of all we have already understood earlier that there are 4 states - strongly not taken, weakly not taken, weakly taken, and strongly taken state.

So strongly taken means, I am always deciding to take the branch and this we have already discussed. If once I go wrong in my decision I come to weakly taken state, here also I want to again take the branch. Again, if I go wrong in my decision. I come to weakly not taken state. Here I am always predicting that I will not take a branch. If I, if I am correct, that means a branch is really not to be taken then I go to the strongly not taken state where also my decision always is to not take the branch. And I shift from strongly not taken to weakly not taking if I go wrong once ie. a branch is taken. I shift from weakly not taking two weakly taken, if I go wrong again. That means I predict that the branch is not taken but it is again taken, and so on so forth. So, just to observe the performance of the system. Let us go back and think of the performance of a one bit branch predictor in worst case scenario.

(Refer Slide Time: 02:27)



So, worst case scenario would be something like, suppose I have a problem where the actual branching decisions is executing with sequence of 1s and 0s, like this. They are going to be alterations. So let us say I am executing a loop inside which there are two branches. It is always the case that the first branch is taken and the second branch is not taken. So that gives me this pattern. How do you think then the one bit branch predictor will work.

So let us say, it is initially in the second state, so it will predict this correctly. And then it will think that in the next branch also I should take the branch. So this will go wrong. Now it seems that it has gone wrong. So in the next state it again wants to not take the branch, whereas the real

thing is that the branch has to be taken, so it will go wrong, and the behavior will continue. So, you are always predicting wrong.

How about using this to 2 bit branch predictor here. Of course, the behavior will depend on my initial state. let us assume that I am in the strongly taken state. Okay. So, assuming that. Let me again write the sequence here. So I am in the strongly taken state, that means, this one I predict that I will take and I am correct. Then here again, I am in the strongly taken state. But the real branch is not supposed to be taken. So I go wrong, because I am always thinking I will take it. So then I will also have a state change right so assuming that in the automata You'll see that I am now going to switch from the strongly taken state to the weakly taken state, because, although I thought that the branch is taken, it is really not taken, but still my decision is, since is a weakly taken state for the next branch I am again going to take the branch. So, I will go correct here. So then, that also means that I switch, since the branches really taken a switch to the strongly taken state right? So, again, for the next prediction, it will go wrong. And then again I go correct, and they go wrong like that.

So, I have an improvement in performance here over a one bit saturating counter. So in that way I can see that with addition of some history of information. I get an improvement in performance. Now, of course, I can keep on making a more complex branch predictor by looking into more amount of history and taking suitable logical decisions.

(Refer Slide Time: 05:39)

The classic 5-stage RISC pipeline

Instruction Level Parallelism (ILP)^(1/2)

Hierarchical Prediction

How about generalizing the idea of prediction with larger branch histories.

- ▶ store m length history of a branch - 2^m possibilities
- ▶ for each possibility use an n -bit predictor : (m, n) prediction scheme
- ▶ a two-level predictor with m -bit history can predict any repetitive sequence with any period if all m -bit sub-sequences are different.



GPU Architectures and Programming

Soumyajit Dey, A.I.T.

Or, alternatively, I can make the branch predictor even smarter and go for what we know as a hierarchical predictor. So if we generalize the idea, it would be something like this, that we look into branch histories and take decisions. Now, suppose I am always looking at an m length history. So there are 2 to the power m possible branch histories. so I store these different history in a location, maybe a shift register. So, I have 2 to the power m possible values of this history. And for each of these situations that means for each of 2 to the power m possibilite, I have one n -bit predictor. So just to understand back.

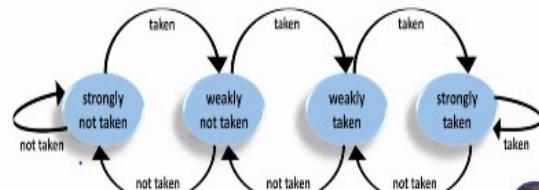
(Refer Slide Time: 06:26)

The classic 5-stage RISC pipeline

Instruction Level Parallelism (ILP)^(2/2)

Branch Prediction assisted ILP

General single level predictor with 2-bit saturating counter



- ▶ conditional jump has to deviate twice from past before the prediction
- ▶ Consider a sequence of altering decisions in a loop and calculate performance improvement over 1-bit saturating counter !!!!



GPU Architectures and Programming

Soumyajit Dey, A.I.T.

Here what we are doing. We are looking at the last decision and doing a prediction. So the prediction decision is based on 2 bits right? It is a 2 bit saturating counter decision and but I am looking at the last decision. So, I can also do is, I can look at the history of m length, and for each possible such sequence. I'll take the decision based on another n-bit predictor. So this is a combination scheme.

I have a hierarchy, first I see what is the branch history, and then I look at for this history for this specific history. What is the state of my n-bit predictor among those weakly taken strongly taken less weakly taken more strongly taken. There will be more steps because if n is greater than 2 and all that. So, I mean, in that way I can take a more informed decision. It is known that if I have a two level predictor with an m-bit history, it can predict, any repetitive sequence with any period if all the m-bit sub sequences are different.

Let us understand what it means, suppose I am doing a two level predictor. And my choice of m means I am always deciding based on an m-bit history. It can predict any sequence which will repeat with some period. If all the corresponding sub sequences, these m-bit sub sequences are different. that means, for the m-bit subsequence the decision is always unique.

(Refer Slide Time: 08:15)

Dynamic Scheduling for ILP

- ▶ Simple pipelines execute instructions in-order


```
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F12,F8,F14
```
- ▶ SUB.D suffers as ADD.D stalls due to dependence
- ▶ different ordering will avoid stall in this case
- ▶ Out of order execution brings in the possibility of WAR and WAW hazards

Robert Tomasulo: developed algorithm to minimize WAW and WAR hazards while allowing out of order execution (tracks when operands for instructions are available to minimize RAW hazards and uses *register renaming* to minimize WAW and WAR).



So that was a bit of diverging into the idea of branch predictors. And looking into some amount of the intricacies, if not all, and we are also not getting into the details because, in general,

hierarchical prediction is a very complex phenomenon. There are a lot of other predictors that tournament predictors you can just go into those details. If you are interested, from an advanced architecture point of view.

Now, the next important topic here for us is the issue of dynamic scheduling for instruction level parallelism. So what is that. Now, we know that simple pipelines execute instructions in-order. Like, I have the sequence of instructions. Divide for double type, ADD SUB and all that. Now look at the operands here, from which it is very easy to decipher that the subtract instruction will suffer unnecessarily as the ADD instruction gets stall to dependence. So what is that that the DIV instruction is going to write to F0. Now this is again the read operand for ADD. So, this is something we have already discussed earlier, it is a read after write dependency, due to which ADD will stall.

Now SUB has no dependency on ADD. I mean, it is basically reading on the only common thing is, F8. But it is a read after read so there is not not not any kind of hazard or stall that is going to happen. So there is practically no reason why sub should suffer due to ADD getting stalled with the preceding instruction DIV. However observed that if I do an alteration here in the ordering. I can avoid the stall, because I can just execute because ADD will get stalled here I can just execute so me before at the end there is no stall.

Now, this is a good thing for this specific case because SUB has no dependency and observe that I can have a recognizer system in the hardware, which can actually identify that yes there is no dependency and this can go before this. So, I can have this kind of reordering of instructions that is executing instructions, out of order to get more instruction level parallelism, and increase the effective CPI.

However, there are issues, I cannot do it in an ad hoc way, in this case it is good. But if we just think of the earlier issues, like, as we discussed the hazards of type WAR or WAW, those can come in if I am doing out of order execution. So, I have to be very careful here, that when I am executing out of order, I should choose the order in such a way that these possibilities of WAW and WAR type hazards do not creep in.

In this way, there is a very famous algorithm by Robert Tomasulo well known as Tomasulo's algorithm, which is basically a way of choosing instructions that would execute out of order by minimizing these WAW and WAR type hazards and resulting stalls. So essentially the algorithm will track when operands for instructions are available to minimize this hazard. And they actually employ some optimization techniques well known as the register renaming to minimize this kind of hazards.

(Refer Slide Time: 11:58)

Register Renaming

```
DIV F0,F2,F4
ADD F6,F0,F8 // (RAW for DIV : F0)
S F6,0(R1) // (RAW for ADD : F6)
SUB F8,F10,F14 // (WAR for ADD : F8)
MUL F6,F10,F8 // (WAR for S, WAW for ADD)
// (RAW for SUB : F8)
```

- ▶ RAW is due to data dependency, stalls in-order pipeline
 - ▶ WAR/WAW constrains out-of-order execution
- ⇒

```
DIV F0,F2,F4
ADD S,F0,F8
S S,0(R1)
SUB T,F10,F14
MUL F6,F10,T
```

- ▶ S removes WAR of MUL, RAW of ADD
- ▶ S removes WAW of ADD, T removes WAR of SUB

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor



So let us look at an example of this register renaming. Just to be clear, we are not getting into tomasulo algorithm. There's quite a lot of stuff to be covered right now for our purposes and also not necessary.

Discussing about the register renaming option. So if you look to the code in the right hand side. As you can see, we have introduced two new registered options S and T. And there is some significant modification that has been done, into the original code of the left hand side, the modification is as follows. The add instruction which is the second instruction. Earlier it was writing to the register F6 now it is writing to another register S. And similarly, F6 has also been replaced by S in the next instruction, there is a stall instruction. And in the SUB instruction, which was writing in the register, F8. This is being removed, and there is a new register T, in

which is writing. And similarly, in the subsequent instruction MUL also, F8 has been replaced by T. So, this helps in removing several of the possible hazards that can come in. So let us look into the issues that get solved by doing this register naming operation. So as you can see this, including inclusion of this register S removes the read after write dependency of the multiply instruction.

Now, where is that read after write dependency. So, this multiply instruction was reading from F8 and F10, and it was writing to F6. Now, what is happening is this read after write was dependent essentially on this F6 in the store instruction. Now, let us explain that once again. So as we can see, this multiply instruction is writing to F6, whereas the store instruction is reading from F6. Now, this is a read after right, which would mean the store instruction, cannot be executing after the multiply. There cannot be any ordering and reordering possible between multiply and store. But now, once I make them, independent, that is, this S, is being used here instead of F6 for the store instruction. That means there is no write after read dependency between the multiply and the store instructions.

So now the store instruction is reading from the register S and writing to the location in R1, whereas the multiply instruction is writing to F6. So now, there can be an instruction reordering between this store and this multiply. So in that way this new register S removes this write after read dependency of multiply on the store instruction.

So essentially this issue gets resolved, as we as I am pointing it out here this issue of write after read is getting resolved.

Now there are some more similar issues that are getting resolved. For example, there was a write after write issue also, that is, as we can see that F6 is being written by MUL. And F6 is also being written by ADD. So, earlier there was a strict ordering that ADD should happen before multiply, but since now that is also being changing because ADD is writing to S and multiply writing to F6. So, they are different registers so in that way, this register renaming of S removes the write after write dependency of multiply also.

Now what else is getting resolved? Let us see, because we have also included some other register which is the T register. So, earlier there was this write after read dependency of the SUB instruction. So it was writing to F8, and F8 was read from by the ADD instruction. So again, there was no reordering possible between ADD and SUB. But now, since SUB starts writing to T. There is a possibility of reordering between ADD and SUB instruction. So in that way this register renaming of T removes this write after read dependency of SUB. Because it was writing here to F8 and Addition was happening from F8 also. But now, Addition is have a reading from F8, but SUB is writing to T. So, they can be ordered differently. So in that way. Several of these dependencies gets removed with with this register renaming operations. And that creates a case for suitable reordering of instructions which can help in increasing the effective parallelism if there is a possibility of executing multiple instructions in multiple pipelines in parallel.

So we are just trying to show that these are the possibilities here. This example has been also taken from the classic book by Hennessy Patterson similar to most of the other examples we discussed in this introductory slide. And we are just trying to point out that with this kind of optimizations they can be easily being carried out by compilers. And once they're done, they really help in exploiting more amount of parallelism.

(Refer Slide Time: 18:03)

The classic 5-stage RISC pipeline Instruction Level Parallelism (ILP)

ILP Using Multiple Issue and Static Scheduling

- Multiple-issue processors - allow multiple instructions to be issued in a clock cycle
 - ▶ VLIW (very long instruction word) - Parallel instructions statically scheduled by compiler; issue a fixed number of instructions formatted as one large instruction
 - ▶ Statically scheduled superscalar - issue a varying rather than a fixed number of instructions (compiler decided) per clock, in-order execution
 - ▶ Dynamically scheduled superscalar - issue a varying rather than a fixed number of instructions (hardware decided) per clock, out-of-order execution

For large issue width VLIW (with multiple independent FUs) is preferred w.r.t. statically scheduled superscalar

CPU Architectures and Programming Soumyajit Dey, Assistant Professor

So, as we have discussed earlier that there can be several possible ways to do these kinds of optimizations. They can be done with hardware support as well as software support. By software

support we mean smart compiler that can do these kinds of operations. By operations I mean deciding, which instructions are going to be issued in parallel, or not. Also, a smart hardware can do the same.

Now, it depends on you. We like to tackle this problem of deciding independence of instructions at what level? At the compiler level or the personal level. Now of course, whatever can be done at each level is also different. Just to take a case like when I am trying to do some optimization by the compiler, the compiler is not aware of the arguments that are provided to the program.

So all that we could do is some static optimization by looking at the problem. More or less, whatever we have discussed is all static optimization. We are not using any concept of dynamic optimization, which is based on what is the value in the register. We are just looking at the dependencies and the hazard. So these are all static optimizations which can be done by a compiler, or can be done by a smart hardware.

So, depending on where you want to do this. There can be several different possible processor design styles. So, When we are looking into the issue of how I can resolve dependencies among instructions and issue multiple instructions to be executed in parallel. So, based on this classification, we have the following three possibilities here. The first one is the VLIW style of execution. VLIW stands for very large instruction word.

So this is a style of micro architecture design, where the effective parallelism is exploited by the compiler. So, the compiler decides which are the instructions that should execute in parallel. And so essentially the compiler schedules these instructions and they're parallel execution .Issues a fixed number of instructions formatted as one large instructions. So essentially the compiler will look at the actual instruction stream and come up with packets of large instructions and each packet kind of sticks together a set of independent instructions which can be executed in parallel. And that is fed to multiple independent functional units so that they can execute in parallel. So here. This is more of a compiler intensive approach, the parallelism is decided by the compiler, the hardware is blindly following it and executing the instructions in parallel using multiple functional units which are already present there in the VLIW style of architecture.

Now, as example there are well known processors, which do digital signal processing, which are designed based on this philosophy. So, apart from this VLIW style of processors, we also have the other two options which is statically scheduled superscalar and dynamically scheduled superscalar.

So by statically scheduled we mean that the compiler decides which are the instructions that are to be issued in parallel. However, the execution of these instructions are in order. So, we have pipelines executing the instructions in parallel. But the compiler still decides which are the instructions that will be issued. The instruction issue width may vary.

Coming to the last option which is dynamically scheduled superscalar here. The hardware decides instead of the compiler that which are the instructions that are going to be issued in parallel. Also the other important thing is the hardware executes these instructions, out of order. Now, if the design philosophy is that we should have a very large issue width, then VLIW is preferred with respect to statically scheduled superscalar. Simply for the reason that both approaches require support from the compiler. And if we are really deciding that will the compiler should be packetizing or doing the issuing of parallel instructions. The VLIW makes sense because of the large issue width, because if we are keeping large issue width but still doing statically scheduled superscalar, then the problem is the number of instructions that will be issued may vary. When this varies and it is not always filling up this large issue width, then it doesn't make much sense to have a large issue width.

So, typically this idea of statically scheduled superscalar is used for narrow issue width which may be two. But for large issue width, VLIW is the preferred way to design, But most popularly used for most of the microarchitectures the design that is followed is dynamically scheduled superscalar.

So, in short summary, these are the different architectural styles that are practically employed for deciding how to exploit the parallelism resident inside instructions and execute more number of instructions per clock cycle. And over this lecture, we have tried to summarize the topics like how a basic RISC pipeline operates, how the memory is organized along with this specific RISC

pipeline, what are the limiting factors, with respect to instruction execution in the pipeline. And to alleviate those limiting factors, what are the optimizations that can be done. So that would be our overall review of existing computer architecture notions. That we will be using for building upon them for the GPU architectures, and programming part.

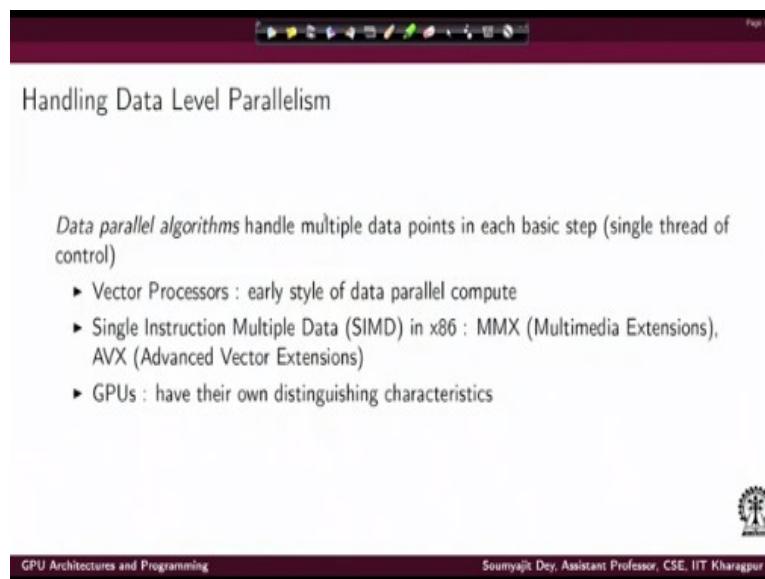
So with this, we will conclude this topic, which is overall review of basic computer architecture, and we will continue with the part on more fundamentals of GPU architectures. How GPU architectures and their execution differ from traditional computer architectures. Now, one thing that is to be very important in this case is this idea of instruction level parallelism. Because we will see that GPUs essentially are designed with the basic philosophy that large number of operations have to be carried on by the processor. Now, even at this level, there are different philosophies of computing, Like how really large number of operations can be carried out. Will it be carried out by executing different instructions in parallel, or will it be carried out by executing an instruction over multiple data points in parallel. And among those philosophies, which is the one that is used for GPU. These are the topics from which we will get on with the next part of the lecture where we deal more deeper into GPU architectures. Thank you.

GPU Architectures and Programming
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 5
Review of Basic COA w.r.t. Performance (contd.)

Hi, so after our basic introduction where we have done a basic review of general computer architecture trends, more specifically, some background on RISC architectures, with that background we are slowly going to delve deeper into how GPU operate, in terms of the architectural details.

(Refer Slide Time: 00:47)



Handling Data Level Parallelism

Data parallel algorithms handle multiple data points in each basic step (single thread of control)

- ▶ Vector Processors : early style of data parallel compute
- ▶ Single Instruction Multiple Data (SIMD) in x86 : MMX (Multimedia Extensions), AVX (Advanced Vector Extensions)
- ▶ GPUs : have their own distinguishing characteristics

GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, we have always fixed on the fact that while we discuss about the RISC pipeline. We have always say that how our RISC processor will execute a single instruction, using the standard phase decode, execute cycle. And at the end of that we also discussed how parallelism is handled in general purpose processors. And that also leads to some specific kind of processing that we know as vector processors.

And as we know that GPUs represent as more specialized kind of vector processor. Which have been found to be very useful for general purpose as well as graphics compute loads. So, the question is how is this data level parallelism handled? So, when we talk about data parallel

algorithms. Essentially, these are algorithms that handle multiple data points in each basic step. So there's a new term we have been talking about this data parallel algorithms.

When we say there is parallelism in execution of a program. As we know that there can be parallelism resident at different levels. There can be instructional level parallelism, there can be thread level parallelism. But when we talk about data parallel algorithms, the specific thing that we mean is the algorithm has a sequential thread of processing, but in each step of the processing the work will be done, on multiple data points simultaneously.

So it is like doing an add of multiple data points followed by doing a multiply of multiple data points followed by doing some other operation on again multiple data points like that. So, if this kind of instances of computation is found very frequently in some algorithm we like to call it a data parallel intensive or in general data parallel algorithm.

So, typically, as we can see that there can be several examples of graphics workloads. And several other kinds of processing, multimedia computation, or there are several such example workloads, which fit nicely into this notion of the data parallel algorithms and coming to the way they can be handled in hardware, So of course to handle such algorithms we require hardware processors that have the capability to exploit this parallelism.

Now, as we understand that parallelism in at the instruction level can be extracted in two ways like we discussed earlier, it can be done at the compiler level, it can also be done at the hardware level. So, in a good case, it should be a combination of two that the compiler is able to extract some parallelism from a piece of code that has been recreated sequentially, and also the hardware while executing instructions is able to mark out which of instructions that can execute in parallel. These are the two fundamental types we have already explored.

Coming to handling of data, parallel algorithms. There are three popular techniques. I mean, the third one of course is the one that is graphic processing units. The first one was vector processors. So, This was the earliest style of data parallel compute, like if you are trying to do operations on a parallel set of data points the same operation. Then, the solution that was

proposed was vector processors will learn a bit about them. Then came the notion of specific instructions, which will be part of normal CISC processors like x86. For example, some MMX instructions, There are some multimedia extension or AVX instructions which came later on the advanced vector extensions. So these were kind of SIMD instructions or single instruction multiple data type instructions that became a part of Intel x86 architecture. They are essentially instructions which perform the same operation on a vector data type, essentially, a set of data type, which are homogeneous. So, there's something that came next. The earliest style of data parallel compute was vector processors. And as we know in recent times, that the idea of graphics processing units, has caught everybody's imagination. And they represent the most modern technique of doing a lot of parallel computations in hardware together.

(Refer Slide Time: 05:24)



Vector Processors

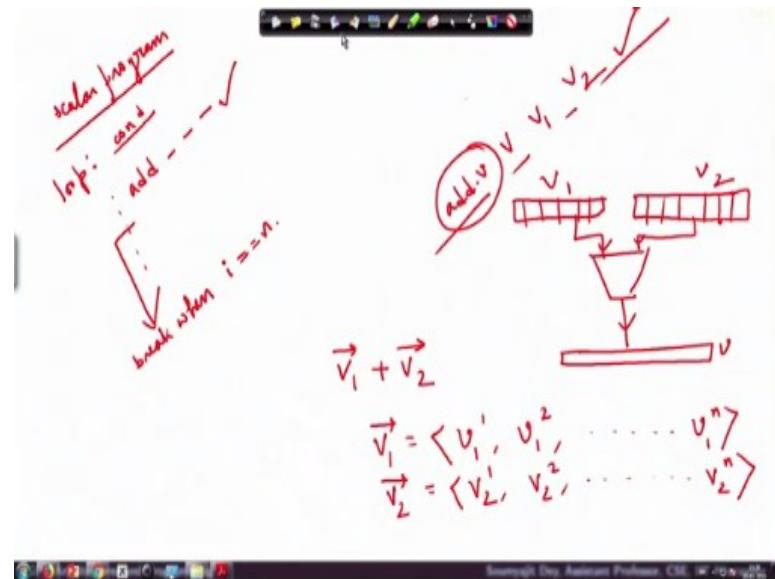
- ▶ Vector registers : Each vector register is a fixed-length bank holding a single vector,
- ▶ Functional units are also vectorized,
- ▶ Original Scalar registers are also present.
- ▶ VMIPS has eight vector registers, and each vector register holds 64 elements, each 64 bits wide.



So coming to vector processor so what essentially is the design philosophy. Behind a vector processor. It is fundamentally like this that you have a register file processor. Apart from having normal (by normal I mean scalar registers) you also start defining what we call a vector register. So what is the vector register? So each vector register is a fixed-length bank holding a single vector. So say you have length vector which is holding the different length values for the different dimensions of a house. So that can be stored as a single type - a vector of type length. It will now start saying that is not located in different registers, ie. the different components of this type are not located in different registers. But they are all stored in a single register which is having a much bigger size, but it also has these compartmentalized parts. Each of which is kind

of holding out some specific component of the vector. So each vector register is a fixed-length bank. It holds the single vector . So when I say that there is an operation I am doing. An operation like vector addition. So there are two vectors v1 and v2. I am having the content of vector 1 in one register and the having the content of vector 2 in the second register and they would get added.

(Refer Slide Time: 06:58)



Just as an example. So if I just say that I have two data points v1 plus v2. So V1 is a vector. That means, it contains components. Both V1 and V2 are n dimensional vectors and I am doing their addition. So, in our normal program, or let us call it a scalar program. So what I would ideally do is I will have a loop structure. And inside this loop structure. I mean, there will be a condition here. And inside this loop structure I will have some added instruction, which is doing addition of each of the components right? And I break away from this loop structure when the iterated variable of the loop is n. Now when I say, I have this kind of Vector instructions, I'd like to think that instead of this, I have some code, which is like this. So I have a specific instruction let sat add.v, which represents its adding content of registers, v1, v2 and storing it at some v.

So, in a way, what's happening is, there is some functional unit. And it is getting data from two vector registers, v1 and v2, and output get stored in another vector register v. Now the question is- this function unit also needs to be a vector. That means it should also be able to work on this

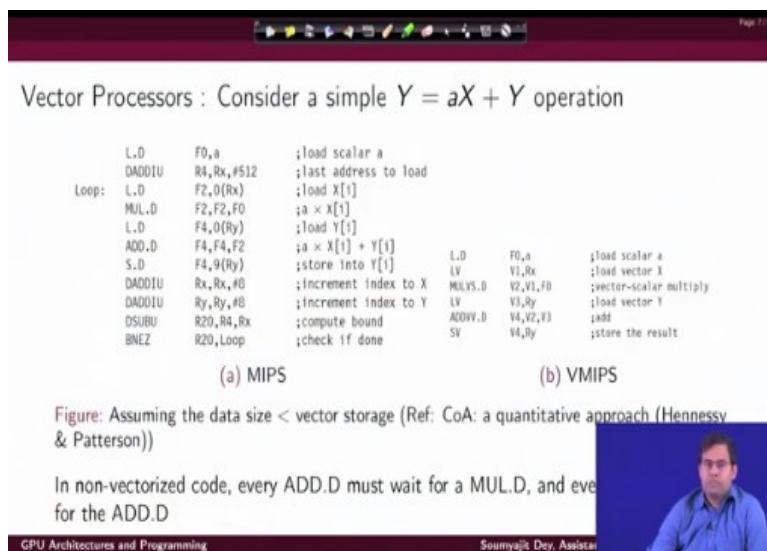
input data types in parallel. Or maybe I may have lesser number of units which work multiple times fast enough to operate them.

We will come to this pipelining idea, but the important thing is instead of having a single scalar instruction, I have a vector instruction. So basically, a hardware instruction which when executed, it operates on vector registers, and the vector registers have got fixed dimensions, holding each of the components of the vector. And in that way the same operation gets done. Maybe at a higher speed up. Why? We will come to that.

So, the vector processors will have a normal register, as well as this kind of a vector register. And functional units may also be vectorized essentially just like I explained earlier that I can have an array of ALUs, which can do addition, subtraction, or other arithmetic operations in parallel. Of course, now original scalar registers are also present.

Now, what was the first proposition in this space. Well, one of the most popular initial academic propositions was the idea of vectorized MIPS, or VMIPS. So the VMIPS comprise the normal register file, along with 8 vector registers, each vector register holds 64 elements. Each of these elements were again 64 bits wide. That means. This VMIPS would have been able to operate on vectors of dimension 64. And in each of the dimensions, the number of bits that were there for holding some value in that dimension will be 64.

(Refer Slide Time: 11:50)



The slide title is "Vector Processors : Consider a simple $Y = aX + Y$ operation". It contains two columns of assembly code:

(a) MIPS

```

L.D      F0,a      ;load scalar a
DADDIU  R4,Rx,#512 ;last address to load
Loop:   L.D      F2,0(Rx)    ;load X[1]
        MUL.D   F2,F2,F0  ;a × X[1]
        L.D      F4,0(Ry)    ;load Y[1]
        ADD.D   F4,F4,F2  ;a × X[1] + Y[1]
        S.D      F4,9(Ry)    ;store into Y[1]
        DADDIU  Rx,Rx,#0  ;increment index to X
        DADDIU  Ry,Ry,#0  ;increment index to Y
        DSUBU   R20,R4,Rx  ;compute bound
        BNEZ   R20,Loop    ;check if done

```

(b) VMIPS

```

L.D      F0,a      ;load scalar a
LV      V1,Rx    ;load vector X
MULVS.D V2,V1,F0  ;vector-scalar multiply
LV      V3,Ry    ;load vector Y
ACVV.D V4,V2,V3  ;add
SV      V4,Ry    ;store the result

```

Figure: Assuming the data size < vector storage (Ref: CoA: a quantitative approach (Hennessy & Patterson))

In non-vectorized code, every ADD.D must wait for a MUL.D, and even for the ADD.D

GPU Architectures and Programming Soumyajit Dey, Assistant

So let us just consider a small example of Vector SMP program. On the left hand side, we have normal MIPS code. On the right hand side, we have a vectorized MIPS code example. This is a standard example which you can easily find in the book on computer organization architecture quantitative approach by Hennessy and Patterson. For the time being we assume that the vector sizes are less than the vector storage.

That means we are working on data, where the data type is vector. And that means they are like array, but the dimension on which I am working on those arrays are smaller than then the vector storage possible in the architecture. That means, let us say, the vector registers have got 64 bit, 64 width that means 64 dimensions are allowed in and each dimension has got some number of fixed number of bits to store the value. So essentially, I am talking about working on arrays where the dimension of the array is limited to 64.

So here we are trying to do a simple scalar operation. So you want to multiply a vector capital X with scalar, a, and then do an addition, with another vector Y right? So, with respect to that. If you look at the MIPS code, the non vectorize scalar MIPS code. There are a few important things to notice here.

So in the non vectorized code, you see there is this addition operation, which is essentially implementing this Addition of $a.X[i]$, along with $Y[i]$. So, this addition operation is going to take one of the operands from the register F4 and store back the result to F4. So, it has got an input dependency, with the multiply operation. Why? because the other operand of this addition operation is in F2. And, F2 is going to be updated by multiply operation, previous to doing the addition. So, this multiply operation essentially does this multiplication of a and i th component of X has stored back to F2. And in addition operation, this has to be added with the content of F4 which is Y right? Now as we know that this is going to be done in a pipeline fashion. So for the multiply operation to n it has to write-back the value F2 to the register file.

And that would happen in the last cycle of its execution. And until unless that happens, add cannot really do the operation right? So, ADD.D needs to wait for MUL.D so there will be a pipeline stall, as we have seen earlier. And also, if you look at the instruction is S.D. So .D of

course means in MIPS mnemonics it means I mean we are working on double data types, just to remember. And so this again has to wait for the execution of ADD to complete. Now why is that, in S.D the store instruction is going to store the content of register F4 to some memory location. But this content would get updated by ADD.D previous to that. Then again for ADD.D to finish the write-back to F4 has to happen. So, S.D has to wait up to the point when the write-back to F4 is completed for the ADD.D the operation.

So, as we can see in the normal scalar implementation. I need to have the loop with the loop iterate variable i. I mean, in each iteration, I am doing, practically $a.X[i] + Y[i]$. And then I am incrementing i. In each iteration, I have the add which needs to wait for the multiply, and also the store instruction which need to wait for add. Now this wait has to be done in each iteration of the loop, there's the whole point here.

If you look into the vectorized code. As we can see, this vector ADD instruction has to wait for the vector multiply instruction to complete. But wait for how many time? Once. Again the vector store instruction needs to wait before the vector add instruction gets completed. Wait for how many times? Once. Because there is no loop here, assuming of course the dimension of the data size is less than the vector storage.

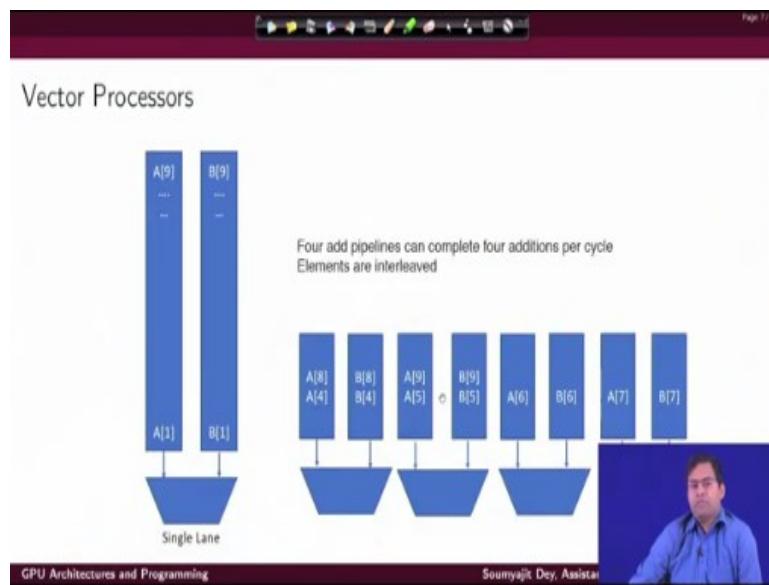
But then again there is a loop here, even if this assumption was not going to work. That means the data size was not less than vector storage. Still, then I would have a loop with much less number of iterations, the number of iterations would have decreased by a factor of vector dimension, so you can understand how many cycles of waiting you're going to save if you have the vectorized code, and the vectorized hardware for executing the code.

So, overall, the summary would be this the vector process, they will be executing scalar as well as vector instructions and vector instructions pass a lot of parallel work to the hardware. Of course, there has to be that much parallel hardware based compute capability, along with register banks which are vectorized. Now, what about the functional units? The hardware must have parallel functional units.

But it needn't be like this that okay my vector registers are of size of 64. So I need to have 64 ALU, it need not be that that the functional units is can be either fully parallel or a combination of parallel and pipeline units. It depends on how the hardware is implemented. For example, if the clock rate of vector processor is halved, doubling the number of lanes will retain the same potential performance. These are standard optimization between pipelining and parallel processing.

We will have an example for this. Now, the question is, if I have support in hardware for such vector instructions, assuming such support, is there some overhead for the compiler? Yes there is. Because then your compiler has to identify which are the segments of code that can be vectorized where to emit a vectorize multiplication instruction and where it cannot emit vectorized instruction. So this loop the vectorization and handling of dependencies is something that has to be done by compilers working towards vector processors.

(Refer Slide Time: 19:02)



Coming to vectorized hardware. So, how can you really implement vector processors? Of course, as we understand that the good thing is I have less number of instructions to execute. Now the instructions may take less amount of time if I have a lot of parallel hardware. It may take, not so much little amount of time, if I cannot really execute all the vectorize instructions I mean all the, all operations in parallel.

There has to be some thread off. But this idea always holds that since I have much lesser number of instructions to execute, the number of waits in the pipeline would be less. So coming back to how Vector instructions can ideally be executed.

So let us consider a trade off scenario here. That suppose I am just trying to show that this a single lane of execution here. So I am assuming that I have this ALU nicely set up with its input connected to two vector registers, each of their dimension is 10. And there is a single piece of ALU here. So I have been provided with an add.v kind of instruction. But then, since there is only one functional unit that adds happens seriously in a serial manner. But the good thing is, there is not much of software assembly to execute. Only with that vectorize instruction, the entire pipeline understands that it has to operate A_1+B_1 . And then it has to update the value somewhere. And then it will operate on A_2+B_2 , update the value somewhere, it will do all the things, but in a serialized way, due to the absence of multiple vector processing lanes as it is a single lane execution.

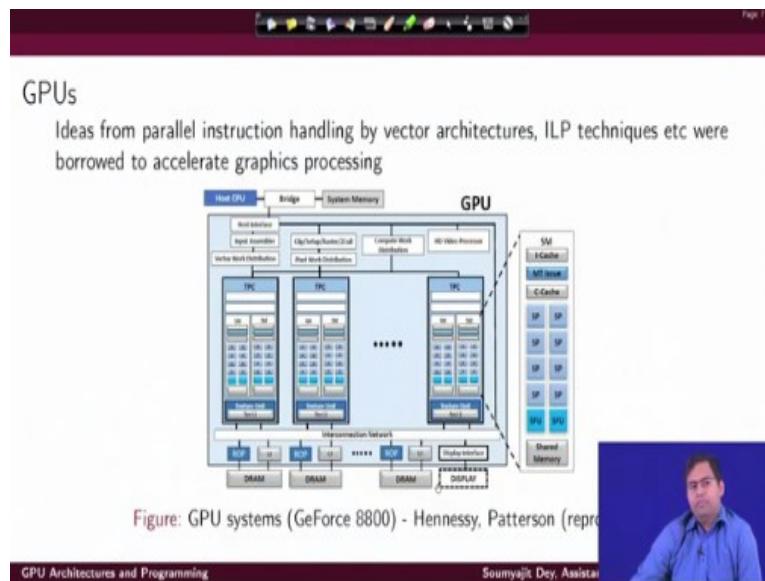
What is the other option. Now suppose I have Vector processing width just like earlier, I have vectorized registers of size 9, like this, A1 to A9.

But I do not have that much width in terms in terms of vectorize lanes, I do not have that many parallel functional units. Let us say I have this kind of 4 functional units. Then what should ideally happen is the inputs will get distributed, like this. So, maybe they will be distributed in this kind of interleaving fashion. So the content of the Bs let say they can be distributed by A4 followed by A5,A6,A7 again A8, again A9 like that.

So across a different functional units. So I can say that, okay, although I have Vector width of 9, but the functional units are present here are only 4. So I have some speed up, but of course the entire addition will not be a single cycle operation, it will take multiple cycles. Of course, in this case it will take less number of cycles with respect to this, but it will not get done by a single cycle.

Again just to repeat the good thing here. I have a smaller assembly to execute, the pipeline is being told that you have to execute a vectorized instruction. So the number of waits, are less, but how fast the instructions will execute depends on how many parallel hardware units are really present.

(Refer Slide Time: 22:35)



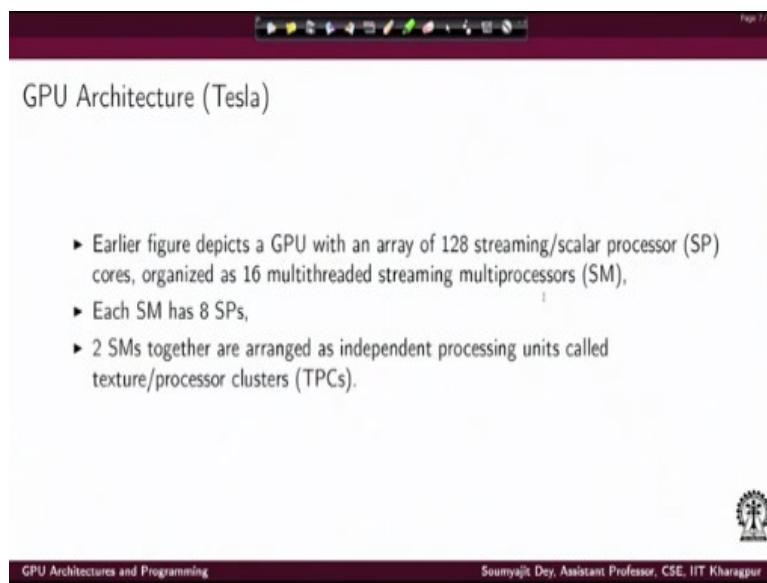
With this background on vector architectures, let us just have a look on GPUs and their basic architecture. So this idea of parallel instruction handling by vector architectures, as well as the earlier well known ideas like instruction level parallelism techniques. They were borrowed in the domain of graphics processing units. First, to accelerate specific graphics workload. And then, as we know that it has been found that they can accelerate most general purpose data parallel workloads also.

So this is a snapshot of our sample GPU system, one of the earlier ones. This is a snapshot of the GeForce 8800 system processor basic block diagram architecture. It is has been reproduced by me from the Hennessy Patterson's famous book. Again, which also has got a special chapter on GPUs. So, as you can see that the host CPU and the system's memory connect through a bridge.

And this is where the GPU will also be connected to. There is an interface with the bridge. And then there are certain functional units, which will be slowly processing. What is important to note here is, in typical GPU, you will have a hierarchy of processing elements. At the high level,

we have what we call as simultaneous multi processing units, the SMs right? And inside this SMs, you have specific processors, which are known as the scalar processors, the popular names for these are SM stands for streaming multi processors, and SP stands for streaming processors or scalar processors. There are two well known terms that are generally used. So they're not streaming multiprocessor, streaming multiprocessor is SM, the higher level in the hierarchy, and at the lower level, each SM contains multiple instances of scalar processors or in some books you will find the name streaming processors. So multiple instances of this streaming processor. And of course there are other units like the memory, the cache, the shared memory which we will be covering one by one.

(Refer Slide Time: 24:48)

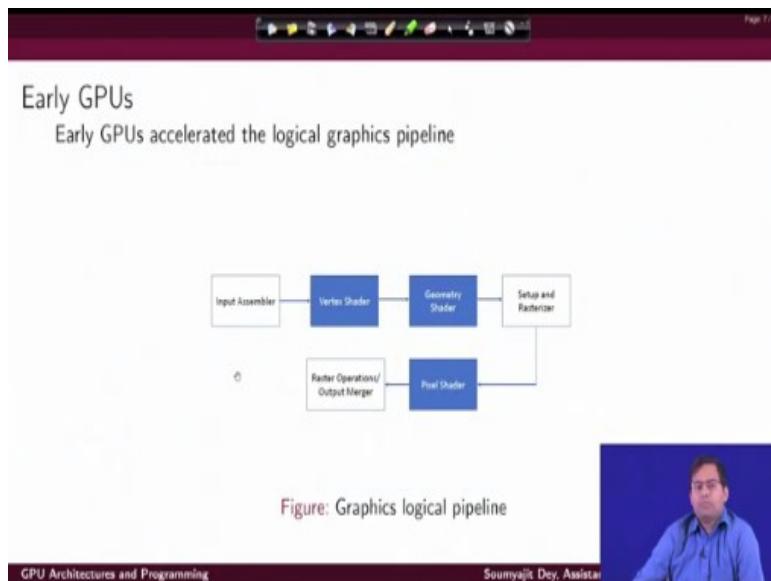


So, coming to this idea of GPU. So, if we take an example GPU architecture as we know that there has been an evolution of GPU architecture in the last decade. And there are several well known families of GPU architecture. So we start with one of the earlier ones, was also very popular, the Tesla architecture. So, the earlier figure depicts a GPU, the GeForce GPU, with an array of 128 streaming processors, or scalar processors course, as we discussed earlier.

And they are organized as 16 multi-threaded streaming multiprocessors. So if we go back to this figure, as we said that there is a hierarchy here inside this hierarchy. I have 16 number of what we call us streaming multiprocessors SMs. Inside each of them, we have eight scalar processors or streaming processors, the SPs. Overall I have got 128 streaming or scalar processor.

2SMs together are arranged as an independent processing unit called texture /processors clusters texture processor clusters, or in some books is also known as thread processing clusters. So these are clusters. I have these clusters, eight of them contain two SMs, and inside I have got 16 of the SPs. So in total I have got 128 SPs, the arrangement is, each SM contains eight SPs 2SMs together constitute one TPC.

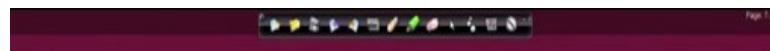
(Refer Slide Time: 26:43)



So, what really was the function of earlier GPUs. So, earlier GPUs were developed with a specific idea in mind that they should be there to accelerate the logical graphics pipeline.

So what's the logical graphics pipeline it represent the standard API, which is used for generating graphics for a computer. For that, there are a sequence of functionalities that need to be executed. And this is what the picture says, so we have the input assembler followed by the vertex shader block, followed by the geometry shader block, setup and rasterizer, and then the pixel shader and raster operations or, in some processor, also known as output merger. Now, these are the well defined significant components of the graphics pipeline, which together provide a standard pipeline through which computer graphics is usually rendered.

(Refer Slide Time: 27:48)



Graphics application sends the GPU a sequence of vertices grouped into geometric primitives—points, lines, triangles, and polygons.

- ▶ The input assembler collects vertices and primitives.
- ▶ Vertex shader programs map the position of vertices onto the screen, altering their position, color, or orientation.
- ▶ Geometry shader programs operate on geometric primitives (such as lines and triangles) defined by multiple vertices, changing them or generating additional primitives.

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor

So, the idea is the software programs which really do that, they have a well known name that is called shader programs. Graphics application send the GPU, a sequence of vertices, which are grouped into specific geometry primitives like points, lines, triangles and polygons. So, if you have an application and the application will try to render the output graphics on the screen. And for that it has to do the painting. Essentially it has to execute the computer graphics algorithm to create the bitmap image of that. For that, it will require to execute this kind of specific graphics pipeline, which has got the six logically divided parts. There are certain parts, which represent specific programmable parts, and these are the ones that we have highlighted here in blue. These are vertex shader, geometry shader, and pixel shader.

So Why is this sequence there? As we have discussed, the graphics application will send this sequence of vertices. Now, these vertices needs to be processed by the GPU, the vertices may be grouped into specific different geometric primitives, they can either be in the form of specific discrete points. They can be in the form of mathematical objects like lines or triangles, or polygons.

The input assembler is the block, which is going to collect these vertices and the corresponding primitives. The vertex shader is a software program, which will map the position of the vertices to the screen. Because finally is the screen where they need to be rendered? So this would mean

a specific change in the values, there will be a scaling and all that. And it will also mean altering their position color or their orientations.

After these vertex shader programs execution, comes the next colored block here, which is the geometry shader. Now the geometry shader program will operate on the geometry primitive. That means, it is going to operate on the specific graphics objects like lines and triangles, which are going to be defined again of course as we know by multiple vertices together, and they will change them or generate the additional primitives for them.

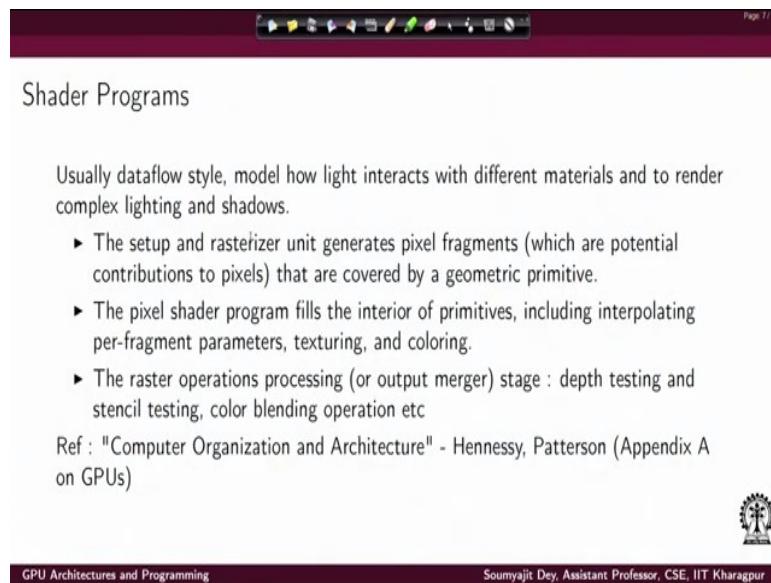
So usually, what is these shader programs I mean, how are they written and all that because as we can see that they are the sequence of three shaders, vertex, geometry and the pixel shaders. So essentially the shader programs are data parallel style of computation that means multiple parallel threads can work on different data points and do some computation in parallel. So we will stop here for this lecture and from this shader program and this style of computation we will pick up in the next part. Thank you

GPU Architectures and Programming
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 6
Intro to GPU Architectures (Contd.)

Alright, so we will start from where we left off,

(Refer Slide Time: 00:27)



The screenshot shows a presentation slide with a dark header bar containing various icons. The main title is "Shader Programs". Below the title, there is a text block and a bulleted list. At the bottom of the slide, there is a footer bar with the text "GPU Architectures and Programming" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

Usually dataflow style, model how light interacts with different materials and to render complex lighting and shadows.

- ▶ The setup and rasterizer unit generates pixel fragments (which are potential contributions to pixels) that are covered by a geometric primitive.
- ▶ The pixel shader program fills the interior of primitives, including interpolating per-fragment parameters, texturing, and coloring.
- ▶ The raster operations processing (or output merger) stage : depth testing and stencil testing, color blending operation etc

Ref : "Computer Organization and Architecture" - Hennessy, Patterson (Appendix A on GPUs)

Which was the parallelism available in shader programs, as we know that these are basic data flow style of computation, and the primary work of shader programs is to model how light interacts with different materials and accordingly to render complex lighting and shadows. So a typical shader program will kind of interact with several specific units that generate some specific components of output graphics.

So for example, there will be the setup and rasterizer unit that will generate the pixel fragments that are kind of potential contributions to pixels, and covered by a geometric primitive, the pixel shader program is the one that fills up the interior of the primitive. So the first one, generates the fragments to be covered by the geometric primitive, the interior will be kind of painted by the pixel shader program.

So that also gives a hint that this is the block which will be more compute intensive. It includes interpolating per fragment parameters, textures, as well as coloring. And by raster operations, we mean the stage where things like color blending, testing for stencil, as well as depth testing these are being done.

(Refer Slide Time: 01:50)

GPUs : massive multi-threading

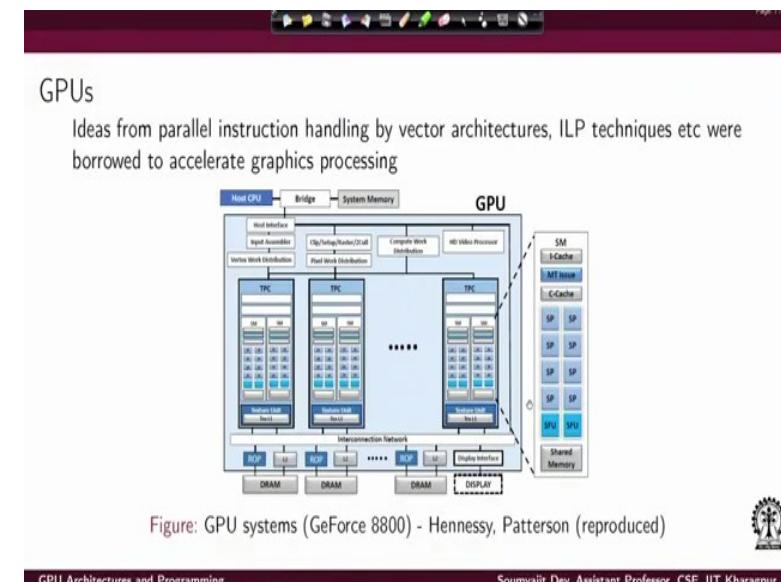
Design goals

- ▶ Cover the latency of memory loads and texture fetches from DRAM
- ▶ Support fine-grained parallel graphics shader (and general parallel compute) programming models
- ▶ Virtualize the physical processors as threads and thread blocks to provide transparent scalability
- ▶ Simplify the parallel programming model to writing a serial program for one thread¹

GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

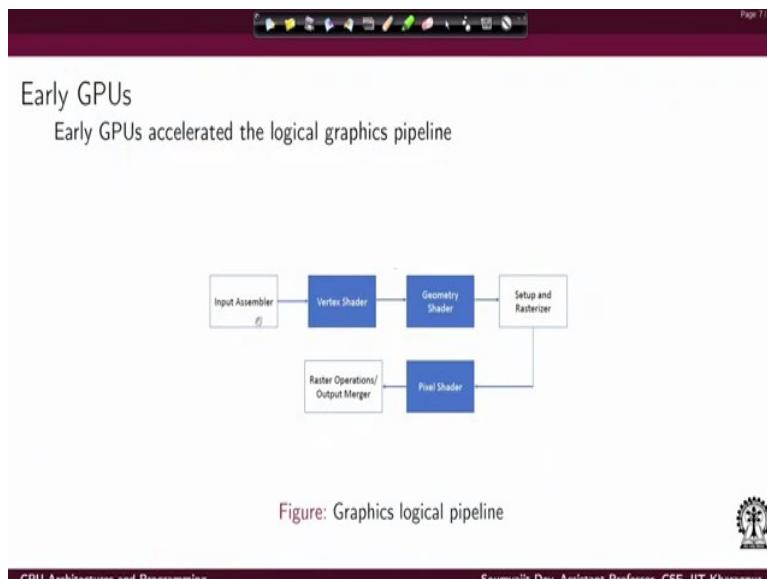
So these are also specific units, if you look at the picture here.

(Refer Slide Time: 01:55)



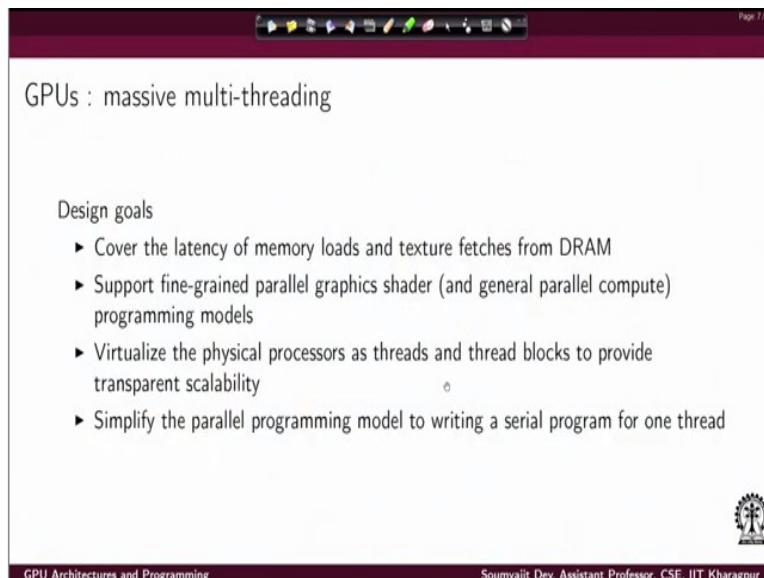
So you have the input assembler, the vertex work distributor, the pixel work distributor. And these are the units, which are present in this block diagram.

(Refer Slide Time: 02:10)



And they also nicely map to the graphics logical pipeline containing the input assembler, vertex shader, geometry shader, rasterizer, the pixel shader as well as the output merger stages. I understand that for a specific shader program we have the setup rasterizer unit, the pixel shader program, and the output merging, or the output processing phase.

(Refer Slide Time: 02:38)



So we will see that how this kind of graphics processing kind of nicely map to GPUs which support this massive multiple threaded computation. So for GPUs. The primary goal at its early age was to provide a solution, which was partly programmable. It was to accelerate the working

of this kind of a graphics pipeline. The reason being that you want the graphics to be rendered with as much resolution as possible as fast as possible. Because the higher rate at which you can render the graphics, the movements will be more realistic. So the overall goal was to cover the latency of memory loads and texture features from the DRAM. So in order to activate the graphics pipeline very frequently, you would like to fetch this data from the DRAM with that much high frequency as possible.

Or, you have to fetch in a large width, that means in each transaction you fetch a lot of data points and work on them in parallel. And that is how you can cover the latency of memory loads. This is what it means by covering the latency. And then comes the point that, well, you have fetched a lot of data points for the graphics pipeline to work or the graphic should have programs to work. You should be able to work on them in parallel and to their compute. So this helps to accelerate general parallel compute programs, but the primary goal was to accelerate graphics shaders in parallel. The third goal was to virtualize physical processors as threads and thread blocks to provide transparent scalability. Now this is an interesting point. We are trying to say that, see parallelism is not a specialized thing.

Rather let us look at every operation as paralyzed, think that that is the general case and try to accelerate parallel operations. That means, you always think in parallel, you start thinking that you have a thread of computation, where every computation has the ability to work on parallel on a set of vectors instead of scalar points. So with this design goes, this idea of GPU design started.

And also the overall objective comprise that you should be able to simplify the parallel programming model by simply making a programmer write a serial code that says about what one thread of computation is going to do. And it's just that you write a sequential behaviour of a single thread, and it's just a case that the thread works on multiple data points in parallel, for doing all the operations sequence theory.

(Refer Slide Time: 05:45)

First generation GPUs

- ▶ GeForce 256, introduced in 1999
- ▶ Contained fixed function vertex, pixel shaders programmed with OpenGL and the Microsoft DX7 API
- ▶ GeForce 3 - the first programmable vertex processor executing vertex shaders

- Ref for contents and here and subsequent places : "NVIDIA Tesla: A Unified Graphics and Computing Architecture" by Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym, (NVIDIA) IEEE Micro, Volume 28, Issue 2, March 2008

GPU Architectures and Programming Soumyajit Dey, Assistant Professor

So, what were the first generation GPUs. So these Geforce 256 was introduced in 1999. So one of the first GPUs. And the content fixed function, vertex pixel shaders, which were programmed with OpenGL and Microsoft DX7 API. So, the graphics pipeline for these GPU, still content fixed function vertex and pixel shaders that means they were not software programmable. These are hardware blocks which understood the lower level API calls from OpenGL and Microsoft DX7.

Then came Geforce 3, and this was the first programmable vertex processor which was executing vertex shaders. So what happened in them was this blocks of the graphics pipeline, the blocks highlighted in blue ie. a vertex shader ,the geometry shader and pixel shader. They instead of being fixed function blocks on the hardware, they became programmable blocks, that means the GPU will be having this general purpose programming interface through which it can run vertex shader also geometry shader problems also these pixel shader programs.

So for this, and all subsequent many contents at many places, we will be referring to this nice paper that came up in IEEE micro in the year 2008. The paper is titled ‘NVIDIA Tesla unified graphics and computing architecture’. So one of the position papers on this Tesla architecture that was published at that point of time.

(Refer Slide Time: 07:37)

Trade-off

- ▶ Vertex processors were designed for low-latency, high-precision math operations
- ▶ pixel-fragment processors were optimized for high-latency, lower-precision texture filtering - typically more busy (considering large triangulation)
- ▶ if these are fixed function blocks - difficult to select a fixed processor ratio
- ▶ Primary design objective for Tesla architecture - execute vertex and pixel-fragment shader programs on the same unified processor.
- ▶ Unification helps in 1) dynamic load balancing of varying vertex- and pixel-processing workloads, 2) introducing other shaders



GPU Architectures and Programming

Soumyajit Dey, Assistant

Now, what was that trade off that the designers of this Tesla architecture, or the Tesla family of GPUs that we are really looking at. So, if you look at what is the activity of a vertex shader program, which was to be run earlier by vertex processors, as I said, the fixed functional blocks which would be part of the early GPUs or early, early graphics systems. That there will be a specific fixed function hardware block which will do vertex processing, there will be a fixed function block which will do the pixel shader processing and all that.

So for them. They have quite well defined objectives. The vertex processors will operate on coordinates, but they will implement high precision math operations at low latency. So that was the design objective of a fixed function vertex processor. When it came to the pixel fragment processing, they were optimized for high latency, but low precision texture filtering. They are typically more busy, considering large triangulations.

So, if I consider large triangulation of an image, there will be a lot of pixel points inside specific triangle. So processing at the pixel level will be having a lot of compute load. For that there is no need to be done at a high precision level, but the objective is to do more of them in parallel.

Now, if this vertex and pixel processors are implemented as fixed function blocks. It is difficult to select a fixed processor ratio. That means how much throughput should be allowed for the vertex processor and how much throughput should be allowed for the pixel processor, because

overall how much this pipeline will work at what is overall throughput that also depends on the input data or images and triangulation scheme. So, if it's really has been as thought here ie. large triangulation, then the vertex processors will be operating on lesser number of points.

So this idea would hold that yes it can work on lessor number of points but compute in high precision, whereas the next level will have lot of things to do per triangle. So they work on low precision. But how about the triangulation is considered in input image as small, so then the vertex processor will have a lot of things to do, because there are a lot of triangles in the input, lot of vertices to process, but the for the pixel process and per triangle job is reduced, right?

So, if they are implemented in hardware as fixed function blocks, what is an ideal image scenario. I mean, the overall throughput of this sequence of fixed function blocks would actually be a function of how the input image has been triangulated and how it has been provided. The common case may get accelerated, but the cases which are not common with suffer from performers in video.

So, the primary design objective for the GPUs in the Tesla architectures was that for different vertex and pixel fragment shader programs they need not be implemented as fixed function blocks but they should be implemented uniformly on a programmable processor, such that the same processing fabric can be used for both of them. So that, depending dynamically on the input triangulation, it can be decided how many threads will be doing the computation for vertex processing and how many threads will be doing the computation for pixel processing so that overall we have a very high throughput. Now this unification helps in the dynamic load balancing of these varying throughputs of vertex and pixel processing.

And it also helps in introducing many other possible shaders. That can be part of the graphics pipeline. As we know this is always the case when you move some hardware functionality to a software functionality with increased programmability you can do more dynamic load balancing things. So essentially, you move out from this fixed fixed function blocks. Make them all part of a programmable parallel processor ie. the GPU, and you just dynamically decide how many

threads will be computing for the vertex or the pixel, what is good for the overall high throughput functionality.

(Refer Slide Time: 12:18)

Tesla architecture

We come back to GeForce 8800 GPU with 128 SPs organized as 16 SMs

- ▶ external DRAM control and fixed-function raster operation processors (ROPs)
perform color and depth frame buffer operations directly on memory
- ▶ The interconnection network carries computed pixel-fragment colors and depth values from SPs to the ROPs
- ▶ The network also routes texture memory read requests from the SP to DRAM and read data from DRAM through a level-2 cache back to the SPs

GPU Architectures and Programming Soumyajit Dey, Assistant

So coming to this Tesla architecture. We speak of, again this GeForce 8800 GPU with 128 SPs that are organized as 16 streaming multiprocessor or SMs. So in this specific GPU, you have an external DRAM control. You have fixed function raster operation processors or ROPs, which perform color and depth frame buffer operations directly on the memory. So, given that you have this ROPs present it is possible to do this specific operations on color and depth, directly on memory, instead of bringing them for doing some computation on the main processor pipeline. The interconnection network carries these computed pixel fragments, colors and depth values from the SPs to the ROPs. So, just to get a feeling of the connectivity. If we look back here into the picture that we had for our GPU.

(Refer Slide Time: 13:29)

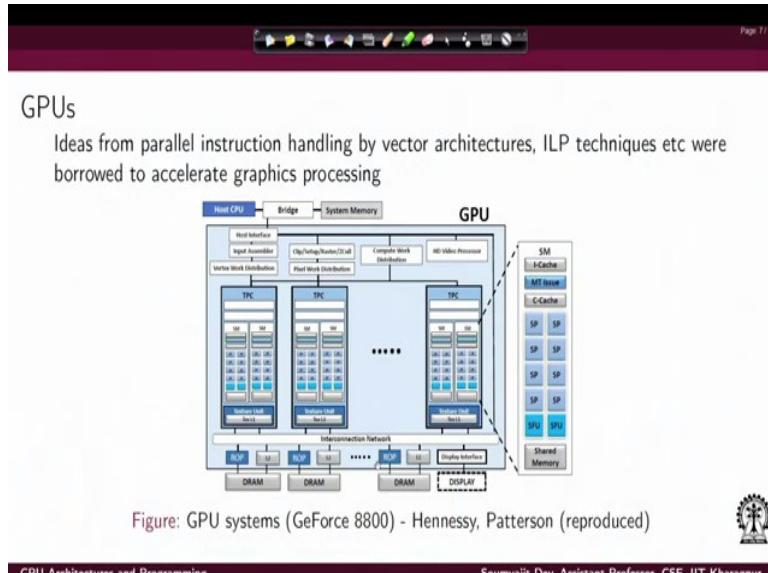


Figure: GPU systems (GeForce 8800) - Hennessy, Patterson (reproduced)



GPU Architectures and Programming

Sounyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So we have this ROPs being connected with the interconnection network. And also, so the interconnection network is here connecting the different TPCs, which contain the SMs and also the SPs. And this interconnection network also connects the ROPs, and they connect to DRAMS through this hierarchy of level two caches.

Now coming to our point here. So, this external DRAM control, and the ROPs. They perform color and depth frame buffer operations directly on the memory, the interconnection network carries the computed pixel fragment colors, so they carry the output pixel fragment colors and depth values from the scalar process to the ROPs for doing the fixed function raster operation. And then they get stored back to the memory. The network also routes texture memory read requests from the scalar process to the DRAM. And they read data from the DRAM through a level two cache back to the SPs.

So, these ROPs still remain as fixed function blocks. And they do their operations and they store back the data to the memory, back and forth to the hierarchy of L2 to caches. And the interconnection network carries this computed pixel fragment colors and depth values from SPs to the ROPs back.

(Refer Slide Time: 15:26)

Page 71

Graphics in Tesla

- ▶ The input assembler collects vertex work
- ▶ Vertex work distributor distributes vertex work packets to the various TPCs
- ▶ The TPCs execute vertex/geometry shader programs
- ▶ output data is written to on-chip buffers
- ▶ buffers then pass their results to the viewport/clip/setup/raster/zcull block

We continue from here to general purpose processing

GPU Architectures and Programming Soumyajit Dey, Assistant

So coming to the graphics in the Tesla what are the different blocks that are present in the architecture. So, first of all, there's the input assembler which collects the vertex work to be done. And then you have the vertice work distributor, that distributes this work, packets to the different TPCs that are present. And this TPCs that are going to execute the final shader programs. both the vertex as well as the geometry shaders.

Now this is where as we already spoken that the programmability comes in. Earlier vertex and geometry shaders also used to be fixed function blocks, but now they become a part of the programmable pipeline. So, with this. Just to finalize once again, you have this raster operation still in fixed function blocks ,the data flows between these ROPs and the DRAM through this level two hierarchy.

And the interconnection network will carry the fixed fragment colors and depth from the SPs to the ROPs for doing the rester operations. Of Course the SPs are there to execute the different shader operations . And coming back here, the TPCs will execute these sheder operations and output data will be returned to the on-chip buffers, Which will further be passing the result to the viewport, or the raster block.

So this is how as we can see that the original graphics pipeline gets represented in the Tesla architecture with the shader operations, being done in the scalar processors, whereas you have

the raster operations, being done in the fixed function blocks, and also the buffers, the over all outputs are written to on-chip buffers and they pass the result to the viewport. So, this gives us a basic idea of how graphics processing changed from fixed function pipeline to a GPU pipeline.

And from here we will continue to how GPUs, in general, handle instruction processing in more and more in the SIMD style. And we will talk about general purpose programs.

(Refer Slide Time: 18:03)



GPGPU

Each TPC has two SMs, each SM has

- ▶ eight streaming/scalar processor (SP) cores,
- ▶ two special function units (SFUs),
- ▶ a multi-threaded instruction fetch and issue unit (MT Issue),
- ▶ an instruction cache, a read-only constant cache,
- ▶ a 16-Kbyte read/write shared memory.



Coming back to the way the different functional units in the GPU. So you have each of these TPCs. They contain two streaming multi processors (SMs). Each of the streaming multi processors have got eight streaming or scalar processors (SPs). So these scalar processors or SMs will also have two special function units or the SFUs. And there is a multi threaded instruction fetch and issue unit.

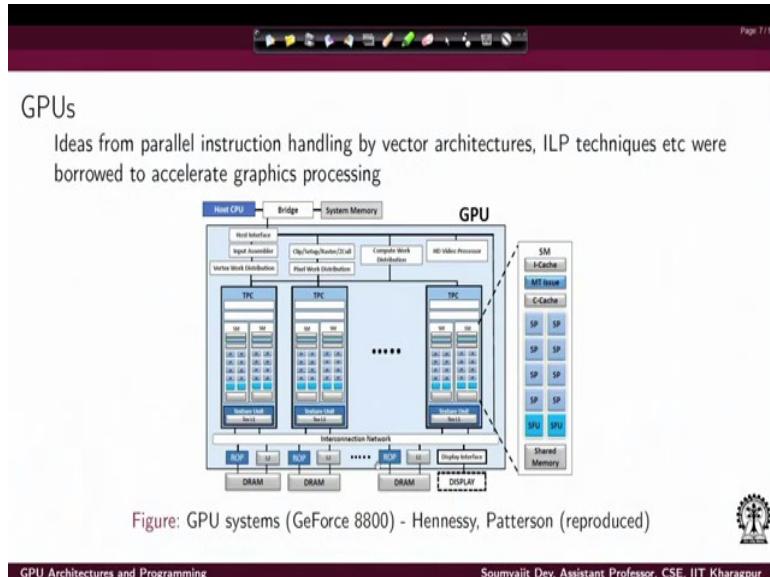


Figure: GPU systems (GeForce 8800) - Hennessy, Patterson (reproduced)

GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So again if we just go back to this figure. So, as we are back you can see, we have this input assembler like we discussed from the graphics pipeline, then the vertex work distribution, which will distribute work, and also the pixel work distribution that distribute the pixel works to the different TPCs. And finally we have the ROPs as fixed function blocks. Also, the texture units as fixed function blocks and outputs finally go to the viewport.

Now, coming here to the internal content of each of the streaming multiprocessors. So you have the I- cache for the instruction fetching instructions. And then you have this multi threaded issue unit, which is going to issue instructions in parallel to each of this scalar processors or streaming processors.

As we have discussed each SM has got eight scalar processors. And also, two special function units or SFU. So, there is a multi threaded instruction fetch and issue unit that are supposed to issue the instructions that are going to be executed by the scalar cores

And also, as we have seen in that picture we have a 16 kB read write shared memory. As part of each streaming multiprocessor. So, what is there inside each of these streaming multiprocessors In summary, each SP core will contain a scalar multiply and add unit. Does that mean inside each streaming multiprocessor you have got eight scalar multiply and add units. Also the SM. as

we have seen, it has got to have these special function units. And this special function units can be used for doing some transcendental computation.

Also, each of these SFUs contains four floating point multipliers. So overall I can say that inside each, the streaming multiprocessor I have got eight multipliers and add units. And also, eight floating point multipliers, I can do eight floating point multiplication operation in parallel. And also, eight scalar multiply operations in parallel.

(Refer Slide Time: 20:39)

SIMT

GPU execution model

- ▶ SIMT architecture is similar to SIMD design, which applies one instruction to multiple data lanes.
- ▶ The difference is that SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes.
- ▶ A SIMD instruction controls a vector of multiple data lanes together, a SIMT instruction controls the execution and branching behavior of one thread.

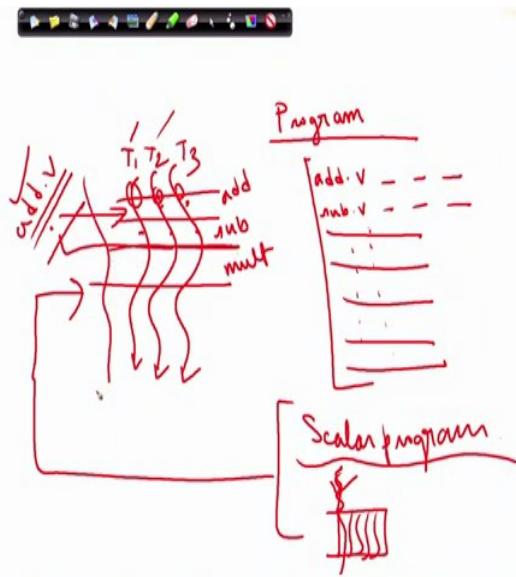
GPU Architectures and Programming Soumyajit Dey, Assistant

Now, coming to the execution model. So, as we have seen this idea of data parallel computation by instructions is well known as a single instruction multiple data (SIMD). This is the idea. I mean, which amended it also from vector processing. And this idea, gives much more generalized when people thought of the execution model for GPUs. So what people started talking about is single instruction multiple threads (SIMT).

So that would mean that you have multiple threads of computation, or in a different way, I would say that I have a single thread of computation. It's executing a sequence of instructions, each instruction is working on multiple parallel vector type data points. So looking at the other way I can just say that. Okay. Since I have a single instruction processing, but is doing the same job for different parallel threads of execution.

So that gives me that idea of what is SIMT that is instead of saying single instruction on multiple data. If those data points, all belong to parallel threads which are making progress in parallel, I would start calling them single instruction multiple threads. So, with single instruction multiple threads. I have a more generic model of instruction processing, because essentially what I am saying that, okay. To let us let us just take a simple example here.

(Refer Slide Time: 22:21)



So, when I am executing a sequence of vector instructions. In general, they may or may not be related. That means. Well, they can be doing computation towards a similar objective, or they may not be doing so. I mean, it depends on how the processor has figured out the parallelism, and it has found that Okay, I have multiple vectorised instructions and I can process them in parallel and all that.

But now suppose I say. Okay, I have multiple threads of execution. So this is thread 1, thread 2 and this is thread 3. And I say that all the threads are going to do an add operation here. And then all the threads are willing to do some subtract operation here. And then, all the threads are going to do some multiply operation here. This is the demand of the program. When I say that, looking at it from another way I can start saying that okay.

For this unit a SIMD instruction. Because well. we are looking to do an ADD.V processing, this a vector processing. But when can you do that, only if the warp for each of the threads are such

that they are similar, this thread also needs to add, this thread also needs to do an add, this thread also needs to do an add.

Once that is done, this thread will do a sub this thread will do a sub and this thread will do a sub. And not only that, when, this thread is doing the add, and the next thread is also doing the add. They are doing the add on consecutive items, they are all doing the work on specific vectors. So, just when I start looking at a program where all instructions are vectorized is not that I have a program, which is basically a scalar program. And for extracting parallelism purpose. The compiler had figured out that okay here, something is done in parallel. And then again, there is some vector processing that can be done. It's not like that.

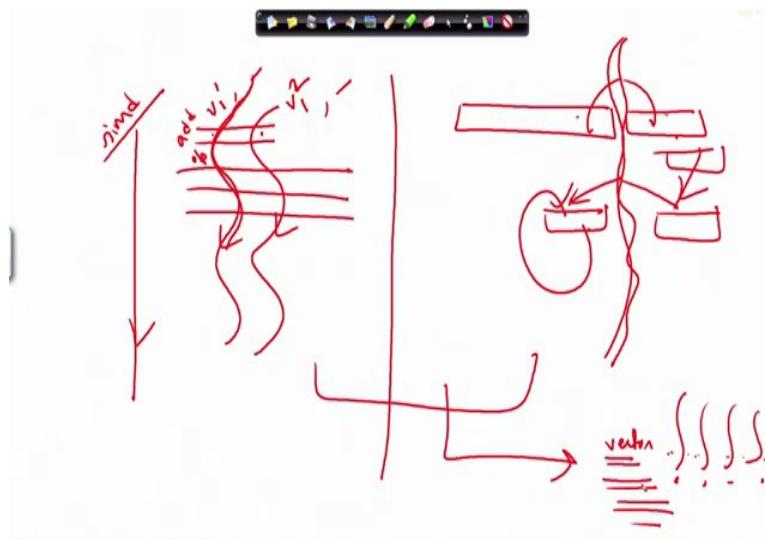
We are going to generalize this concept. And we are starting to say that no, the entire program is like a vectorized operation. Just start looking at it in a way that with each operation you have a set of threads that are making progress. Then I can say that I have a sequence of these kind of instructions or single instruction, which is operating on multiple threads. Once that is done, again I have another such instruction which will again operate on multiple threads. And that is how the computation progresses.

So coming to SMIT. Instead of starting to think just to just doing a basic summary again, because this is the most important thing here. Instead of saying that, okay. It's not that you are doing some program execution, where some part of the code is scalar. Some part of the code, the compiler has identified to be data parallel and hence has emitted some vector instructions. Let us generalize that. let us think that every instruction execution is of some type vector. And not only that, the entire sequence of instructions is such that they represent the progress together by multiple parallel threads of computation.

When we start talking like that with mean that okay, we are thinking of a programming model, where I have a sequence of instructions, which is such that every instruction is like an SIMD instruction. Not only that, every instruction is doing an SIMD operation which is part of some set collection of threads. Then again, there is another SIMD instruction which is again making progress with the operation for that specific collection of threads.

And in that way things are going on. So in that way, what we have is a nice data parallel program. And that data parallel program has got multiple threads of computation. And they are all being executed synchronously by a sequence of instructions. Alternatively, I can look at it, just like this, that, okay I have a single thread of execution, but in each point of execution is working on multiple different data points. So just to make things very clear

(Refer Slide Time: 27:55)



And not to get confused. I can say that these two pictures are quite equivalent. I have two threads, which are executing like this. But whatever they do here. There is an add on vectors so there is vector component v_{11} and vector component v_{12} . And then again, so there is the first operand, and then there is some second operand. They're doing an add. Then again, they are doing some other operation, like that.

So I can say that I have a sequence of SIMD operations going on, on multiple different threads in parallel. Alternatively, I can say that I have data, sitting in vector datatypes. There is one thread going on which is going on, and doing a computation on them, producing a result. And then again it is doing a computation on some other vectors. And again, producing a result, while there may be dependencies and all that.

So there's less like thinking that I have a single thread, which is making progress, doing operations on parallel data vectors, or I can think that I have multiple threads, which are

making progress in parallel. But both of these pictures at the instruction level, maps to a scenario that I have a vector instruction, which is doing computation for different data points, belonging to these different threads followed by another vector instruction, doing some other computation to this same set of threads, and they are overall making some progress.

So just to summarize, that when I speak of SMIT, I can say that it's about one instruction, getting applied to multiple independent threads in parallel, not just multiple data lanes. And one SIMD instruction that controls a vector of multiple data lane together, Whereas an SIMT instruction controls the execution and branching behavior of one thread.

So, this may sound confusing, but this is what it means, when I talk about SIMD instruction. It controls a vector of multiple data lanes together. So, you have vector type sitting on multiple data lanes and the same SIMD instruction is operating on them right. When I say SIMT, it controls the execution and branching behavior of one thread. Because when there is a branching happening. These different threads may have different behaviors. That is why you have this instruction, which would mean something for one of the threads, but it may mean something else to another thread. I have one SIMT instruction which will mean something for one thread but, essentially also going to mean something completely else, a different branching style for some other thread. This is something we will explain in more detail later on. For the time being, just remember those two pictures that we were discussing. The overall idea is very simple. When I talk about SIMT, it is just like saying that well lets generalize this concept of single instruction multiple data and start thinking that you have instructions operating across multiple programs or their lightweight versions ie. threads and all the threads are making progress in log steps That means, fundamentally, all your instructions are SMIDs. So you have one SIMD instruction operating on different program threads. Again, another SMIDs instruction operating on different programs threads, so on so forth. And this is how it keeps on continuing.

(Refer Slide Time: 32:04)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'SIMT'. Below it is a list of bullet points:

- ▶ In contrast to SIMD vector architectures, SIMT enables programmers to write thread level parallel code for independent threads as well as data-parallel code for coordinated threads
- ▶ SIMT - essentially a single thread of SIMD instructions
- ▶ Each SM's multithreaded instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*
- ▶ Each SM manages a pool of 24 warps, with a total of 768 threads
- ▶ Each SM maps warp threads to the SP cores

At the bottom left is the text 'GPU Architectures and Programming', and at the bottom right is 'Soumyajit Dey, Assistant Professor'. A video player interface is visible on the right side.

So, in contrast with SIMD vector, SIMT will enable programmers to write thread level parallel code for independent threads, as well as a data parallel code for coordinated threads. Lets understand what we mean. So, as we have said, this is SIMT, that means a single instruction working across multiple threads. But when you look at the behavior of the program, you have multiple threads.

But you need to specify the functionality of only one thread, how that thread works for its data points and all that right? So, I can say that it enables programmers to write the thread level parallel code for one thread. If different threads are supposed to do different jobs, then there will be some more complex code to write, which we will see how that works. But ideally you are going to write a per thread behavior and that is going to be replicated across the different parts of the data segment. So in that way I can say that it is essentially a single thread of SIMD instructions, so that's the alternate way I was talking about. I have essentially a single set of execution, but all the instructions are executing as a SIMT. So each of these streaming multi processors multi threaded instruction unit.

They are going to create, manage and schedule and execute threads. So they will spawn the threads, They will manage their execution, they will map them to the real hardware that is the scalar processor or SPs, and finally they will say that okay these are the threads that are executed

they have committed and these are the threads that have to start and all that. So, this multi threaded instruction unit that what we showed in the last picture.

This is the thing that is going to do this is also known as a hardware scheduler we will talk about that again later on. But the important thing is this management of threads are done in blocks of 32. 32 threads together are packed as something called a warps. So this is a basic atomic unit of parallel execution in a GPU. So when I said it's a warps that means, that I have 32 parallel threads , they together are making progress in the SPs.

So when a warps makes progress by executing an add instruction. That means all those 32 parallel threads have executed the add instruction. Each SM, now this is specific to that architecture that we've been talking about. It manages a pool of 24 warps with a total of 768 threads, again this is specific to GeForce architecture. Each SM maps warp threads to the SP cores. So, as you can see that there are lot many threads to manage.

So essentially, there are 768 threads, they are, they are being distributed across 24 warps 768 / 32, ie. 24 warps, but each SM will map this warps to the SP codes, how that is done is also the job of the SM. Now of course, and I will just repeat these are the numbers that are specific to the GPU architecture we've been talking about. With different GPU cards, these numbers keep on changing and they become much bigger with modern GPUs.

(Refer Slide Time: 35:32)

Page 12 / 15

Warp execution

- ▶ In each operation cycle, the SM warp scheduler selects one of the 24 warps
- ▶ An issued warp executes over four processor cycles
- ▶ The SP cores and SFU units execute instructions independently

GPU Architectures and Programming Soumyajit Dey, Assistant

So execution of warps. In each operation cycle, the SM warp scheduler will select one of these 24 warps. This is the synthetic example I have 24 warps. One of them will be picked up, and for each cycle of activation of the SMs warp scheduler, it will select one of these 24 warps and map it to SPs so that the SPs can process that warps. An issued warp executes over four processor cycles.

The SP cores, and the SFU units, execute instructions independently. Of course the SP cores have got the scalar multiply and add units, eight of them, And there are two SFU units in each of them, and they contain in total, eight floating point multiply and add units. So they are operating independently. So with this we will stop here and we will start again with more details of how the SPs execute the warps in the next lecture. Thank you.

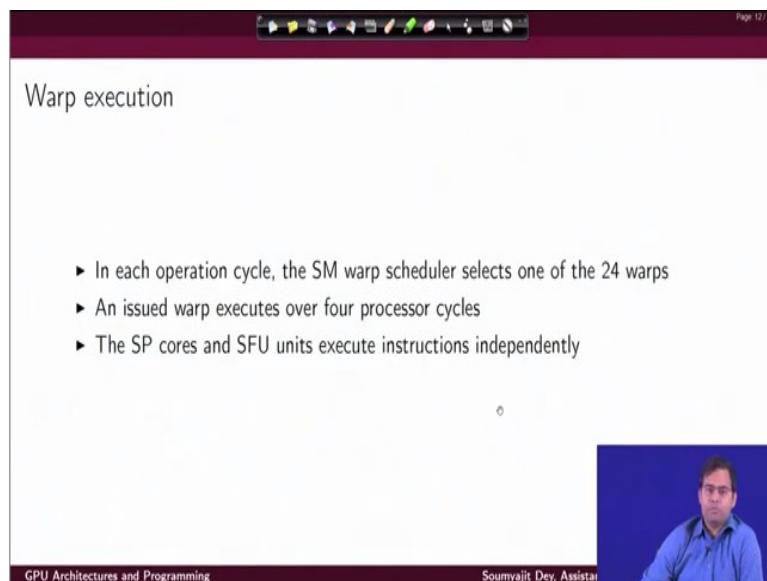
GPU Architectures and Programming
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 7
Intro to GPU Architectures (Contd.)

Hi, So, welcome back to our lectures on up we are hitting certain programming. So we have been discussing this SIMD model of computation. And based on that, how the different threads that are launched via kernel on a GPU kids can use. And as we have seen that threads, get packeted in the form of 32 parallel threads running executing together as what we own as warps and this warps essentially getting mapped to the circular process of course.

And the GPUs multi, I mean multi level scheduler is responsible for scheduling this warps into the processor architecture as we shall see.

(Refer Slide Time: 01:10)



Warp execution

- ▶ In each operation cycle, the SM warp scheduler selects one of the 24 warps
- ▶ An issued warp executes over four processor cycles
- ▶ The SP cores and SFU units execute instructions independently

GPU Architectures and Programming Soumyajit Dey, Assistant Professor

And so coming to warp execution, as we have discussed earlier, that for this specific GPU, the SIMD warp scheduler selects, one of the active 24 warps and accordingly. This warp is executed. And, of course, at a time they are many many parallel warps that are active which also depends on the size of the SM And the size of the and the number of SMs in the system and all that. So, how does we warp really execute.

So, for the example system that we have taken here since one warp will have 32 parallel threads is and being executing. And there are essentially 8 SP course. So, an issued warp will execute over 4 processor cycles. And of course, there will be the SP cores feature comprising the interior ALU, and, and also the floating point units. And also there are the special function units.

Which are separate from the SPcores, and they are going to execute those the corresponding instructions independently. So, coming to the GPUs, ISA

(Refer Slide Time: 02:18)

ISA

- ▶ Support for floating-point, integer, bit, conversion, transcendental, flow control, memory load/store
- ▶ Floating-point and integer operations include add, multiply, multiply-add, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers
- ▶ Transcendental function instructions include cosine, sine, binary exponential, binary logarithm, reciprocal, and reciprocal square root.
- ▶ Bitwise operators include shift left, shift right, logic operators, and move
- ▶ Control flow includes branch, call, return, trap, and barrier synchronization

GPU Architectures and Programming Soumyajit Dey, Assistant Professor

Like, what are the different kinds of instructions that are supported by the execution model of a GPU. So as we know that this is to be defined by the instruction set architecture of the graphics processing unit, And the instructions that architecture specifies what are the different classes of instructions that are going to be supported. So it happens that there is support for a lot of floating point operations, apart from standard integer and bit level operations.

Also, specific operations like transcendental operations are supported. And there are also specific instructions which control the flow of execution into the GPU is some topic that we will get into in more detail later on. Of course there are also instructions for doing memory load and store of data points. Now, what are really the floating point and integer operations that get executed of these are pretty similar to standard processor instructions.

With respect to addition, multiplication. There are also fused multiply add units. So you have one instruction which will do the multiply add. There are instructions for performing. I mean, minimum, as well as maximum value extraction comparison instructions and set predicate instruction. So there is something also which is very important with this the two instructions flow control, that is also something we touch upon in more detail later on.

And also there are instructions which do the conversion between integer and floating point numbers. And as we have discussed earlier, that we have inside the GPU Apart from this scalar process we also have the special function units which take care of executing the transcendent and functions. That means functions for which you do not have a nice codes on, but there are some. I mean, this our standard numerical algorithms.

We could actually implement their approximate versions, for example. Cosine Transform, sine , binary exponential binary logarithm as we know that in terms of algebraic expressions that exact values will equal to compute an infinite series of terms. But inside the standard ecosystem, the way they are implemented is you have a numerical algorithm which does an approximate computation which is good enough.

In terms of the number of pieces that are provided to the represent the value. So, you have transcendental constant instructions for all the trigonometric functions, then binary exponential binary logarithm computing reciprocal as well as reciprocal square root. With respect to bitwise operators, you have the standard shift left shift right logical operations, and also move instructions. You can also do control flow execution, you can actually have instructions.

Of course you need them in any standard barrier by saying you need instructions which will manage the control flow that is managing branches managing function calls, managing returns the, trap, and also something known as barrier synchronization that is very useful for handling parallel threads in, in any kind of parallel programming interface, it's may be GPU or its may be MPI or something else.

(Refer Slide Time: 05:45)

Register File

Each SIMD processor (SM)

- ▶ has a large vector register file
- ▶ like a vector processor, these registers are divided logically across the SIMD Lanes, i.e. the SPs
- ▶ These numbers vary across across architecture families.

GPU Architectures and Programming

Seumyajit Dey, Assistant Professor

So what about the register file, like we know that in any processor you have a register file which contains the registers called the data, which immediately needs to the ALUs, and so that for doing any kind of a new operation. The operands have to be present in the register, otherwise they have to be brought from the main memory to the cache. So, every SM has a large vector register file.

So it's like a vector processor is registers are divided logically across the SIMD lanes, that is a SPs. So as we have seen earlier, in our small discussion on Vector processors you have Vector registers, that means a register which can hold an array of values, the same type. So similarly like that. I also have a big register file inside on SM, and the register certainly divided logically across the assembly lines.

So that SPs, kind of, present the individual scalar computer elements of a large vector. And then, what are the values of the registers, that is something that keeps on getting across architecture families with the older GPU architecture we had smaller numbers. But with the newer GPU architectures definitely the number of registers part of them, keep on increasing we will have some figures on this later on.

(Refer Slide Time: 07:07)

Now coming to the second architectural example so we move from Tesla, to the fermi family of NVIDIA architectures, which provided some new facilities in terms of processing. So coming to an example. We decided to Fermi GTS 480 GPU. These are representative example of the Fermi family one of the earlier examples. It has got 16 SMs. Together they can process, 512 CUDA cores.

So you have these 512 CUDA cores inside the system itself segments each of them has got 32 of these CUDA cores, 32 scalar process. And you have got this 32 768 number of 32 bit registers divided logically across the executing threads. Inside each SM. So, I have each SM is comprising this 32 SPs. And these many 32 bit registers. So, if I look at the system from the perspective of a single SIMD thread. It is limited to no more than 64 registers.

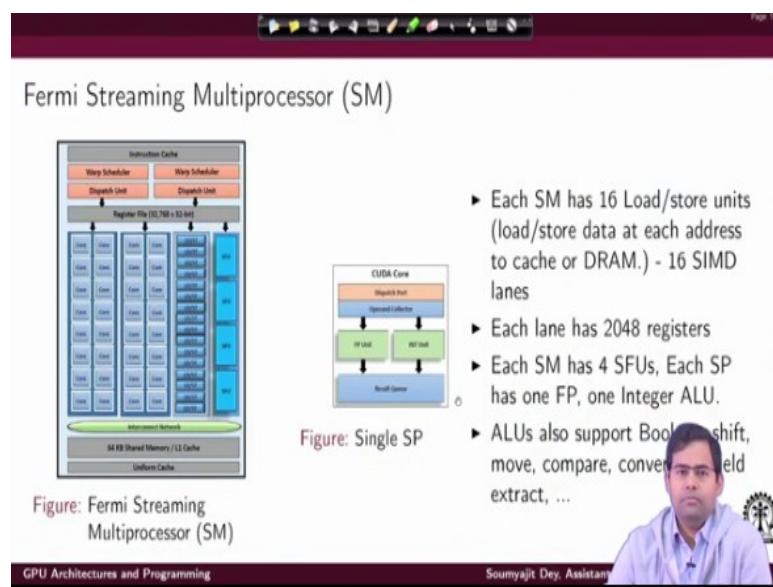
A warp has access to 64 times 32 registers, of course inside a warp, I will have 32 such threads. Since each thread has got an access to 64 registers. When warp is executing inside a GPU. It has got access to this overall number of registers. Of course, each of these registers a 32 bit. So, just to get to them values again, overall I have this many number of registers for a single warp. I have access to this 64 cross 32 registers, each of which are 32 bit.

However, they can hold different kinds of data types right? So I can use the 32 bit registers to hold 32 bit data. Also, if I am doing operations on double precision, floating point operations. I

need to 64 bit data values. So we might consider this kind of double precision floating point operations, then I would start saying that warp has access to 32 vector registers. And each of these vector registers have 32 elements.

And each of these elements have 64 bit wide, so essentially this figure of 64 cross 32 registers which are 632 bit changes to 32 vector registers each of them are 32 wide. So, each of them is 32, the width is 32. So, essentially 32 cross 32 registers which are 64 bit while the same register file can operate in both the modes.

(Refer Slide Time: 10:03)



So this is an example picture of our Fermi family SM. So, this is a streaming multiprocessor example from fermi family. As we can see, the cores are all, the SP course can be seen here. So, as we say that each SM has 32 SPs. In total there are 16 SMs, we are having a deeper look into one of the SMs right? So inside one SM, I have this kind of 32 SPs. I have got this 32 SPs. And there are 16 of the load store units.

We do memory load and store via the cache to the DRAM. So each SM has got the 16 load store units, essentially, is contributes to 16 SIMD lanes. And each lane has got these 2048 number of registers. So essentially you divide these many registers across this 16 SIMD lanes. So you will be laned up with 2048 registers for SIMD lane. Each lane has got one low storage unit loading data points on the, on the DRAM.

Now, if I look at to the other functional units that are present in the SM, then I have got 4 special function units as we can see that they are not directly integrated into this pipeline. So I can see that if I can have an alternate loop into this figure from a horizontal way, then I can start saying that okay for every load/store unit. I have got 2 cores there. That is like an one SIMD lane. I have got 16 SIMD lanes.

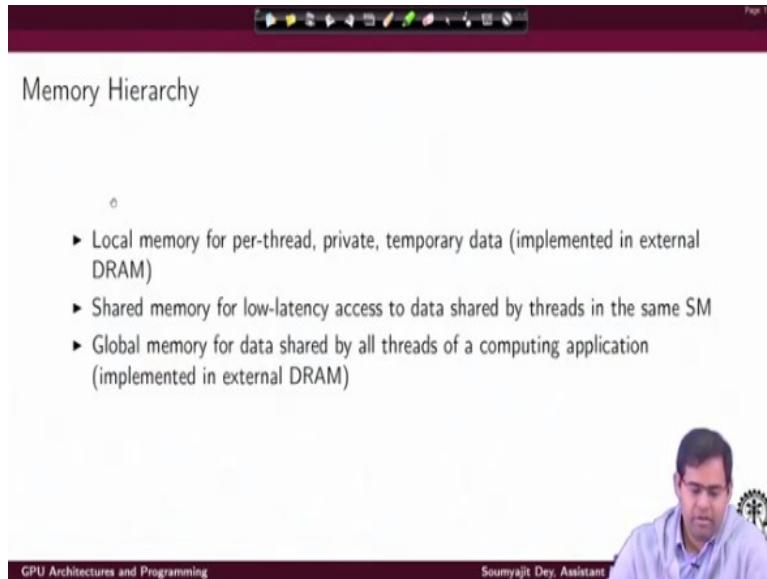
These issues are sitting there, they are not directly integrated into each of these lives, but of course for operations. They are available. And I can have in the best case 4 possible special function operations going on. If I take a closer look into each of the scalar processors. So there is the figure we have here. So in this bigger as you can see that you have got inside it so there is the dispatcher, there is a result queue.

And inside this CUDA coree that is the scalar processor. You have one floating point unit, and one integer ALU. So these operations are supported by these units, you can have an integer operation. You can also have a floating point operation here inside the core. The ALU also support standard Boolean, shift ,move, compare convert kind of values. Extracting specific bit fields from an input value in a register and all that.

So that is about the computation part in the SM right? So I have these many compute cores. Inside the SM. Just to summarize, there are total 16 SMs each SM was got 32 SPs, there are 16 load/ storage units, 16 SIMD lanes and this is a over all register file inside the SM. And this register file gets divided across the SIMD lanes. So, each lane gets 2048 number of registers. And if I start looking at the point from the execution view of a warp.

Then the warp has access to these many registers which are all 32 bit. I mean, it has got access to this many registers out of this total. But they can also be configured in a different way and I can say that it also that says to 32 vector registers. Each holding 32 elements which are 64 bit wide.

(Refer Slide Time: 13:57)



Now coming to the memory hierarchy of the SM. So as you can see, here I have the organization of the compute units. Also, there is something called a shared memory, but it's also written as an L1 cache with an oblique, and the amount of it available is 64 kilobytes. So what is the shared memory. So, in a memory hierarchy. The memory is organized from the programming point of view into the following parts. So there is a local memory par-thread.

So this is the private temporary data, which is booked par-thread in the external DRAM outside this setup. SMs I have one big DRAM here, to which things are connected through a interconnect. For computation by each thread in the SPs. We have a specific set of registers available as we have discussed that there is a specific set of registers available for warp inside, the warp I have gotten a state of history cores that are engaged.

But of course, those registers may not be enough to hold all the computations values, the intermediate values that are getting computed in a par-thread. This is, in case a thread is very computation heavy, and it is generating lots of intermediate data, which needs to be stored somewhere, and the register files allocated set of registers for the thread and not enough, then definitely you need to have space in the DRAM, which can be used right, for the storage of data.

So this is known as the local memory per-thread, essentially insight is in every thread, will have access to some segment of the external DRAM, definitely uses the DRAM the access is slow. So

this will happen, the storage or the access to the local memory of the thread will happen when it is doing some local computation and the values cannot be stored in the register there are too many things to store for the registers available to this thread.

Then we have shared memory for low latency access to data shared by threads in the same SM. So what is this shared memory. So as we can see that the register file debuggers divided in a bar warp basis but suppose the thread across the warp needs to collaborate among each other. They want to collaborate on the computation that is going on. For that, you need something which is available here, was the shared memory.

So this is the memory segment, which is transparency visible to all the cores. So, somebody's updating some data in this memory segment is visible to other computing thread in any of the other cores. So, the good thing is the shared memory sitting inside the SM. So, the access of the shared memory is very fast. If I compare it with the access of that DRAM. So, if multiple threads, executing across cores you want to do some collaborative computation.

The good thing for them to do, would have the certain data points defined as shared memory type data and access them. Of course, register the fastest access, but next to be access time for the shared memory. So, the shared memory is useful for low latency access to data shared by threads inside the same SM. And then of course you may need a lot of data for the threads to warp on, which is basically sitting in the global memory.

And it is being brought into the system that is inside the SMs hierarchy of shared memory L1 cache and the registers as and when required. So this global memory for data is shared by all threads of a computing application. And this is again implemented in the external DRAM chip of the GPU. So. So just to summarize, you have the global memory available for a CUDA program, defined as a space in the external DRAM chip of the for par-thread.

If there is something that has to be stored beyond the register, because the registers is already loaded. And then you also have some access to the local memory per-thread, which is basically prior to the thread, and that the physical location for that would also be the DRAM. So, we can

understand what is the difference between DRAM segment which is a global memory and a DRAM segment is a local memory.

The global memory is again shared by all threads so any update to any data point on the global memory is visible to all threads. But the local memory is also something that is implemented in the external DRAM, but it is defined in a per-thread basis. So any update done by a specific thread is going to be used by that thread only it's not going to be visible to the other fits. So, essentially, how this memory model for CUDA programs is organized.

You have a very high level DRAM. There is a physical location where you have a global memory that is defined that is shared by all the threads, and that is to be used for collaborating computation across SMs, because as you can see, the shared memory is also sitting inside SMs, and this way I have 16 more SMs, and all day sales update hulu's finally to the global memory. So, if I have to do some collaborative computation across SMs threads.

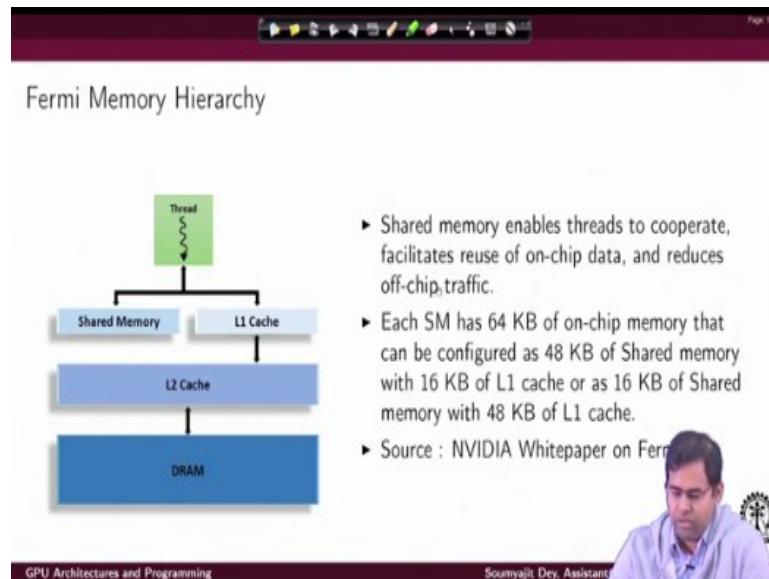
Across SMs in that memory update has to happen to the global memory segment that is defined in the DRAM. Again, I will repeat this part that local memory is also something defining in the DRAM, but is defined in a per-thread basis it is used in a per-thread basis used for computation and holding of temporary values for that specific thread. And this is, this is true for every thread individually.

So for holding the temporary data, apart from the register by segment which is assigned to that thread. They have some segment in the local memory, which is again physically located in the external DRAM. For faster computations, and collaboration among threads, inside in SM, you have this shared memory, which is allowing you low latency access. If I compare the access time with this to the global memory.

And this helps for sharing data by threads inside the same SM. So sharing data by threads across SM has to happen to global memory, sharing data by thread across SM has to happen to shared memory because it provides a low latency access for each thread in the SM the fastest access of data happens to the part of register assigned to it but if it needs more place for holding temporary

data, it has to access some segments. Inside the DRAM which is defined as that thread local memory.

(Refer Slide Time: 21:23)



If we look into the specific memory hierarchy Starting from this Fermi family of GPUs. Then there is something fascinating about the shared memory organization here. So if I look at the computation from the point of view of a thread. If I look at the memory organization from the computation of a view of a thread, then the nearest to me is the register file, not showing the bigger.

The next level is a shared memory or L1 cache then I have L2 cache. And then I have the DRAM. So, the good thing about shared memories as if you can look into the figure, it's inside SM. So, if I have to do the computation inside the SM two different course updating values and exchanging values across the themselves without crossing the boundary of the SM, then there is no point in updating value to the global memory and then going to the other SM. Rather, it can be done to the shared memory right? As we have discussed already.

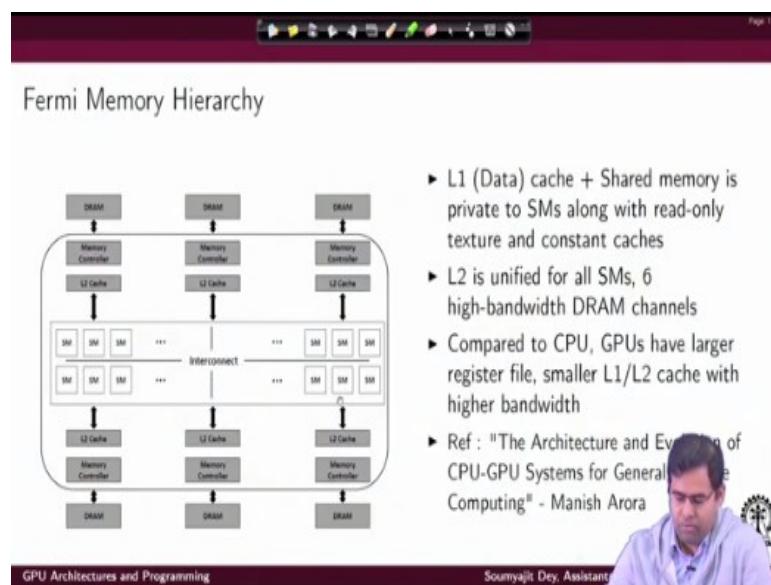
So that is what it enables it enables the threads to cooperate there to facilitate the use of on chip data and reduce of off chip traffic by off chip traffic we mean that access to things that are outside the GPU chip that is the DRAM. So that's outside the processor. Now, each SM will have

64kb of on chip memory. So this memory is configurable, when I mean this on chip memory I mean this shared memory and L1 cache apart so this is 64 kb.

As you can see its written here. Since we wrote this already right is shared memory, oblique L1 cache that isn't is this. It can be configured in two ways it can be configured as 48 kilobyte of shared memory with 16 kilobyte of L1 cache. Or, alternatively, it can also be configured as 16 kilobyte of shared memory with 48 kilobyte of L1 cache. So both things are possible. It depends on what really you want to do.

If you need to have more amount of collaborative execution than you actually across the inside the same across the different CUDA cores inside the SM you may like to have more amount of shared memory.

(Refer Slide Time: 23:52)



- ▶ L1 (Data) cache + Shared memory is private to SMs along with read-only texture and constant caches
- ▶ L2 is unified for all SMs, 6 high-bandwidth DRAM channels
- ▶ Compared to CPU, GPUs have larger register file, smaller L1/L2 cache with higher bandwidth
- ▶ Ref : "The Architecture and Evolution of CPU-GPU Systems for General-Purpose Computing" - Manish Arora

So, taking a more distributed loop into this memory hierarchy. So, earlier whatever we have been discussing was specific to one SM like this one, this is the picture of the architecture block inside the SM. If you have a loop into the memory hierarchy. From the point of view of the entire GPU chip, you have all these SMs arranged here, right? And they are connected to this interconnect network.

Each of the SMs contained inside them that shared memory or the L1 caches. The register files, the CUDA cores, which are functional units everything right? And you have all the systems here. So, this L1 data cache, or shared memory is private to the SMs, along with some other memory segments feature. The readonly texture and constant cache. So, these are specific cache types, which will also be located inside the SM.

So, the reason is, you can always have specific variables which will you, which your threads will be operating on in read only mode right? So you can put them in the constant cache inside the SM for faster access. And then you have the L2 cache for the L1 cache is private to the SMs. The next level of cache is not private to SMs, but L2 cache is unified for all the SMs. So, essentially, you can think that this L2 cache, its a unified thing.

So you have a common L2 cache to which all SMs can also collaborate. The access to the DRAM has to be done through memory controller from L2 cache, because of course if you do not get the memory element from the L1 you will access L2. If you if the L2 cache also give some miss then the memory controller will reference the DRAM. So the L2 is unified across all the SMs, and the access to DRAM is actually bank.

By bank, what we mean is that DRAM is not organized as a big junk of our physical memory, but is divided into multiple vents of memory, and all these banks can be accessed in parallel. So that is why we are seeing that 6 high bandwidth DRAM channels are present. So we saw here 6 possible access places for the DRAM. And we will of course come back to this topic of shared memory, bank conflicts. And how DRAM banks are access and all that later on.

So, if I compare this idea of CPUs and GPU architectures. It can be quickly observed that GPUs have a larger register file, much larger is to file with respect to CPU. The reason is very simple. GPUs have got more number of cores inside them. By core I mean the parallel processing elements, the basic computes the CUDA cores. And they do simple operations, but a lot of them together in parallel.

For sustaining that you need a very large register file to provide them with a request requisite number of operants. However, if I compare the L1, and L2 cache size. They are much smaller. Of course the L1 cache is divided across, physically divided across a SMs while the L2 cache is unified. But they are much smaller. But again, they provide much higher bandwidth. They provide much higher bandwidth if I compare with CPUs

(Refer Slide Time: 27:53)

The instruction set target of the NVIDIA compilers is an abstraction of the hardware instruction set

- ▶ The instruction set target of the NVIDIA compilers is an abstraction of the hardware instruction set
- ▶ PTX (Parallel Thread Execution) provides a instruction set for compilers that remains same for different generations of GPUs
- ▶ PTX code gets translated to target hardware instructions while being loaded to GPU

GPU Architectures and Programming Soumyajit Dey, Assistant Professor

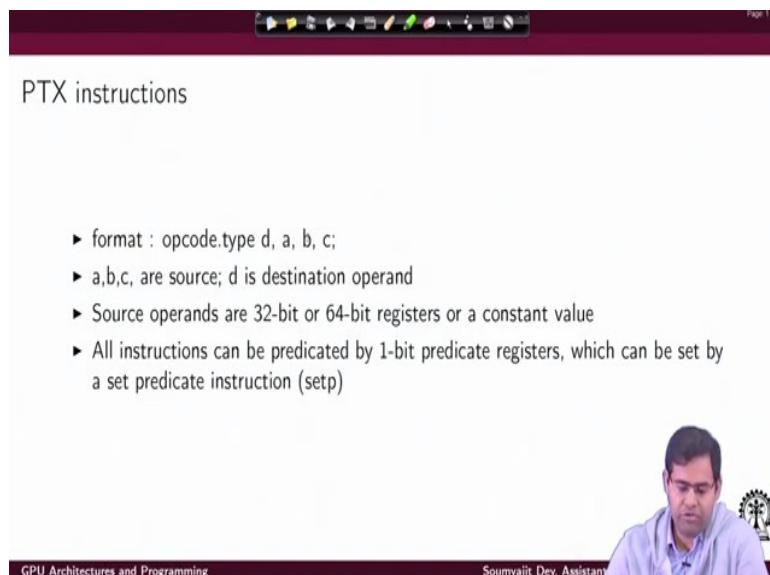
Coming to the instruction set target of the NVIDIA compilers. So, we have already discuss what are the instructions that are supported in general by GPU. But what exact instructions are going to be executed on the GPU. So that is specific set of instructions which are going to be executed in the GPU keeps on changing optimizing, and possibly being refined by suppliers like NVIDIA. So they do a different possible implement a different possible approach in terms of defining the instructions set, If I compare with CPUs.

So what they do is from the programmers point of view, they define an instruction set, which is the target of all NVIDIA compilers. And this is something that doesn't change. So, this is essentially an abstraction of the hardware instruction set, because we change in CPU heavily, the hardware instruction set goes to modification. But the NVIDIA compiler, or if somebody developed some other compiler.

They will, need not always upgrade themselves or conform to the ever changing actual hardware instruction set. But they just need to emit code in a well defined instruction set, known as PTX, PTX full form is Parallel Thread Execution. So, this is the abstract instruction set that is defined for generating code. And if you are trying to write a compiler for the GPU system, you like to generate code, which is in the PTX format will go into details later on today.

I mean, in his lecture, we are just introducing this idea of PTX. But then the question is finally it needs to be translated to hardware instructions. Well, PTX code gets translated to this instruction while its is actually loaded into the GPU. So when the compiler emit the code, you emit PTX code. Compiler optimizations. They are also defined on the PTX code.

(Refer Slide Time: 30:07)



PTX instructions

- ▶ format : opcode.type d, a, b, c;
- ▶ a,b,c, are source; d is destination operand
- ▶ Source operands are 32-bit or 64-bit registers or a constant value
- ▶ All instructions can be predicated by 1-bit predicate registers, which can be set by a set predicate instruction (setp)

GPU Architectures and Programming Soumyajit Dey, Assistant Professor

The PTX code format is something like this. So you have a opcode, then you have a destination operand is followed by three possible source of operands. The source operands can be 32 bit or 64 bit registers are also a constant value. There is something interesting about this instructions, each instruction can be predicted by a one week predicate register. Which can be set by a specific instruction called a predicate instruction.

So this is a facility that helps to decide whether or not to execute an instruction. Based on some specific conditional that will be there in the program. So this is something that handles that plays a big role in handling branches in GPUs. And that is something that is a that is very important

because we can understand you are trying to execute multiple instructions in parallel. So, how really you execute a branch, depends.

Because if you are executing a warp, you have 32 threads progressing together in lockstep by lockstep I mean that you have. I mean, if I go to that older architecture example that we just kept. So there we defined a warp execution inside 4 clock cycles. Each clock cycle. Since because there are eight of the SP cores in that example, but of course you can understand things new things keep on changing the evolution of GPU architectures.

So, from a programmers point of view, it is a good way to think that, warp, all the instructions that are executing in a warp are executing exactly in a lockstep. So, all the threads execute the same instruction together. Next, they execute the next instruction together like that. When these threads face a branch instruction, it may so happen that some of the threads, they are thread ids satisfy the branch construction.

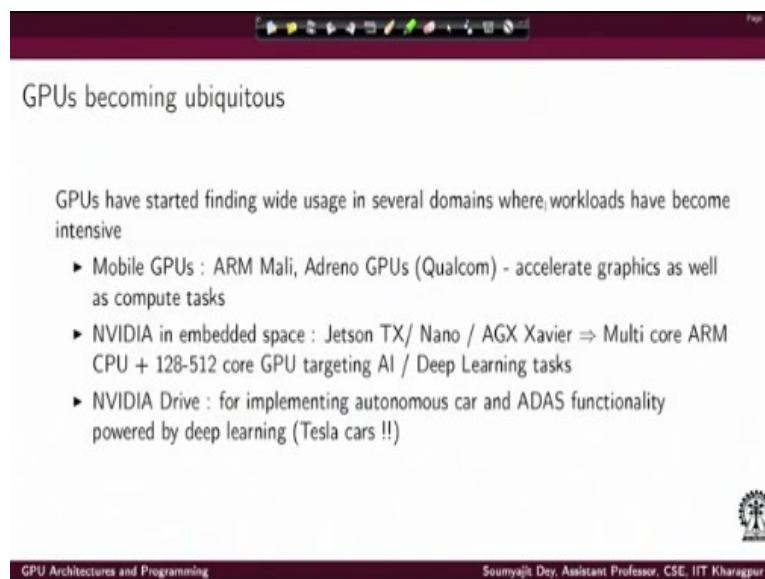
For all the thread ids do not satisfy the branch constitute construction. So when that happens, then the GPU is to handle, which of the strip should make progress and which of the church should not make progress. And this is something that is decided by the set P says predicate instruction we will get into that later. Thank you. So we will end with this for the time being. And in the next lecture, we introduce something more in this regard.

GPU Architectures and Programming
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture no. #8
Intro to GPU Architectures (Contd.)

So, in the last lecture we have been discussing about the different internal architecture sub standard families like Fermi family of GPUs. So, in this lecture, this is going to be the last part.

(Refer Slide Time: 00:37)



GPUs becoming ubiquitous

GPUs have started finding wide usage in several domains where workloads have become intensive

- ▶ Mobile GPUs : ARM Mali, Adreno GPUs (Qualcom) - accelerate graphics as well as compute tasks
- ▶ NVIDIA in embedded space : Jetson TX/ Nano / AGX Xavier ⇒ Multi core ARM CPU + 128-512 core GPU targeting AI / Deep Learning tasks
- ▶ NVIDIA Drive : for implementing autonomous car and ADAS functionality powered by deep learning (Tesla cars !!)

IIT Kharagpur

Of our topics on GPU architecture basics, we just will just introduce you to this idea that whatever we have been discussing. Maybe there are, I mean they are more about GPU architectures, but we also need to discuss a beat on the current reach of GPU architectures. Otherwise one may start thinking that ok GPU is nothing but a card that you attach to a desktop system, or maybe it's also a specific kind of processing hardware which is present in the cloud.

In this high performance systems around us, which are executing all the nice and fine workloads that kind of priority to their lives and all that. However, in contrast, there is a significant amount of GPU computing going on in devices around us. Which is something many of us may not be very much aware of. So is the sequel in the domain of embedded GPUs. So, just to make my point again is not that we have GPUs present in our workstation class systems.

Our laptops for gaming workloads, or in the cloud. They are also present around us in mobile systems in the embedded space. And also in autonomous driving systems which are slowly getting built. So coming to this domain of mobile. So, in most modern smartphones, you have a GPU core, which is responsible for accelerating the graphics, or the rendering pipelines, for your mobile screen.

A very popular choice of GPU core, which is provided by arm is a mali GPU code. It is present in several families of processors, for example the Samsung Exynos chips, which are present in many of the Samsung smartphones. There is also another very popular offering called the Adreno GPUs, which are present in, mostly I mean almost all Qualcomm chips, there's a Snapdragon chips, which also actually drive mobile smartphones.

A significant number of mobile, mobile systems. So, more or less, these are the two popular choices of GPUs. Of course there are many other choices also but these are the significant ones. So they can be used to accelerate the graphics workloads and provide a smooth rendering in the smartphone display. Also they can be leveraged for doing computers, just as they can be different for doing computers in the desktop, or the SPs systems.

So, apart from mobile GPUs. We have significant amount of offerings from Nvidia in the embedded space, for example, is Jetson family of devices. The Jetson T Jetson nano, which is the smaller IOT is the type of system offering, and also the high end AGX Xavier of your kind of system. So essentially, these are all embedded boards, comprising multiple from multiple cores, so they have a processing system they have an SOC.

Which has got multi core arm CPU. + GPU, comprising cores varying numbers, maybe starting from 128 to up to 512 cores inside the GPU. So, and that depends on the kind of embedded board we are talking about if we talk about just a nano is the most lightweight embedded board, which are available from Nvidia for accelerating tasks like deep learning or AI in edge devices, not in cloud systems but in the edge devices.

And the next important class of systems where we have significant drive coming from GPUs is the Nvidia drive systems. So, these are the system architectures which are used for implementing autonomous car, which are used for implementing the ADAS functionalities and autonomous driving systems in cars, as we are, I think everybody here is aware of the company (04:55()) manufactures Tesla cars.

And it happens to be the case, at least that many of them. Many of these companies actually implement. They are autonomous driving computers inside the cars on NVIDIA drive kind of platforms. We also implement significant number of ADAS functionality. So what is ADAS? Its autonomous drive assist systems. So, just to introduce you a bit to this topic. I mean, we always keep on hearing about autonomous driving systems which are present inside cars.

So that a car is going to be driven automatically in like a driverless in a driverless fashion. Where there would be a deep learning computer sitting inside it, which is that computer is this Nvidia drive kind of system is not a normal computer, but it is. It contains a significant amount of engineered resources. Currently containing specific GPUs specialized for this purpose, and also arm cores and other kinds of cores.

So, they will execute our deep learning pipeline, which will try to take all the driving decisions. On the behalf of the driver, when to accelerate ,when to turn right,when to turn left when to change gears and all that. However, while these are still some of them on an experimental stage, but there are significant vehicle functionalities which get automated is not that the vehicle has always got to move in an autonomous mode.

But these are known as drive assist systems that means the vehicle is not fully autonomous. But certain operations of the vehicle can be autonomous automated, for example a park assist system, which can, which can park the vehicle slowly automatically without assistance from the driver. So all those kind of systems the assist systems reform, these kind of ADAS functionalities for a vehicle.

And some of these ADAS functionalities also include are implemented using AI based techniques. And of course, AI based techniques to the NVIDIA drive system for popularly used.

(Refer Slide Time: 07:01)

GPUs as mobile workload accelerators

Figure: Typical architecture of an ARM based Mobile SoC

GPU Architectures and Programming Soumyajit Dey, Assistant

Objective : Maximize performance and reduce power consumption
Developers need to map the workload across the whole CPU + GPU system
RenderScript for Android SDK, OpenCL - language support for data parallel computation on Mobile devices

So, coming into this idea of mobile workloads, like, what is the GPU doing inside mobile? As we discussed that the primary reason can be, for driving the huge amount of graphics computation required for rendering, nice videos or high speed gaming graphics in a smartphones big display. Also, it can be used for compute loads. For example, trying to do some AI based inferencing on a mobile platform to do an inferencing task on a mobile platform.

For example this smart cameras and live recognition systems you are taking a picture in using a mobile camera and trying to also recognize the people in the picture that would that is also an AI task. If It can be executed on the mobile can save a lot of communication with the cloud. So for this also the mobile GPUs can be leveraged. Now, in this slide we present a typical picture of how current generation mobile resources are organized representative picture.

And is trying to show that a mobile SOC can be quite heterogeneous in nature. For example, you can have multiple CPU systems, along which if GPU cores. So here in this picture we should interconnect system, which is connecting a cluster of CPU big cores. So these are typical kinds of course which which are provided by arm, and there can be a cluster of new small cores. And then even have a mali based GPU.

So why are these CPU big cores cluster together because they can, big core I mean, it is clocked at a higher rate, it can have more number of SIMD units to present. By small Core I mean it's a low power core, having low amount of processing bandwidth. They have their internal L2 caches so that again the structure as you can see is similar or the SM designs at a larger scale for Fermi. So L2 is unified.

It is available to all of different CPU bit cores inside them we will have the L1 cores. Again, they'll do is again unified and is available individually for all the CPU cores, is the big cores from a cluster, the small cores form of clusters. And again, here you have the Mali GPU with four GPU cores. Again, they're connected by unified integration, and there is a memory management tool.

So why is a mobile chip going to be managed in this kind of way, where there are good reasons, as we discussed that this bit core means it can do lot of computing compute ,also executing and consuming more power. These can do less amount of compute, because they are clocked at a smaller frequency, but they will also take less power. And for doing graphics worker processing, you can use the mali GPU.

With this fabric of heterogeneous cores available on the mobile platform. It provides significant handle to the developer to decide, and map workloads across this whole CPU,GPU system. So if I have a heavy workload, I would like to map it on this big cluster. If I have a lightweight workload. Or maybe I have a workload which is a bigger of, which is a mixture of heavyweight tasks.

And light weight tasks, I can decide and accordingly intelligently map as a programmer the tasks into the big cluster, or the small cluster, if I have a graphics workload I can intelligently map it to the Mali GPU. So, all these options are available in a modern mobile SOC which is kind of heterogeneous in nature. Now of course, what are the different programming languages available from the SIMD paradigm.

Now we are in this course we are almost speaking about SIMD parallel programming languages which are of the style that is single instruction multiple data. So from that perspective. We are parallel programming support available for mobile devices in terms of languages like open CL, so they provide is data parallel select a language for writing code which is data parallel in nature and the code will execute in a mobile device.

Also, if you are looking into the well known Android SDK Android SDK provides a facility called Render Script, through which you can specify your data parallel program that can be executed inside this cores.

(Refer Slide Time: 11:44)

Integrated GPUs in Desktop Systems

With the release of AMD's Fusion and Intel's Ivy Bridge architecture (i3, i5, i7) in 2011, the trend of fused CPU-GPU architectures started

- ▶ CPU and GPU access the same physical memory such that zero-copy transfers can be employed
- ▶ Zero-copy transfers ensure coherency; translate pointers to memory buffers for the common CPU and GPU address space, but do not actually transfer data.
- ▶ Bad effect - CPU and GPU compete for memory bandwidth of the shared physical memory

GPU Architectures and Programming Soumyajit Dey, Assistant Professor

Now, coming out from the mobile space. Let us also take a look into desktop systems. Now, in desktop systems. Of course you can add an Nvidia card and you have that you have a GPU. But it's not necessary that without that card you do not have a GPU. Now this is something new, right, because we have always thought that is a GPU comes with a separate card. Well in modern CPUs, starting from if I go into AMD is offering starting from AMDs fusion.

And also, if I look into Intel supporting Intel's ivy bridge architecture. So they provide kind of CPU, which also has a fused GPU inside it, so it's like an integrated CPU, GPU system. So, the difference between a large GPU card and this kind of system is that you have a small graphics

processing part, sitting on the same dye, and they are sharing the same physical memory. So that zero copy transfers can be employed.

So, essentially, if you have a CPU chip. And you attach a separate GPU card on your desktop system, then they communicate to the PCI Express, and there is always this communication over it. So until and unless you are going to throw some significant amount of parallel computation for the GPU. Maybe the mapping doesn't make sense because apart from gaining in terms of parallel processing you are also losing in terms of the data transfer over it. But that problem is not present here because CPUs and GPUs access the same physical memory.

So, that there is no physical data transport, or you can have this logical zero copy transfers that can be employed. Zero copy transfers, ensure coherency. And essentially what they do is they transferred the pointers to removing their first for a common address is shared by both the CPU and GPU, but physically there is no transfer of data, because everything is in the same memory space, but again there is a benefit to this.

So since the moment is this the physical memory space is shared between the CPU segment and the GPU. Both of them compete for the memory bandwidth of the same shared memory, and that can create interference issues. So, I mean if both of them are trying to compete for the same segment of the memory, some of them, only one of them will will always fit a stall and due to that overall computation will slow and all that.

So, there is always a good thing and also a bad thing with respect to

(Refer Slide Time: 14:25)

Page 11

Integrated GPUs in Desktop Systems

Figure: Fused CPU-GPU with shared LLC

- ▶ In more recent architectures, Intel Broadwell and beyond, CPU and GPU were further integrated
- ▶ They access the shared last level cache (LLC)
- ▶ This helps in CPU and GPU executing computational kernels on the same data in parallel collaboratively (LLC enables cache coherence between CPU and GPU)
- ▶ "Co-Scheduling on Fused CPU-GPU Architectures with Shared Last Level Caches" - Henkel et. al.

Soumyajit Dey, Assistant Professor

This kind of an arrangement. Now, something more we have is, with respect with integrated GPUs So, in recent times, from Intel broadwell architecture and beyond CPUs and GPUs are further integrated. So earlier there was the CPU chip and CPU and GPU together sitting on a same chip, and they were faster they were they were actually looking into the same physical main memory.

So, but, from this broadwell time. What happened is they added a 3rd level of cache, which is the shared last level cache or LLC. Earlier it was not there earlier you have the GPU with GPU cache, CPU, you have multi well cores of the CPU, each of the cores will have the L1s for instruction and data the L2s. And that is all right. So you have these multiple cores of GPUs and CPUs, and they will be accessing the memory.

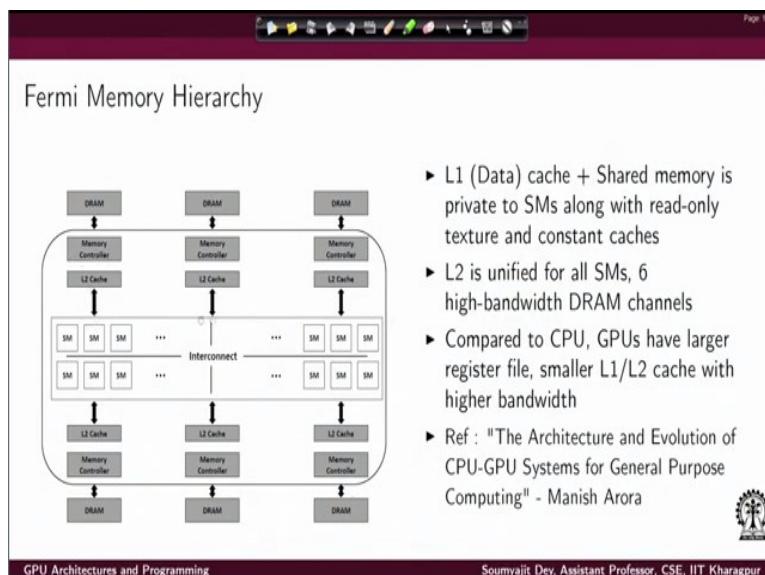
But we found the broadwell architecture, what happened is the people added this last level cache inside the system. So what is the good thing about the last level cache, it provides you additional on chip memory. So this helps the CPU and GPU in executing compute kernels on the same data in parallel, collaborative. So, earlier it was not possible, but since I have this last level cache on chip.

They can execute on the kernels, compute kernels both CPU and GPU. And, I mean, I will be accessing the same data and coherence of the data will be maintained through that last level

cache. Now, coming to this issue of coherence. So what was the cash really going to do this going to take care by hardware of the coherence decisions. But again, is also will introduce some problems with respect to this we will discuss.

But just to understand the point that, to which the systems have evolved earlier, there was only CPUs and GPUs which are integrated on the chip but with the addition of more cache the hierarchy. Since the cash is also uniformly available to the motor CPUs and GPUs. They do not need to synchronize using the main memory. They can synchronize their computations using the shared last level cache itself so that is the takeaway from this. Coming to this issue of coherence and all that,

(Refer Slide Time: 16:59)



Let us jump back, jump back to one of our earlier slides to understand it better. So, if you remember we were discussing. Let us take this picture that you have in a standard so I'm going back to our original discussion on Fermi GPUs where we have discussed that the L2 unified, and the L1s are distributed across the SMs. Now observe one thing since L1s are distributed across SMs.

That means each thread. Can, can load a copy of data into, I mean the threads are sitting in different SMs, so they can load copies of data and update those data. And those updates will be visible to the ones, so it can always happen that across SMs, L1s have different updated copies of

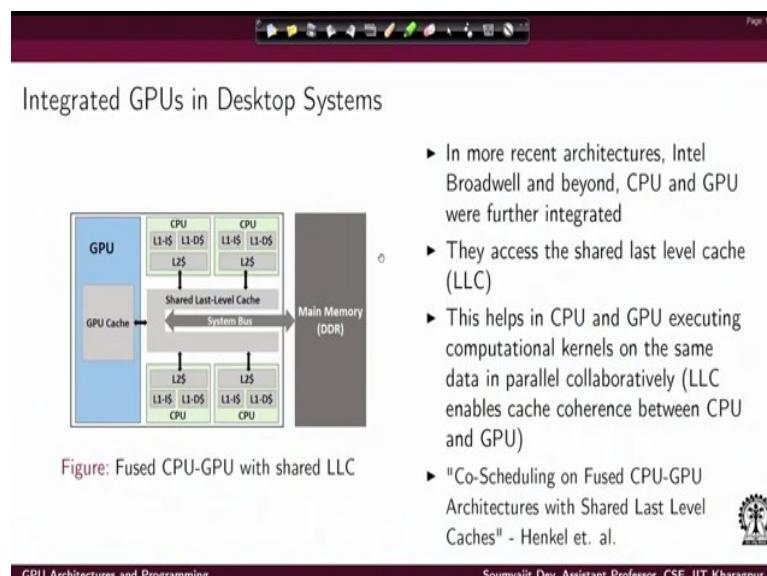
variables. So, that since the L1s are not unified here. I can technically have the L1s which are not synchronized.

They are, they are containing different possible updated values or variables, and then may lead to consistency issues with respect to data values. This is something we are not going to detail, we are just trying to give you an idea that what can be a bad thing that may have been due to the memory being segmented right. Now this is something that a programmer also needs to keep in mind, and accordingly write the code.

When is trying to do a specific kind of computation such that threads across SMs are trying to work on similar data segments. We're just trying to highlight that there can be problems with respect to this which a programmer has to manage. These are the things which we will explore in more detail later. So coming back to our current discussion. So as we were discussing here that since we have the shared last level cache.

The good thing is the CPUs and GPUs need not collaborate on data updates. By the by, by synchronizing through them in the morning, but they can do that using this unified cache itself.

(Refer Slide Time: 19:03)



So coming to other kinds of offerings from Nvidia that is the Jetson series, so if you remember we were discussing the embedded space embedded Nvidia has offerings from Jetson for different

kinds of offerings a lightweight nano nano systems also the heavy weight AGX Xavier kind of systems. So what are these. So we take specific example of a TK1 SOC from the Jetson series. It incorporates a quad core arm machine.

So it incorporates a quad core arm machine so you have more CPU cores. Each clocked at 2.2 gigahertz, and they are 32 bit systems, and it also has an integrated Kepler family of GPU, the CPUs again share a 2-MB L2 cache, so the L2 is again unified for the CPUs, and the GPU has a 192 core system, and 128 kilobytes L2 cache, so that L2 is also unified for the cores for the GPU.

Now, what's not shown in this picture is that apart from having this cores, if you bit cores, the CPU sub part also has little arm course, of course they are there for low power, low I mean low power compute, which is not frequent too much of performance but then I can always optimize the power and then I will map such workloads to the little, arm cores. Again for you, increasing the bandwidth of memory access that DRAM is banked into 32 banks.

Here banks is of 64 MB. So this is a typical architecture of a Jetson kind of system from Nvidia.

(Refer Slide Time: 20:42)

NVIDIA Drive series of systems

The NVIDIA Drive PX 2 is based on 1/2 Tegra SoCs where each SoC contains 2 Denver cores, 4 ARM A57 cores and a GPU from the Pascal generation

Useful for implementing high throughput real time neural net processing - self driving / drive assist systems

Figure: Source- Wiki, NVIDIA Drive PX Platform

GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, okay, what are the kinds of computers, there are people are offering for cars. So this is one such example, which is basically the Nvidia drive series of system. So the Nvidia drive PX 2 is

one of the standard systems that is based on these Tegra Associates from Nvidia Tegra associates another, more powerful associate containing 4 arm a A57 cores And some cores called the Denver cores most that is something that Nvidia separately created.

I mean, for this kind of system, and they also include a GPU from this latest pascal generation. Now, what is the use of such a high performance system in a car like as we discussed that a car in future will have lot of drivers is, as well as still driving facilities, which are, which are essentially deep learning computers. Which also need to take deliver lot of, regular decisions in high throughput inside real time. And those can be implemented in this kind of system targets.

And that is one of the reason for building these kind of drive systems which are much more powerful if we compared them to those in the embedded offer. So just to summarize these are the different kinds of other systems, other computer architectures which are available to us, which provide several forms of data processing capabilities. You have GPUs available in mobiles. You have GPUs available in the embedded space in form of this Jetsons series computers.

You also have heavyweight compute systems from involving GPUs build for the Nvidia drive systems which are part of many well known electric cars, and they provide significant drivers is functionality and also. So with this will end our coverage of the basics of GPU architectures and will shift to the introduction to CUDA programming. And will introduce CUDA as a language and how to write specific kind of CUDA codes.

The kind of programs for the GPU, and leveraging this parallel architectures. That is something we will start covering from the next lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 02
Lecture No # 09
Intro to CUDA programming

Hi let us get started with the third topic in this course on GPU architectures and programming. Now we are to introduce the notions of CUDA programming language, one of the most popularly used GPU programming languages.

(Refer Slide Time: 00:48)

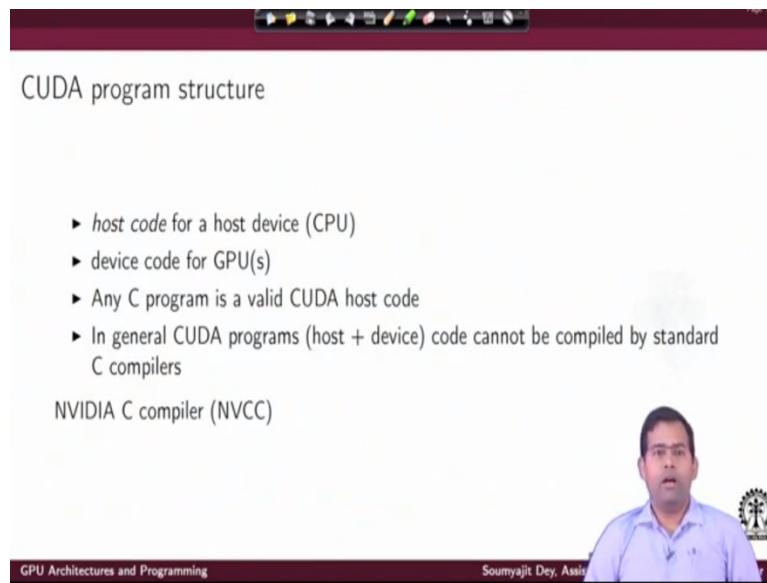
Compute Unified Device Architecture

- ▶ CUDA C is an extension of C programming language with special constructs for supporting parallel computing
- ▶ CUDA programmer perspective - CPU is a *host* : dispatches parallel jobs to GPU devices

So, what is CUDA? The full form, first of all, is compute unified device architecture. So CUDA is essentially an extension of C programming language with special construct that support parallel programming. Now, this is a programming language extension developed by NVIDIA primarily for their GPU's, as we know that NVIDIA GPU's are very popularly used as accelerating a device in conjunction with - I mean high performance CPU's.

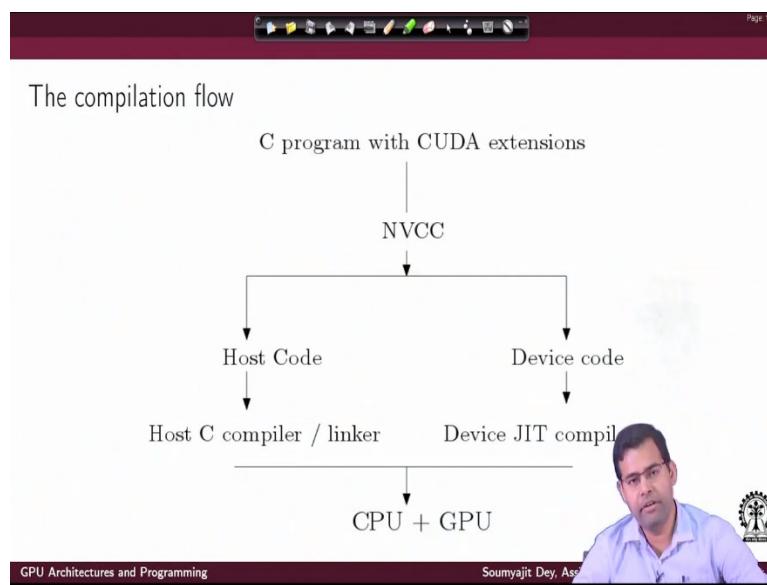
So for the CUDA programmer, the perspective is that the CPU is the host that means in CPU you have an orchestra program which is running and it is dispatching parallel jobs to GPU devices. Well, there may be more than one GPU device attached with the CPU. So generally, it can dispatch multiple parallel jobs to these GPU devices. This job will execute in the devices and get back the results to the host.

(Refer Slide Time 01:57)



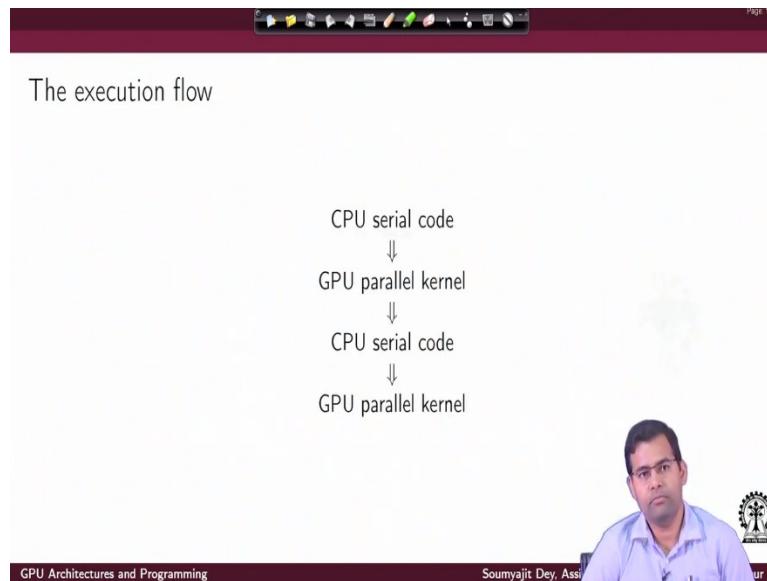
So, the way a basic CUDA program is structured is as follows. There is a host code which is resident, that means it is executing on a host device that is the CPU. And there is a part that we called as device code which is supposed to execute on the GPU's. Technically speaking any C program is a valid CUDA host code. Essentially, it is a code that it can execute on CPU. So in general CUDA programs constitute of host plus device code and they cannot be complied by any standard C compiler, and for this purpose we require this specific compiler from NVIDIA that is the NVCC compiler. That is the NVIDIA C compiler.

(Refer Slide Time 02:43)



The compilation flow for NVIDIA C compiler is as follows. You write the CUDA program essentially C program with CUDA extension, you compile it with NVCC; what you get is the host code and the device code that is the part of the code which will be compiled further by the host C compiler and linker. And that device code will be JIT compile for execution on the device. So overall these are the two different segments and code that you get one to execute on the CPU and the other to execute on the GPU devices.

(Refer slide Time 03:19)



The execution flow is as follows. The host code is the code that is executing in the CPU serial. The host code will launch the device code that is the GPU parallel kernel. The GPU parallel kernel will execute in the GPU, this is what we referring to the device code. It will return results to the host code. With those results the host code may again execute, do some functionality and then again launch some parallel kernels for the GPU devices. And this computation can go on back and forth.

I can have a simple CUDA program which will launch one kernel per GPU device, get back the result and print me the result; Or I can a sufficiently complex program which will launch some kernel get some results, do some processing in the CPU and then again launch another kernel and so on so forth.

(Refer Slide Time 04:16)

Examples : Vector addition CPU only

```
void vecAdd(float* h_A, float* h_B,
           float* h_C, int n)
{
    for (i = 0; i < n; i++)
        h_C[i] = h_A[i] + h_B[i];
}
int main()
{
    float *h_A,*h_B,*h_C;
    int n;
    h_A=(float*)malloc(n*sizeof(float))
    h_B=(float*)malloc(n*sizeof(float))
    h_C=(float*)malloc(n*sizeof(float))
    vecAdd(h_A, h_B, h_C, N);
}
```

GPU Architectures and Programming Soumyajit Dey, Assistant Professor, IIT Kharagpur

So before discussing our first CUDA program, let us start with an example vector addition code that is executing in a CPU - the very simple C program. So you have a main in which you have this float types define. So this are the pointers to floating point arrays which are dynamically allocate with the malloc calls. Once this dynamically allocation is done, you call this vector add function. Inside this vector add function, off course I am just trying to show you an example, we have not written any code for initialization of arrays and all that assume those are there.

So once that is done, the vecAdd will be called and after vecAdd is called inside a loop I am just trying to add the 2 vectors and return the result in the h_C array. So in that way, this vecAdd with three arguments - the first two are kind of the input argument and h_C is a dynamically allocated array which we contain the output argument. Off course, the initialization code is missing here as discussed earlier. So this is how the typical vector addition will execute in a CPU.

(Refer Slide Time 05:48)

```

#include <cuda.h>
#include <cuda_runtime.h>
__global__ void vectorAdd(float*, float*, float*, int);
/*-----*/
__global__
void vectorAdd(float* A, float* B,
float* C, int n){ //CUDA kernel definition
int i=threadIdx.x+blockDim.x*blockIdx.x;
if(i<n)
    C[i] = A[i] + B[i];
}
/*-----*/
void vecAdd(float* h_A, float*h_B,
float* h_C, int n)
{//host program
    int size = n* sizeof(float);
    float *d_A=NULL, *d_B=NULL, *d_C=NULL;

    // Error code to check return values for CUDA calls
    cudaError_t err = cudaSuccess;
}

```

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor

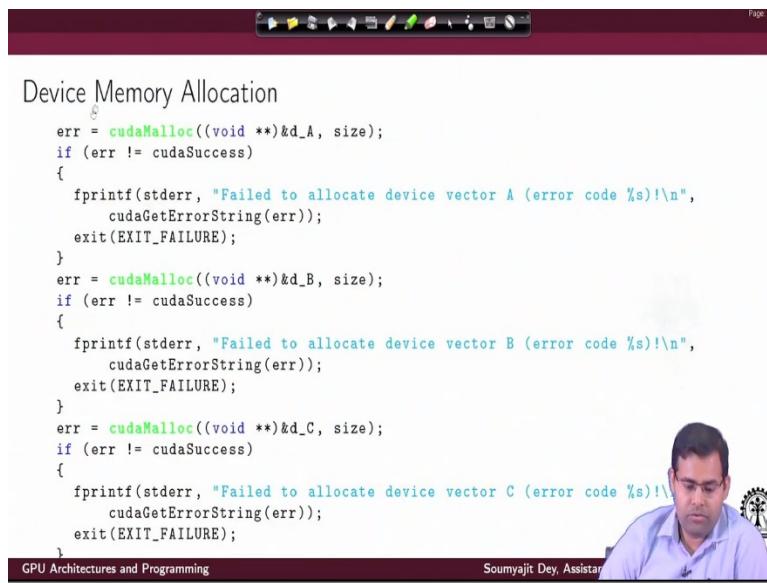
How it will work on CPU-GPU system? So, here comes the first CUDA program. First of all, you observe that we have hash include <cuda.h> and also we have a hash include <cuda_runtime.h>. This are the header files which contain the required CUDA functionalities that will be using in our code. In this first CUDA program, as we have discussed earlier, there will be a piece of code that will execute in the CPU, it will launch the parallel code or the device code for the GPU. This is what we called a kernel.

Now, what is the code that will execute in the CPU? That is essentially this vecAdd program. This is the program that is commented as a host program, as you can see this is similar to our earlier CPU only vecAdd program. So essentially this is a program that is called from main, it is expecting to be passed three pointers. So this pointers that containing base addresses for three arrays, which has been dynamically allocated before the call has been made and also suitably initialize the input arrays. They are added as vector and output array is return. That was my original vecAdd code. Here, I can have the VecAddcode which is the CPU side host program. Similar input arguments are there. Now, what we are trying to do here is that we do not want the addition to be done in this CPU. This host code is supposed to launch something called a GPU kernel which is the code that will be executed in the GPU. So, let see how it is done. We will just watch through this program step by step.

First thing, inside this host program vecAdd, we declare some pointers, and then we also declare a specific enumerator data type cudaError_t and we initialize it with one of the types that is

cudaSuccess. So this is the error code to check return values from CUDA calls. Now, these are defined in the header file that we have defined earlier. These are not our own design they are already defined.

(Refer slide Time 08:25)



```
Device Memory Allocation
{
    err = cudaMalloc((void **)&d_A, size);
    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
    err = cudaMalloc((void **)&d_B, size);
    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
    err = cudaMalloc((void **)&d_C, size);
    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n",
                cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
}
```

GPU Architectures and Programming Soumyajit Dey, Assistant Professor

Now coming to the code, the first we are trying to do is just like we do malloc for dynamically allocating memory in a CPU, we fire a function cudaMalloc. Now again, these are all defined in the CUDA headers and what cudaMalloc will do is that it will allocate memory not on the CPU side, but rather the memory resident on the GPU device. It will allocate a specific amount of the memory of size equal to this size and return a pointer for that.

This is the pointer d_A. So as you can see, we have already declared this d_A and now after this cudaMalloc call d_A address is containing a generic pointer and this pointing to size amount of memory resident on the GPU device. In case this malloc call is not going perfectly then this next if condition will not be satisfied. So error will not be equal to cudaSuccess. Again, cudaSuccess is another enumerator data type which should match provided the malloc call goes perfectly fine.

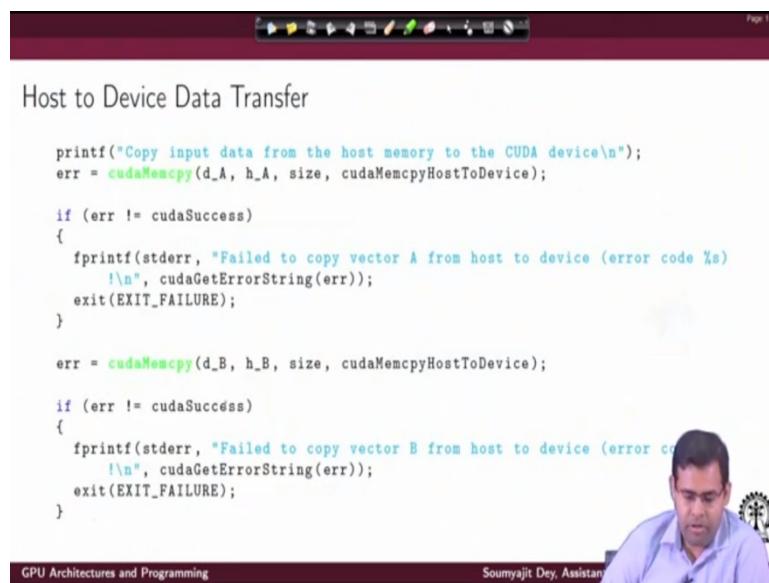
If it is not so, then this condition will fire, and we will have the fprintf a standard error where this message will be printed with the suitable error code. Now, how is the error code coming in? The error code can be figured out by this directive cudaGetString. So cudaGetString will take as a argument with the error of type cudaError_t and from that error code it will figure out the suitable error string which will be printed here using the fprintf command.

Again I am assuming that you are very much conversant with fprintf for printing strings to standard error and everything. So please get acquainted with it if you have kind of forgot and all that. Hence forth, exit will happen with this suitable code. So following this schema, we will do memory allocation for the other pointers that is d_B as well as d_C. So as we can understand, there is a difference between the original pointers that have been passed to the vecAdd host program i.e. h_A, h_B and h_C.

They point to memory location on the CPU memory; they point to resident arrays on the memory location which have been dynamically allocated and initialize before the call has been made for vecAdd. Internally, vecAdd defines more pointers and make the cudaMalloc call to assign these generic pointers so that they can point to memory locations not on the CPU side but on the GPU side.

We will soon see why this is necessary. As you can see, we have a kind of repetitive code of CUDA Mallocing for d_A, d_B and d_C and in case of error, suitable error commands will trigger as has been written down here.

(Refer Slide Time 12:02)



The screenshot shows a presentation slide with a dark header bar containing icons for navigation and search. The main content area has a white background. At the top left, the title "Host to Device Data Transfer" is displayed. Below the title is a block of C code. On the right side of the slide, there is a small video window showing a man with dark hair and a light blue shirt, identified as Soumyajit Dey, Assistant Professor. The video window has a decorative circular logo in the bottom right corner. At the bottom of the slide, there is a dark footer bar with the text "GPU Architectures and Programming" on the left and "Soumyajit Dey, Assistant Professor" on the right.

```
printf("Copy input data from the host memory to the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector A from host to device (error code %s
    !\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector B from host to device (error code %
    !\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

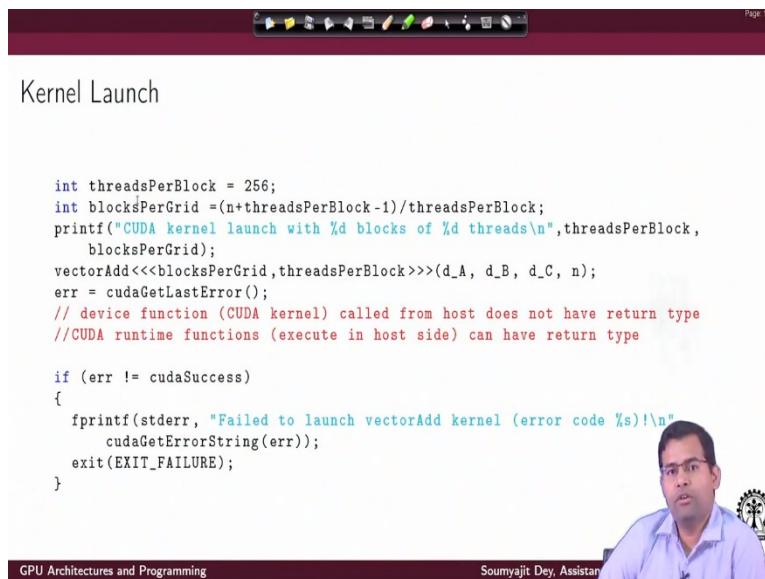
Following this, we have print statement which is saying that copy input data from the host memory to the CUDA device. So if this is print that means all the previous malloc operation went fine, we have 3 locations in a memory of size equal to size allocated on the GPU memory

side and they are being pointed to by `d_A`, `d_B` and `d_C`. So the next thing that I need to do is, I need to copy the input arrays from the host side memory to the device side memory. From the CPU memory to GPU memory. Because essentially, I want to do the vector addition on the GPU, in parallel, not on the CPU. That is the basic idea of this program which is the vector addition on a CPU-GPU system. So now for doing this transfer of data from the host side to the device side, we have this command `cudaMemcpy`. So what it does! So this is the device side memory `d_A`, this is the host side memory `h_A`.

So there is a copy from `h_A` to `d_A`, the copies size is dictated by this parameter “size” and the type of copy is “`cudaMemcpyHostToDevice`”. Following this directive, all the data resident in `h_A` up to “size” amount of space gets severely copied to the location pointed to by a `d_A` in the GPU side memory. Now in case this is not successful, suitable error comments will get printed as has been already discussed. Similarly we also do the “`cudaMemcpy`” for the array `h_B` to the device array `d_B`.

As you can see that the codes are again kinds of similar. Once both these operations get done, we have the 2 input arrays containing the 2 input vectors ready for addition in the GPU side memory.

(Refer Slide Time 14:26)



```

int threadsPerBlock = 256;
int blocksPerGrid = (n+threadsPerBlock-1)/threadsPerBlock;
printf("CUDA kernel launch with %d blocks of %d threads\n", threadsPerBlock,
       blocksPerGrid);
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, n);
err = cudaGetLastError();
// device function (CUDA kernel) called from host does not have return type
// CUDA runtime functions (execute in host side) can have return type

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n"
            cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

```

At this point, we are thinking of launching a program on the GPU which will access this array location in the GPU side memory, do the addition and stored the result and transfer the result

back to the CPU side memory. For doing that, we declare certain suitable parameters known as “threadsPerBlock” and “blocksPerGrid” and we pass this parameter to function called vectorAdd.

Now as you can see, the call of this function is very different from our standard C function because apart from its argument `d_A`, `d_B`, `d_C` and `n`, there is something here called `blocksPerGrid` and `threadsPerBlock`. Now what this is we will discuss a bit later on, for that time being just think that this is specific type of function with some extra parameters. Essentially, this is how we are launching program for the GPU.

We are going to launch a function in the GPU which will do the computation of vector addition in parallel. This kind of function which does computation on the GPU side is traditionally known as GPU kernel. GPU kernel launches follow this kind of parameter reiterations. The parameter definition is something we will speak a bit later on. Now just observe that the function has been, apart from this `threadsPerBlock` and `blocksPerGrid` parameters, passed the GPU device memory location `d_A`, `d_B` and `d_C`.

`d_A` and `d_B` are essentially pointing to the vectors which are to be added and the result is supposed to be stored in `d_C`, all of them are in device memory. Now let us look at the implementation of this kernel which is here. So this is how a CUDA kernel can be defined. So first we have the declaration of the `vectorAdd` and then we have actual definition of the `vectorAdd`.

Inside `vectorAdd`, we have a computation of an index that decides what is the part thread behavior and then we have the familiar code of the locations `C` getting the value for $A[i] + B[i]$. So `C[i]` is being written with $A[i] + B[i]$. Now the difference with the CPU side is here, if the GPU has got `n` number of codes; inside it `n` number of scalar processor that many of number of threads will be launched which will do all this addition in parallel. And in that way all this vector addition will happen in parallel. How exactly is something which we will discuss.

So coming back here, this was the call for `vectorAdd` at this point and with this `vectorAdd` call we land up here, this addition is done. Now going back here the `vectorAdd` call, we expect that `d_C`, this device memory location, contains the added vector value. Now some thing important to

be noted here is, this is a function which is executing on the GPU side, for such a function it can take as operands value from memory i.e the GPU memory and write back values again on the GPU memory. So they do not return anything.

So that is what we are saying the device function CUDA kernel call from the host side does not have a return type here. Now this is unlike CUDA runtime directives like cudaMemcpy or cudaMalloc which can return an error code. This function cannot have a return type. But in case this function runs into some issues while executing, it will create a signature which can be caught by cudaGetLastError, it is again a run time function.

It is a function which is a feature of the CUDA runtime system. The point I am trying to make here is the vector add does not itself directly provide the return but rather in case there was some issue in executing the function. This runtime function cudaGetLastError can provide a suitable error and if that is not a cudaSuccess, again we have a way to know that the launch of this vectorAdd kernel got into some issues.

(Refer Slide Time 19:34)

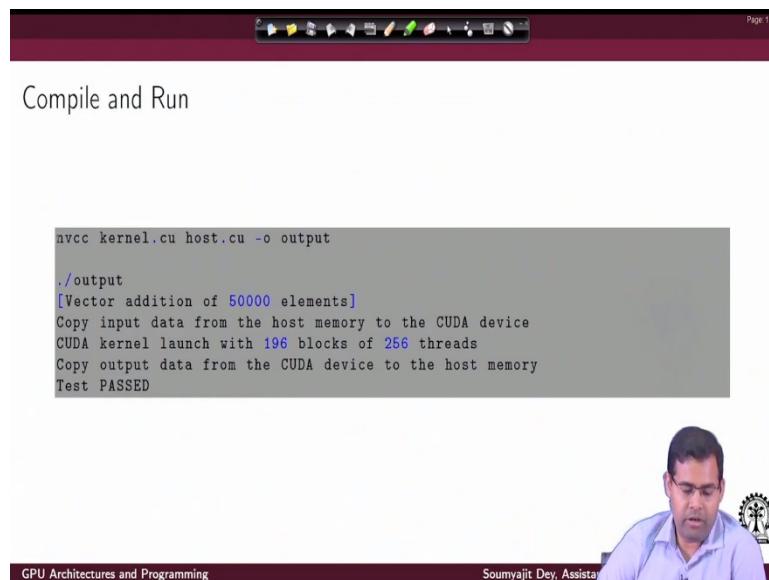
The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Device to Host Memory Transfer". Below the title is a block of C code. On the right side of the slide, there is a video feed of a person speaking. At the bottom of the slide, there is a footer bar with the text "GPU Architectures and Programming" and "Soumyajit Dey, Assistant Professor".

```
printf("Copy output data from the output device to the host memory\n");
err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector C from device to host (error code %s
    !)\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
// Verify that the result vector is correct
for (int i = 0; i < n; ++i)
{
    if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
    {
        fprintf(stderr, "Result verification failed at element %d!\n",
        exit(EXIT_FAILURE);
    }
}
printf("Test PASSED");
} // End of Function
```

Now, we have the vector addition done and the result is there in the device memory d_C which can be copied back again by cudaMemcpy directive to the CPU side or host side memory h_C. Now observe that earlier when we copied the values from host side to device side the directive inside cudaMemcpy was cudaMemcpyHostToDevice. But now when we copy from device side to host side it is cudaMemcpyDeviceToHost. So that is the slight alteration.

So with this directive we have the result back in the CPU. So once I have the results back in the CPU, I can deallocate all the allocation from the device side memory that is we can free of this `d_A`, `d_B` and `d_C` using the `cudaFree` directive. So essentially it is very much C like with some CUDA annotation. And then we have just written some small checking code which is just checking whether it is a absolute value of the sum of `A + B` subtracted from `C` within the error bound or not, otherwise we will say that there is some issue. And and then we do the test pass print.

(Refer Slide Time 20:57)



The screenshot shows a terminal window titled "Compile and Run". The terminal displays the following command and its output:

```
nvcc kernel.cu host.cu -o output
./output
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
```

At the bottom of the slide, there is a small video thumbnail of a man speaking, with the text "GPU Architectures and Programming" and "Soumyajit Dey, Assistant Professor".

So, this code has to be compiled. We run the NVCC compiler with the kernel definition in `kernel.cu`, the host side code `host.cu` to compile them to create the binary output. If you run it, you get this sequence of print statements firing and this should be the output we would expect.

(Refer Slide Time 21:25)

Page 17

Observations

Figure: CPU/GPU Mem Layout

- ▶ `cuda.h` → includes during compilation CUDA API functions and CUDA system variables
- ▶ `h_A, h_B, h_C` → arrays mapped to main memory locations

 Soumyajit Dey, Assistant Professor

Now coming back to how really the execution is going on. So as we know that the GPU is a separate card, it's an accelerator card which is kind of getting attached to the CPU. The GPU has got its own device memory. For doing the computation in the GPU from the last example we could figure out that suitable data points need to be transferred by the CPU to the GPU device memory.

Then the GPU kernel is suitably launched by the first program. The GPU kernel executes on the GPU with input parameters taken from the device memory of the GPU. It writes data back on the device memory of the GPU which has to be again copied back to the CPU. So this is the overall execution flow in a bit more detail. This header file `cuda.h` includes the compilation CUDA API functions and the different CUDA system variables.

We have been exhibiting with some examples inside the code and we also seen that there were the host code using variables that were mapped in the main memory of the CPU whereas before executing the device code we needed to initialize pointers and allocate them suitable memory in the GPU DRAM or the device memory.

(Refer slide Time 22:59)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Observations'. Below the title is a block of CUDA C code:

```
cudaMalloc((void **) &d_A, size);
//allocate memory segment from GPU global memory
//expects a generic pointer (void **)
//the low level function is common for all object types
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
//transfer data from CPU to GPU memory
//d_A cannot be dereferenced in host code
```

At the bottom of the slide, there is a footer bar with the text 'GPU Architectures and Programming' and 'Soumyajit Dey, Assistant Professor'. To the right of the footer, there is a video player interface showing a man in a blue shirt speaking, with a progress bar and other video controls.

A few more observations; we have been using functions supported by the CUDA runtime. So there is cudaMalloc now what was exactly doing? it was allocating memories, it was allocating memories segment from the GPU global memory which is physically different from the CPU's global memory. Now this expects a generic pointer whose type is void **. It is a generic pointer, it can point to any kind of data here.

Now this low level function is common for all object types, that is the reason for why it is a generic pointer. Now the other things is the cudaMemcpy comment as we have discussed earlier, it will transfer data back and forth between the CPU and GPU. If it is having the directive cudaMemcpyHostToDevice, it is copy from CPU to the GPU; if it is cudaMemcpyDeviceToHost, then it is copy from the GPU memory to CPU memory.

The important thing is this device memory pointers d_A. Once the cudaMalloc is done with d_A, this device memory pointer cannot be dereferenced in the host code. Simply for that reason they are pointing to a different physical location i.e the GPU memory. So they can be dereferenced only inside the kernel code.

(Refer Slide Time 24:18)

```
//d_A cannot be dereferenced in host code
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
//transfer data from GPU to CPU memory
//can also transfer among different device mem locations
//can also transfer data host to host- we do not need that
//cannot transfer data among different GPU devices
cudaFree(d_A);
//free GPU global memory
```

So with respective cudaMemcpy, we can have this kind of cudaMemcpy from device to host, host to device and just like the transfer of data being supported among the GPU and CPU's with cudaMemcpy.. We can also do a transfer among different device memory locations. So even if I have two different device memory locations on the same device, I can do a cudaMemcpyDeviceToDevice using the directive.

Also we can do a transfer data from host to host but we do not need to do that, because since I had a normal CPU so that is normal movement of data inside two different location in the array. But I cannot transfer data among different GPU devices using a cudaMemcpy directive. It is very easy to understand, because what are the things that we can really do just to summarize: I can copy data from host side memory to device side memory or from device side memory to host side memory. I can copy data between two different locations in the same device memory. I can copy data between two different memory locations in the same host side memory, although that is not required because it is inside the CPU's DRAM. But what I cannot do is, I cannot have cudaMemcpy copying data from one GPU devices memory to another GPU devices memory.

Because in that case, they are two different physical devices and cudaMemcpy supports transfer between one host and one GPU. So in that case, I have to copy data from device one to the host and then I have to copy from the host to the device 2.

(Refer Slide Time 26:13)

A CUDA kernel when invoked launches multiple threads arranged in a 2 level hierarchy, check the device fn call.

```
vectorAdd<<<ceil(n/256),256>>>
(d_A, d_B, d_C, n)
```

- ▶ The call specifies a **grid** of threads to be launched
- ▶ the grid is arranged in a hierarchical manner
- ▶ (no. of blocks, no. of thread per block)
- ▶ all blocks contain same no. of threads (max 1024)
- ▶ blocks can be numbered as (..., ...) triplets : more on this later

GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now coming back to the call of a CUDA kernel. How a CUDA kernel is called? When a CUDA kernel is invoked, it launches multiple threads in a two level hierarchy. The threads are the basic computing units which engage the scalar processor in the GPU. Each scalar processor will execute one thread; all the scalar processor executes threads in parallel. Now going back to our example CUDA kernel as we have discussed earlier that when this vector add kernel was launched there were 2 parameters blocksPerGrid and threadsPerBlock.

Now let us try to understand the significance of this parameters. So as we have already discussed that we are trying to do computation using multiple computing threads in a GPU, that is the fundamental thing. So when I am doing vector addition, I want to launch a lot of compute threads. All of them will add components of the vector in parallel. Now this arrangement of multiple threads follows a two level hierarchy.

So suppose I am trying to launch n number of thread. So what I can do is, I can define their arrangement in 2 levels at the higher level I say that, ok I have $n / 256$ scaling number of element and then at the lower level is say that each element comprises 256 threads. So in total I have n number of threads being launched. So this call specifies the grid of thread to be launched. This is the technical terms people use that you launch a grid of threads.

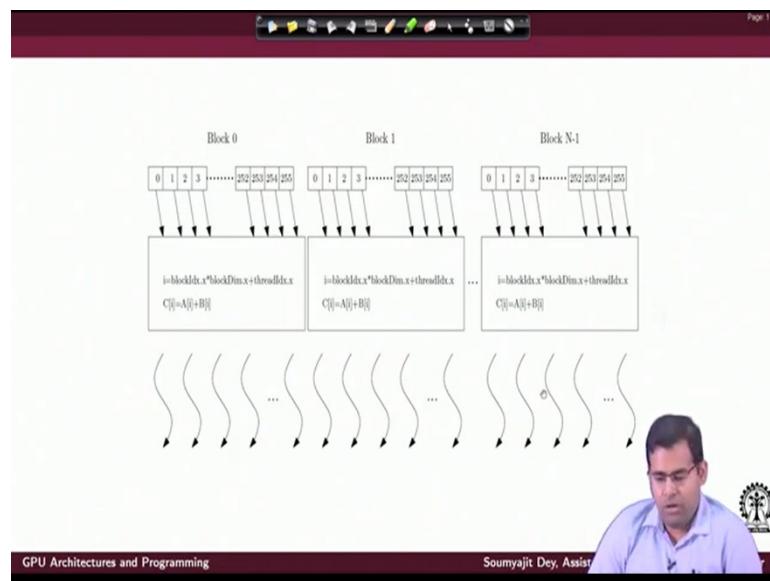
The grid is arranged in a hierarchical manner, you have the number of blocks. So the first component here in the hierarchy is the number of block. So you have $n/256$ number of blocks.

And then you say that each block contains 256 number of threads that is the second parameter. So overall, I have a number of blocks and number of threads per block there is the specification which tells me that how many threads are launched.

Now if we go back to the definition of the vectorAdd kernel. We had threadsPerBlock defined as 256 and then we define the number of blocksPerGrid or just the number of blocks which was just $(n + \text{threads Per Block} - 1) / \text{threads Per Block}$. So essentially, we are looking at total n number of threads here. So you are if this print statement fire it will tell me how many threads are there per block and how many blocks has been launched, essentially here we have the vectorAdd being launched with the number of blocks in the grid, number of threads inside each block.

So off course, since I have a definition of threadsPerBlock that also means that all blocks contains same number of threads maximum 1024. Now I can even have a hierarchical specification of blocks that means I can number blocks in 3 dimensions using triplets. We will discuss this later on.

(Refer Slide Time 29:33)

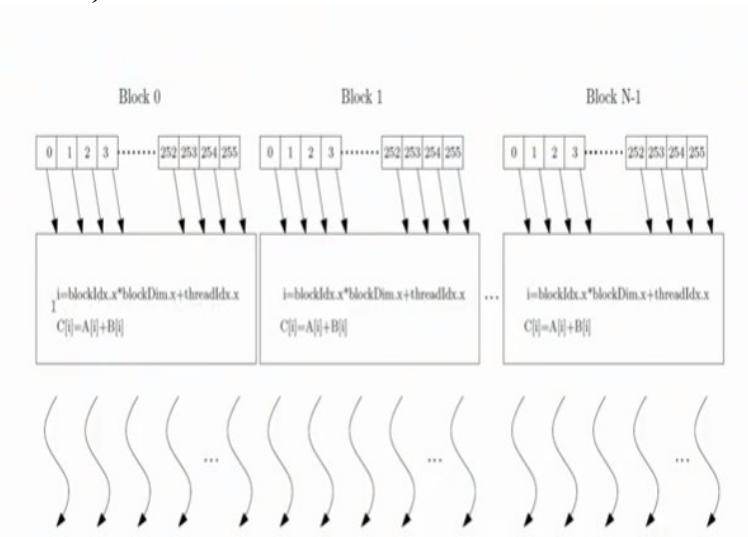


So how are really blocks are arranged so when a CUDA kernel is launched I have this hierarchical arrangement of threads and suppose I have n number of blocks and each block is containing this 256 sets.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 05
Lecture No # 10
Intro to CUDA programming (Contd.)

(Refer Slide Time: 00:28)



Hi, so as we have discussed that there is a hierarchy of blocks and threads for blocks which defines the grid of threads so looking at how the arrangements can be thought of pictorially we have this n number of block each block is comprising 256 threads as you can see in the picture. All of the threads are trying to progress their computation in parallel. This is how the threads are indexed here. So now let us go back the code here the vectorAdd kernel program once again

(Refer Slide Time: 01:13)

Examples : Vector addition CPU-GPU

```
#include <cuda.h>
#include <cuda_runtime.h>
__global__ void vectorAdd(float*, float*, float*, int);
/*-----*/
__global__
void vectorAdd(float* A, float* B,
float* C, int n){ //CUDA kernel definition
    int i=threadIdx.x+blockDim.x*blockIdx.x;
    if(i<n)
        C[i] = A[i] + B[i];
}
/*-----*/
void vecAdd(float* h_A, float*h_B,
float* h_C, int n)
{//host program
    int size = n* sizeof(float);
    float *d_A=NULL, *d_B=NULL, *d_C=NULL;

    // Error code to check return values for CUDA calls
    cudaError_t err = cudaSuccess;
```

So we have a line here which says $i = \text{threadIdx.x} + \text{blockDim.x} * \text{blockIdx.x}$. So this are again internally defined variables `threadIdx`, `blockDim`, `blockIdx` and `i` is the local variable which is being used by this kernel to compute the global id of a thread of a CUDA thread. Now let us go back to this picture of this thread arrangement and try and understand what is happening.

So for every thread I have value of its block number so for example any of this threads whose global thread if I want to compute so for block 0, I have threads numbers from 0 to 255. So for all this threads the block id is 0, the block dimension is 256 and thread id if any of the possible values from 0 to 255.

(Refer Slide Time: 02:30)

CUDA kernel

A CUDA kernel when invoked launches multiple threads arranged in a 2 level hierarchy,
check the device fn call.

```
vectorAdd<<<ceil(n/256),256>>>
(d_A, d_B, d_C, n)
```

- ▶ The call specifies a grid of threads to be launched
- ▶ the grid is arranged in a hierarchical manner
- ▶ (no. of blocks, no. of thread per block)
- ▶ all blocks contain same no. of threads (max 1024)
- ▶ blocks can be numbered as $(_, _, _)$ triplets : more on this later



Now coming back to the earlier slide, we discussed that blocks can be numbered as triplets that is why we have this facility of blockIdx.x. In this we are only doing everything 1 dimension, so block id's are all linearly numbered and I only have defining block id's doing x dimension. So for all the threads inside this block, we have them with the block id 0; for all the threads in the next block, we have them with the block id 1. All the threads in the last block have the block id N -1.

The block dimension as already said as 256 now again for block dimension, I can have a hierarchy block can be multi-dimensional. In that case the block id's are actually triplets but here it is only in the single dimensional. So it is basically 256 here, I have block id 0 and whatever is the actual thread id comes here. So for the first block, the value of i evaluates to be equal to the thread id.

For the second block the value of i evaluates to its own thread id plus since the block id is 1 so the own thread id plus a 256. So essentially what is this i? If I can think of all this threads arranged together and I am making an arrangement of their ordering, so then their thread id would be 0 to 256 here followed by 257 sorry 0 to 255 here, this should be 256 this should be 255 + 256 like that.

That is the value that is get stored in 9 so it is the global thread id. That gives me the actual id of the thread and with that value of i, I can decide that this thread will act on which element of the arrays. So using this notion of blocks and threads per blocks, I am dividing the arrangement of threads hierarchically. Finally when I do, I have to give the actual computation, I have to compute from this hierarchy what is the exact ordering of this thread and that is defined by this global thread id that I do compute with this kind of access expression.

And then this thread id is used to decide the corresponding thread will act on which elements of the data points in A and B. So as we can see that, let us say, I pick up thread 1 here of block 0. If I take thread 1 of block 0 so that will give me i as 1. So this will write to C[1] the value A[1] + B[1]. If I pick up something ,let us say, thread id x 1 from block 1. So essentially i.e 1 + for block id x have the value 1 times block dimension 256.

So $256 + 1$ that would be the value of i here and accordingly suitable locations in $A[i]$ and $B[i]$ will be used to do the addition. So in that way every thread maps to a different unique id and using that unique id, it is decided which data elements with the thread warp 1 and which data element will be thread update. So with this background, we have possibly a clear understanding of how vector addition is done in parallel using a simple CUDA program. Now let us try and understand this notion of thread and block definition in a more detail with respect to CUDA syntax.

(Refer Slide Time: 06:44)

Kernel specific system vars

- ▶ `gridDim` - no. of blocks in the grid
- ▶ `gridDim.x` - no. of blocks in dimension x of multi-dim grid !!
- ▶ `blockDim` - no. of threads/block
- ▶ `blockDim.x` - no. of threads/block in dimension x of multi-dim block !!
- ▶ For single dimension defn of block composition in grid, `blockDim = blockDim.x`
- ▶ `blockIdx.x` = block number for a thread
- ▶ `threadIdx.x` = thread no. inside a block

So as we have discussed that the packing of threads together is defined as the grid, the grid is divided into blocks and inside each blocks I have the number of threads i.e threads per block. Now thinks can be upto 3 dimensions, here that mean I can have a grid defined in 3 dimensions. So if I say grid dim that is the number of blocks in the grid if I say `gridDim.x` that means the number of blocks in dimension x considering that the grid is multi-dimensional.

If I say block dimension that is the number of threads per block, if I say `blockDim.x` that means the number of threads per block in dimensions x of a multi-dimensional block. So for single dimension of block and `blockDim.x` is same. So whenever we are talking about a single dimension of block, if I say `blockDim`, it is essentially meaning `blockDim.x`.

If I say `threadIdx.x` or `blockIdx.x` essentially they are talking about the single linear dimension. So by default I mean using a single value instead of a triplet in definition of thread id's or the

block id's. I can access them using blockIdx.x as well as threadIdx.x variables. So in that way for a single dimension I have blockIdx.x meaning the block number for a given thread and threadIdx.x meaning the thread number for the thread inside a block.

So for any thread it has got its own values of these variables threadIdx.x and blockIdx.x considering a single dimension. So if I have a hierarchical arrangement of the grid divided into blocks and threads per block both in single dimension then for any thread I will have a value of blockIdx.x and threadIdx.x. This along with block dimension can be used to find out the global thread id, that is the summary.

If I am considering grid to be packed in multiple dimensions, 2 dimension or 3 dimensions then the blocks will be arranged as per following as triplet or duplet arrangement accordingly we will define for blockIdx.x, blockIdx.y. Similarly threads can be packed inside the block in 2 dimension or 3 dimension accordingly we shall have values for threadIdx.x, threadIdx.y, threadIdx.z like that.

(Refer Slide Time: 09:33)

```
__global__
void vectorAdd(float* A, float* B,
float* C, int n){
    int i=threadIdx.x+blockDim.x*blockIdx.x;
    if(i<n)
        C[i] = A[i] + B[i];
}
```

- ▶ The code is executed by all the threads in the grid
- ▶ Every thread has a unique combination of (blockIdx.x, threadIdx.x) which maps to a unique value of i
- ▶ i is private to each thread



So for this vectorAdd code, the code is executed by all threads in the grid. Every thread has a unique combination of blockIdx and threadIdx. Essentially for no thread, I will have all these values same, I mean, block id and thread id all of them will not match. Hence we will evaluate to the global thread id that is the unique value of i which is private for this thread, I mean, unique to that thread.

For example, we already have here, suppose I pick up thread which ID 253 a thread with ID 253 in the block 1 then I can see that for this thread the block id is 1 block dimension dimensions of blockDim.x is 256 and threadIdx.x is 253 so that will give me a value of i and this will decide exactly 256 + 253. So I have 509 that is the value of i. It decides exactly which of the A and B array positions are to be added by specific thread.

So that's kind of summary of how threads are getting declared and arranged inside a CUDA data space, how the threads get to know what is that index in the overall data space and how that index can be used to decide what are the data points on which the thread is going to act on.

(Refer Slide Time: 11:18)

Function declaration Keywords

```
__global__
void vectorAdd(float* A, float* B, float* C, int n)
```

Table: CUDA Keywords for functions and their scope

Keywords and Functions	Executed on the	Only callable from the
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

Following this, we have some discussion on the different keywords that are to be used for declaring CUDA functions. Now if you noticed that for this vectorAdd, we have used a keyword here highlighted in green that is global. Global is a key word followed by standard definition or standard function like definition written type is void because, off course this is the device function I mean this is kernel, it will execute in the GPU device and it has its read and write parameter, the device memory locations which are to be passed to this pointers.

And this keyword global actually is indicator of the nature of this function. So these are CUDA keyword and this keyword helps to define what is the scope of the function. Overall for such functions we have 3 possible keywords device, global and host. So when I say the keyword is

device for a specific function, there meaning that it is a function that can execute on a GPU device and it is callable from the device.

Note that the function we used, here, a keyword `global` which mean, function can execute on a GPU device but it is callable only from the host that is why we add host side vector addition program which was actually calling the kernel or launching the kernel on GPU device. So that is like functions with `global` keyword, there can be function which are device keyword level which mean they are kind of kernels that can be launched from other kernels, basically some other device function. That is why, it is only callable from device function and we also execute on a device function. Now if I use this keyword “host” that is a normal program execute on the host side i.e CPU and they are only callable from another function execute on the CPU, that is essentially a host type function can call another host type function and they are to be level by keyword host. So these are the 3 different possibilities of CUDA functions with respect to their scope.

(Refer Slide Time: 13:39)

CUDA functions

- ▶ Every function is a default `__host__` function (if not having any CUDA keywords)
- ▶ A function can be declared as both `__host__` and `__device__` function
 - ▶ `"__host__ __device__ fn()"`
 - ▶ Runtime system generates two object files, one can be called from host `fn()`s, another from device `fn()`s
- ▶ `__global__` functions can also be called from the device using CUDA kernel semantics (`<<< ... >>>`) if you are using *dynamic parallelism* - that requires CUDA 5.0 and compute capability 3.5 or higher.

Now by default, every function is definitely a host function if nothing is specified. That means the default is host, I mean it is a normal C function. A function can be declared as both host and device. Now, when will you like to do that? The definition will look like this. I have the declaration with the host followed by device, then the functions normal declaration. If it is given like this then the run time function we will generate 2 object device.

By default, as we have seen that the host side function is compiled by the normal CPU's C compiler and linker, pipeline and gives a host side object code. Whereas the device side function compile from GPU to device to create a separate object codes. But if for a function, we have both this host and device stag then the run time system will generate 2 object files. One can be called from host functions and another can be called from device functions.

Since it may be required that the function, sometimes, a host code calls it and it will execute in a GPU or it can be also such that at the run time some other GPU device function calls it. so having that kind of facility, I may like to have 2 of this object files ready and available with me. Now another important thing is this functions which are declared as global by default if you look into the definition.

So this global tagged functions are essentially device functions which can only be called from host side functions. But if you are using CUDA 5.0 with compute capability 3.5 or higher then a global function can also called from the device using the CUDA kernel semantic, that is this location, if you are dynamic parallelism. In case, I want the global function to be also called from other I mean from a device side functionality that requires this support of dynamic parallelism.

(Refer Slide Time: 15:52)

CUDA functions : more observations

- ▶ `__device__` functions can have a return type other than void but `__global__` functions must always return void
- ▶ `__global__` functions can be called from within other kernels running on the GPU to launch additional GPU threads (as part of CUDA dynamic parallelism model) while `__device__` functions run on the same thread as the calling kernel.

So to break it down, what will that mean, let us just explain. First of all, what is the device function? It can have a return type other than void. But global functions must always return void. This is because, a device function is basically called by another function that is executing already

in the GPU. As you can see in the device keyword is function which will be executed on the GPU and it is called by another function from the GPU side only, it is called by device side program.

So definitely, in that case, this function executing on the GPU and return back to the callee which is also resident on the GPU. But that is not allowed when I have a device function executing and it is being called from the host side, basically a device function with global keyword. Because if the callee is resident on separate processor whereas this function is executing on GPU card physically.

That is why the device function that can have return type other than void, a global function must always return void. Then we have to use cudaMemcpy directives to copy back the values to the host side callee. But as we have discussed earlier that global functions can be called from device type functions, in case I have this higher levels of CUDA, this is known as dynamic parallelism.

Let us understand what will happen in this case, suppose a CUDA kernel is executing that means it has launched multiple kernel of a multiple thread. So let us say there are n number of threads that are launched. And in case I have this support of CUDA dynamic parallelism, then this device side function can also launch a global function and suppose the global function is going to launch n number of threads in each invocation.

So overall, I will have effectively n times m number of thread launches. Because this m number of threads of the global function will be launched for each of the n threads that are already executing by the callee device function. So this is known as an example of dynamic parallelism in CUDA which came into existence to CUDA 5.0. So with this, we have discussed the classification of different possible types of CUDA functions that we have based on from where they are called. We have provided a discussion on how a basic CUDA function can be return I mean how a CUDA kernel operates in conjunction with the host program.

How the host program can set up memory elements which are to be copied for as the necessary arguments for execution of a CUDA kernel. Following that copy, how a CUDA kernel is actually invoked and once the CUDA kernel execution is done how the computed values can be copied

back. So with this, we would like to end this lecture and in the next iteration we would like to go bit more advanced concepts of basic CUDA programming. Thank you.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 03
Lecture No # 11
Intro to CUDA programming (Contd.)

(Refer Slide Time: 00:36)

Matrix Multiplication (CPU only)

```
void MatrixMulKernel(float* M, float* N, float* P, int N){  
    for(int i=0;i<N;i++)  
        for(int j=0;j<N;j++)  
        {  
            float Pvalue=0.0;  
            for (int k = 0; k < N; ++k)  
            {  
                Pvalue += M[i][k]*N[k][j];  
            }  
            P[i][j] = Pvalue;  
        }  
}
```

Hi, so after the last lecture, today, we will reviewing a few more examples of CUDA program writing in terms of, I mean, we will start with simple CPU only programs and also show that how the corresponding CUDA variant of the program which will run on a GPU can be written. Now we will have some very simple examples. For example, we will we start with a basic matrix multiplication program.

So first, we consider only the CPU only code. This is the basic matrix multiplication kernel which is designed to run on a CPU. Let us see how it works, as we know that any standard matrix multiplication should have 3 level nesting of loops so we have a outer loop which is essentially iterating over this $i \in 0 \text{ to } N$ and we have this inner loop $j \in 0 \text{ to } N$. So basically I am using i and j to select the i -th row of a matrix M and I am also using the inner iterated j to select a specific column and in that way once in every possible choice of i and j the inner lop carries out the computation of the ij -th value of the result matrix.

There is the essence of any matrix multiplication program that we know a standard for any CPU only case. So essentially we have this outer 2 loops selecting the i and j and inside this inner loop we carry on this computation of the ij-th entry of the output matrix, as we can see that inside this loop what is going on. We are kind of traversing the I-th row of the M matrix and we are traversing the j-th column of the N matrix we are performing point wise multiplication followed by addition and they are result gets accumulated in the variable P value which is written back to the P[i][j] matrix which is showing the result.

So that is how the standard CPU only code works and we like to see that what will be a CUDA variant of this course.

(Refer Slide Time: 03:04)

Matrix Multiplication Host Program

```
int main()
{
    int size = 16*16;
    cudaMemcpy(d_M, M, size*sizeof(float),
    cudaMemcpyHostToDevice);
    cudaMemcpy(d_N, N, size*sizeof(float),
    cudaMemcpyHostToDevice);
    dim3 grid(2,2,1);
    dim3 block(8,8,1);
    int N=16; //N is the number of rows and columns
    MatrixMulKernel<<<grid,block>>>(d_M,d_N,d_P,N)
    cudaMemcpy(P, d_P, size*sizeof(float),
    cudaMemcpyDeviceToHost);
}
```

So for CUDA variant as we know that there has to be a host program and a device program. So this is how the main will look like. So you will have a cudaMemcpy because, off course, we need to remember that we need to send the input matrixes M and N to the GPU side for doing multiplication. So we execute this command cudaMemcpy and you use them to send the matrix M as well as the matrix N to the GPU.

Once this is done, we define suitable grids and blocks here as you can see, they are given some specific numbers here. So essentially, we are defining a 2 dimensional grid and this is denoted by this 2, 2, 1 so we have a 2 dimensional grid in each dimension I have the dimensionality value of

2. So in that way I am having 4 thread blocks that are getting defined and then inside each thread block that I have 64 threads that is getting defined by this definition of block 8, 8, 1.

So inside each thread block I have 64 threads and they are also arranged in a 2 dimensional pattern right. So with the configuration of grids and blocks that defining the arrangement of threads for the program we are firing the matrix multiplication kernel. So this is the code that is going to execute on the GPU and it is assuming that it has got the matrices M and N available to it from the inside GPU DRAM and that is already available due to this cudaMemcpy that are already executed.

Once this kernel executes, I have expecting that the results should be available in another part of the GPU device memory that is `d_P` and I will be using this cudaMemcpy command again to copy back from device to host this values from the location `d_P` to the original target location `P` in the CPU corresponding dram. Let us just have a small look at the basic matrix multiplication kernel essentially it is very simple code so it is going to execute this part of the code sequentially in a par thread basis.

The kernel has been launched with this many number of threads the threads are arranged in the grid, block has been discussed. So if I ask what is the total number of threads that has been launched ? let just work it out. So it would simply be this $2 * 2 * 8 * 8$ so that would give me $64 * 4$ that is the number of threads that are getting launched here.

(Refer Slide Time: 06:12)

Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int N){
    int i=blockIdx.y*blockDim.y+threadIdx.y;
    int j=blockIdx.x*blockDim.x+threadIdx.x;
    if ((i<N) && (j<N)) {
        float Pvalue = 0.0;
        for (int k = 0; k < N; ++k) {
            Pvalue += d_M[i*N+k]*d_N[k*N+j];
        }
        d_P[i*N+j] = Pvalue;
    }
}
```

So for each of this threads, the thread id and block id variables will get their own specific values here. As we have learned earlier also, so for example, the first thread should have a block id 0 and thread id 0 here and so in that way I can use those values to compute i and j , so this essentially is giving me what is the index for which I am going to do the computation. Now let us try to map this properly, so what do I really have? I have got 1 thread that thread has a specific block id and a specific thread id.

So if I do a $\text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}$ then I get an i value and this i value is denoting a row of the matrix M or here that is revise side its d_M and similarly. If I do a computation of j what do I really get here. So I have a blockIdx value and I am multiplying it with the corresponding dimension and I am adding it up with the threadIdx value and this gives me a corresponding column position in the other matrix on the device memory and that would be d_N .

(Refer Slide Time: 07:50)

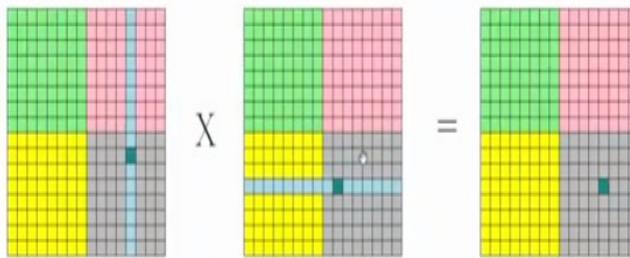


Figure: Matrix Multiplication

$$d_P[i * N + j] = \sum_{k=0}^N d_M[i * N + k] * d_N[k * N + j]$$

So if we take this pictorial example and this gives me the more holistic view of the system. Now have a look at the original code here. So you have defined a grid containing 4 thread blocks and for each thread blocks you have got a block of $8 * 8$ size, that means you have got 64 threads executing. So this is how it is going to be arranged so you have this many number of threads that have been launched and each thread is trying to compute one of the elements of the target matrix.

Question is how will each thread know for which value it is going to do the compute? this is the main problem here. Now as you can see that we have 4 blocks due to this grid definition of 2, 2, 1 each of the blocks are getting marked here in different colors in 4 different colors I have and for inside each of the blocks I am going to have this 64 threads as per the block definitions we have discussed earlier.

Now suppose I pick up any one of this threads let us say the thread has got a this point I am talking about the thread at this point which is highlighted here. So if I ask what is the block id of this thread so it is easy to see that the blocks since the blocks at the dimensions from 0 so I have block id's of this 4 blocks are 00, 01, 10, 11. I would have for this thread the blockIdx.x as well as blockIdx.y value both as 1.

Now suppose I am trying to get back from this blockIdx and threadIdx values what is the corresponding row and column position here in this target matrix for which the thread is going to

do the computation. Now this is what is getting done here. So let us break it and observe it here. I have already understood that how I can get the `blockIdx.x` and `blockIdx.y` values.

Now whatever the values of the `threadIdx.x` and `threadIdx.y` values for this specific thread, as we can see that here, we are looking at the forth row. So `threadIdx.x` is if I say what is the `threadIdx.y`? That should be 3 and what is the `threadIdx.x` so that is this much. So that is 0, 1, 2, 3, 4 so that should be 4. So in that way, looking back for that `threadId` I can compute what is the `threadIdx.y` value and `threadIdx.x` value.

I mean these are the value back the thread as already got and it has got to use this values to compute the `i` and `j`. So that is the motivation of why this equation is there but the issue is now to compute the value. So the value of the row is `i` and the value of the column is `j` so `i` would essentially mean this height there is a row and `j` would essentially mean this shift there is a column.

The height I would be now getting by following this formula. We have to look at what is the block id. If I am talking about `i` so essentially I am thinking in terms of what is the `blockIdx` that `blockIdx` needs to be multiplied by the dimension of the blocks. Since this block id 1 and there was already a block here so you multiplied by the block dimension so you shift this much that is why you have `blockIdx.y` times `blockDim.y` and then you are trying to figure out what is the shift in the `y` axis further inside this block.

Now that shift is as we know 0, 1, 2, 3 so that is the value of the `threadIdx.y` variable so that is why for computing the `i` we will have to write this `blockIdx.y` times `blockDim.y` all in the `y` dimension and we have to add the shift that is the `threadIdx.y` value. So this gives me the value of the row, the target row for which I am going to the computation. Now similarly I can do a computation of the value of the `x`, sorry the value of the `j`.

Now `j` represents, I mean what is the column index for which I am going to do the computation but how do I get the `j` in terms of `xy` axis as I can see that `j` should correspond to the `x` dimension of the block as well as the threads since the `blockIdx.x` is 1 for this block so that means I would multiply it with the block dimension so that is what I do. So `blockIdx.x` times `blockDim.x` and I would like to act with the `threadIdx.x` so that is what I have plus the `threadIdx.x`.

So then in that way using this values of blockIdx.x, blockIdx.y, threadIdx.x and threadIdx.y I have been able to come to the i and j and that tells me what is the target location for which I am going to do the matrix multiplication operation, after been able to recover this i and j value from the values of the system variables in terms of threads and blocks the kernel will proceed into this loop it has a loop which is iterating from $k = 0$ to N inside this essentially this thread would be traversing this row for matrix d_M in the device memory and it will be traversing this column.

So I mean we are sorry so this picture should be actually I mean reverse like this a row should come on this side and this column should come on this side. So essentially it will be multiplying them to get the value here, so just consider this one on this side and the column matrix on the opposite side of this. So anyway looking at that, the question is what should the expression here, as you can see that the expression would nicely match.

So for every value of k all you are doing is in for a fixed row that is the i -th row you have got i times N so you have already covered all the other elements visiting to this row and then you shift by k , so when I am talking about this entity I am referring to this matrix. I think this is on the left hand side. So I would do i times N is the x as well as y dimension of the matrix and you shift by k position to get to any of the k positions here and your multiplying it with the this columns corresponding value.

So that is why you are also thinking that essentially, we are doing computation of the ik -th element times the kj -th element. So the ik th element from this matrix is getting multiplied by the kj -th element of this matrix right. So in that way you multiply the contents of this row point wise with the contents of this column again point wise and you are accumulating that sum and that sum goes to this position.

Now off course why do you do this multiply by N because as we know following C and other programming languages that I mean this is the way we always do for the simple reason that the arrays would be stored in the memory sequential. So coming back to the code for every thread the first thing that the kernel as to do is find out what is the location in the memory for which this kernel is suppose to do the matrix multiplication computation.

That means you are given a kernel when you launch the kernel you launch that many threads you have launch the number of threads as is the size of the 2D matrix right. So 1 thread is responsible 1 thread is going to be responsible for doing a computation of each location in the matrix but it has to figure out what is the location. For that what it does is if it recovers the i and j value the way it does is it as a loop into the block id x thread id x values corresponding to the thread and it tries to figure out what should be the i and what should be the j using this i and j values it is going to find out.

In that case I am picking up this row from the matrix M and this column from the matrix N and does the point wise multiplication and accumulate once this is done the value is stored in this variable P value and it is read back. So the question is all that is going on is for the GPU case the computation is done in a per thread basis the possibly problematic scenario here you have to identify that how to map the computation in terms of thread id's and block id's.

So like you have you are firing the kernel with a suitable launch parameter set that is specifying how the threads are going to be arranged hierarchically and then you are using this launch parameters to identify your working set part thread in this case the working set for this thread is this location ij. And we have explained how to compute the value of i and j block id's and thread id's and you are going to use them for doing the computation.

So in that way you have the matrix multiplication operation done now one thing as to be remember that this is not an optimize GPU program why I say that this is not an optimize GPU program we learnt later in the course where we show that how this matrix multiplication computation can be accelerated by using several primitives that GPU additionally offered to you. Here all we are trying to do is we are trying to parallelize the program.

We are simply saying that the operation of matrix multiplication have got lots of parallelism because I can compute each of the ijth element of the product matrix come independently there are no conflicting rights that are going to happen in the product matrix space. Due to this region I can launch a GPU kernel having as many threads as is the size of the matrix so that each thread is responsible for computing 1 element of the product matrix and the way it does is as we have already discussed.

That it will recover the working set and then it will decide how to do the computation and so every thread is executing this sequence of code completely sequentially and when this kernel is completed we would have got the result available in the GPU memory and from the GPU memory the data has to be copy back to the CPU memory by using the CUDA main copy command.

So that would be our way of doing the matrix multiplication computation. With this, we will complete this simple example of matrix multiplication and thank you for your attention. In next, we will go through another a bit more example and this will be part of our coverage of basic CUDA program without much of optimization. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 03
Lecture No # 12
Intro to CUDA programming (Contd.)

(Refer Slide Time 00:31)

Slightly Advanced Example: Julia Sets

A Julia Set (named after the French mathematicians Gaston Julia who worked on complex dynamics during the early 20th century.) \mathcal{J} represents a set of points contained in the boundary of a certain class of functions over complex numbers.

- ▶ Given a set of points in a complex plane, the set \mathcal{J} is constructed by evaluating for each point, a simple iterative equation given by $Z_n = Z_n^2 + C$ where Z_n represents a complex number and C represents a complex constant.
- ▶ A point does not belong to \mathcal{J} , if iterative application of the equation yields a diverging sequence of numbers for that point.



Hi, So let us go for some more advanced examples on parallel program writing in CUDA and here we will pick on some mathematical thing entity called Julia sets. So what is a Julia set? It is named after French mathematician Gaston Julia who worked on complex dynamics during the early twentieth century. Now J represents a set of points contained in the boundary of a certain class of functions defined over complex numbers.

And we define this kind of shapes as follows that you are given a set of points in complex plane a set J can be constructed by evaluation for each point a simple iterative equation which is given by this. So J_n is iteratively computed using the previous value of J_n being squared plus a constant complex constant C being added. And if a point does not belong Julia set then it would provide a behavior, that is, in iterative application of the equation will yield a diverging sequence of numbers for that point.

So essentially what would happen is for certain specific points on the complex plain if I continuously iterate over that equation it will I will get a sequence that is diverging then I will say that the point does not belong to J and otherwise if I get a sequence which is not diverging I will say that yes this point is part of the Julia set. Now of course for you can understand that this computation is very much a function of what is my choice of this complex constant for different choices of complex constants I will be getting different Julia sets.

(Refer Slide Time 02:17)

```
Complex Numbers (CPU)

struct Complex {
    float r;
    float i;
};

float magnitude(struct Complex a){
    return ((a.r * a.r) + (a.i * a.i));
}

void add(struct Complex a, struct Complex b, struct Complex *res){
    res->r = a.r + b.r;
    res->i = a.i + b.i;
}

void mul(struct Complex a, struct Complex b, struct Complex *res){
    res->r = (a.r * b.r) - (a.i * b.i);
    res->i = (a.r * b.i) + (a.i * b.r);
}
```

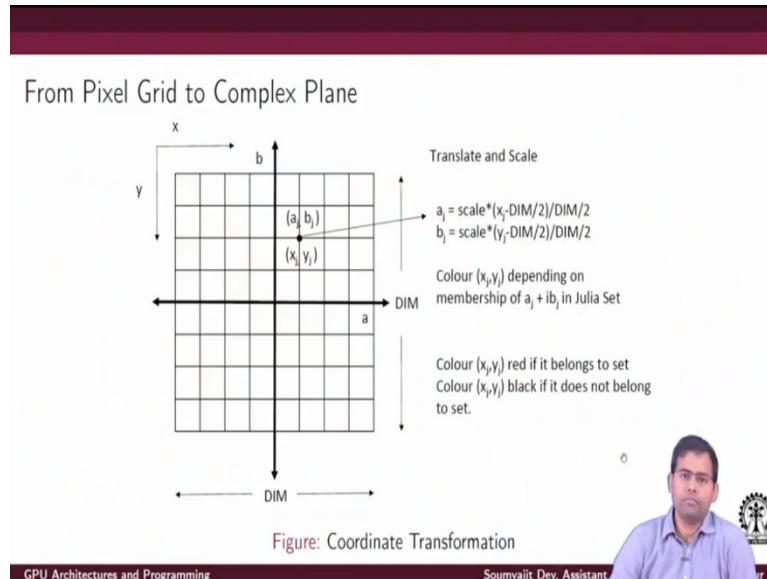
GPU Architectures and Programming Soumyajit Dey, Assistant Professor

So I mean just again I mean how do we represent complex numbers in CPU's. So I mean you can have a structure complex with variables r and i I mean they are there I mean our float variables and they represent a real and i mean real imaginary part and I can have several different operations for example computing the magnitude and doing an addition and also doing some basic multiplication.

So if I look at doing an addition I am expecting 2 complex numbers are provided and I am going to write back the result of the addition by using a pointer to a complex types structure arrays. So it is as simple as this and again I can do a multiplication in the complex plane using standard mathematical formulas like this. So as you can see that I have been provided with complex type input a complex type input b and I am do I am computing these results of the multiplication in terms of the real part and the imaginary part separately.

So this first equation will evaluate the real part and the second equation will evaluate the imaginary part and once it is done I have the result return back in this point using this pointed arrays.

(Refer Slide Time 03:32)



So the first thing our program will do is it will try to make a mapping from the pixel width to the complex plane. Of course we are trying what we are trying to do our objective here is we will also like to write the GPU code to compute a Julia set and represent those values as a bitmap on the screen that will kind of create a very nice looking pattern. So for doing that the screen is represented by a pixel grid and I have to map each point from the screen is coordinate from the x y location to a corresponding a b location.

So this is how I have the x y's coordinate starting from this corner and here I am going to compute the corresponding a b. So first thing we will do is we will do a for computing this ab points for the given xy points I do a scaling. Before that scaling I subtract from this xa dimension by 2 but and that gives be a value here with the respect to origin. And also if I do a subtraction over the y that also gives me a value in terms of b.

And I do a further division by dimension by 2 to do a scaling. So when I say that I have a corresponding value in terms of ab and I will be doing some computation on this and then I will determine whether this value is whether this coordinate is a member of Julia set or not. The fact

that it is member will be represented by a different color. So the color coding we are going to follow here is that color xy color xj yj.

If it is red that would mean that it belongs to the set and if it is black that means it does not belong to a set. So what all I would expect at the end that there is a blackened screen and on which I have a red colored pattern which is representing the Julia set.

(Refer Slide Time 05:36)

Julia Function for a point (CPU)

```
int julia( int x, int y) {  
    const float scale = 1.5;  
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);  
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);  
  
    struct Complex c,a,r1,r2;  
    c.r=-0.8;c.i=0.154;  
    a.r=jx; a.i=jy;  
    int i = 0;  
    for (i=0; i<200; i++) {  
        //a = a*a + c;  
        mul(a,a,&r1);  
        add(r1,c,&r2);  
        if (magnitude(r2) > 1000)  
            return 0; // return 0 if it is not in set  
        a.r = r2.r;  
        a.i = r2.i;  
    }  
    return 1; // return 1 if point is in set  
}
```

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor

Now how do the Julia function look like for a specific point I mean when I am doing a CPU side computation and I am trying to evaluate whether a specific point on the screen is a member of the Julia set or not. So you assume that you are supplied with the xy coordinates that is the coordinates with respect to this corner here. The first thing you are going to do is your going to do this transformation in terms of scaling and you are trying to find out what is the corresponding ab values of the coordinate.

Of course DIM represents that dimension that means the total size here of the screen size that I am considering and I am also considering that it is a square sized area or DIM cross DIM So using this transformation that we have discussed here in terms of aj and bj we now create this variable in this as jx and jy. So when I have these jx and jy created I used them to now denote the new coordinates here in terms instead of x and y.

So this essentially will give me a result of applying this formula here. So essentially I will be doing all my Julia set computation on these jx and jy and it is very standard. So just to look at the original mathematical formula all we are supposed to do is we are supposed to iterate over this equation multiple times see what are the result? Whether the result is converging or not and based on that we take a decision whether it is a member of Julia sets or not.

And of course, there is this complex param constant parameter which we assume is provided to us. So here for our purpose this complex parameter is assumed to have a real part this and imaginary part as given here. And so we consider that the input value has a real part that is jx and the input value has an imaginary part that is jy . And then we are into inside this loop which is iterating 200 times and it is basically executing this equation $a \times a + c$ getting recomputed and recomputed.

So for doing that since a is of the type complex and we carry out this function multiply. Let us you multiply a and a write back the result in $r1$ and then you take the content of $r1$ and you add it to c and then you return it $r2$ and you check whether the magnitude is greater than something some space given value or not. If it is really so then you return back, you say that the item is not in the set otherwise you say that yes indeed the item is member of the set.

The way you say this it is a member of the set is you write back this value of $r2$ here and you also go back to the computation here. So at the end what will happen is if inside this 200 number of iterations it is never the case that the magnitude is crossing this thousand then I have the final value here in a and it is and if the loop executes up to this 200 iteration never returning back here then you finally make a return from this point and you have the final value that is computed stored in the complex data type a .

(Refer Slide Time 09:37)

Driver Code(CPU)

```
void kernel( unsigned char *ptr )
{
    for (int y=0; y<DIM; y++)
    {
        for (int x=0; x<DIM; x++)
        {
            int offset = x + y * DIM;
            int juliaValue = julia(x,y);
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
}
```

A 32 bit per pixel color bitmap represents a 2D grid of pixel values where each pixel is represented by 4 channels (R,G,B, α) and where each channel has values in the range [0 – 255]. (α represents transparency).

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor



Now so this is a code that is trying to design a membership decision for a provided x, y value that would be summary here. You are given the x, y value from that you are trying to create a corresponding complex number a you evaluate this equation. You see whether it is diverging or not. If you and your definition of divergence here is that I will execute this loop 200 times.

If even after that this magnitude does not cross thousand then I will simply return back assuming that yes this is a member of the Julia set corresponding to of course this choice of complex parameters see here. Now of course there is a membership decision function and it has to be called so for that I have a driver code which is here in terms of the kernel. So essentially I am thinking that for the entire input bitmap there is representing the entire screen area.

I am going to do a pixel by pixel membership testing for whether that because any specific pixel is a member of the Julia set or not that is my target. So for that I set up this nice cascade of loops each loop is a rating from 0 to dimension because off course I am going to cover the entire screen here. So coming back here I have this cascade of loops from 0 to dimension and inside this cascade what i do is first I make a computation of this offset that means what is the location I am interested in offset is of course x + y times the dimension.

Why that is required we will come into that but before that we compute the Julia value that means I passed these parameters x and y to this membership testing function Julia that we have discussed earlier. And in return backs and tell me that yes this x y value is indeed a member of

this I mean this x y parameter value is indeed a member of Julia set or not. So that is available here so if it is really a member then Julia value would get a value 1.

Now if it gets a value 1 then we have this corresponding location. So that means the offset times $4 + 0$ is being provided with 255 is being assigned the value 255. Now why do I have this offset getting multiplied by 4. Essentially we are thinking that in standard bitmaps a 32 bit pixel color bitmap is being represented by a 2D grid of pixel values and each pixel has got its color being represented in terms of 4 channels. So each channel is for red, green, blue and alpha. Where alpha represents the transparency and each channels value is to be inside this range of 0 to 255.

So I mean with proper with suitable combination of this values I can generate the corresponding color. So since I have to give for any pixel value I have give the corresponding value of these channels this is how we are doing it. First we are thinking that this ptr array is representing the bitmap. We do the offset computation to traverse to the corresponding location in the ptr. So offset is nothing but $x + y$ times the dimension.

So essentially I can think that these offset represent one cell here. Now the value in this cell I mean it is of course each cell has got 4 channels so this are the multiply by 4. And after that you have to keep the corresponding value for each cell. So that is why you have 4 values $0 + 1 + 2 + 3$. As we have decided earlier that if it is a member then we are going to give a color it with red and otherwise it is all black.

If I have to color it with red but we do is Julia value is 1. In the first channel represents red. So I multiply 1 with 255 and rewrite in the red channel. For the other G and B channels we anyway keep them at 0 and we also have the transparency value to full. So this is the kernel code for CPU which is going to take each of the ptr's that means each pixel position and transform and tell me back whether this is going to be a really inside a member of Julia set or not and if it is a member of the Julia set then it is going to paint that pixel position by itself.

(Refer Slide Time 14:37)

Driver Code(CPU)

```
int main( void )
{
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();
    kernel(ptr);
    bitmap.display_and_exit();
}
```

We leave out intricate details of how bitmap data is constructed. The primary focus of discussing this application lies in depicting the underlying computation involved in constructing Julia sets.



GPU Architectures and Programming

Soumyajit Dey, Assistant Prof

So this is the kernel code in terms of the CPU. Now of course that code needs to be called for that we will have a main function in c where we first do the bitmap setup by using these functions I mean. So these are the basic bitmap set up which will return the CPU bitmap and we get the pointer that I mean this p pointer which is going to point to the screen locations.

Now of course in this context we leave out the details of how bitmap data is constructed. Since the primary focus here is that we are trying to see that how to do so some computation on the bitmap. Of course I mean getting bitmaps and with using suitable headers and all that is very simple. There are standard programs and we of course provide you the references here for that.

So here we get the pointer to this differ bitmap for the screen we call the kernel with these pointer locations for each of them and we finally do a display and exit here. Now how about doing this computation for the GPU side of the program. Now again I would like to say that this kernel code that has been demonstrated here is only for one location here. But how does it work because the call has been made as you can see for the base position.

This CPU side port makes the kernel called for the base position and then from the that mean I am passing kind of this location in ptr and then I have these two outer loops which are going to traverse through the entire grid of locations and do the color compute a this coloring part and before that it is going to do the membership testing by using this Julia function.

So this as you can see that just like for matrix multiplication I had a cascade of loops first the cascade of loops the outer cascade of 2 loops were there to choose the row for the matrix 1 and the column for the matrix 2 and then the inner loop was doing the computation. So in a similar structure I have this outer 2 cascade of loops which are traversing over this bitmaps. And that inner code is actually doing the Julia set computation and then deciding whether this is the member of Julia set and then correspondingly coloring up with a suitable value whether it is red or black.

So the overall idea is very mean very simple that since we are assuming that the CPU is single threaded. So there is a one thread of computation that is going to over this multiple outside cascade loops and in each iteration of the inner loops it is going to do the Julia set computation and coloring business.

(Refer slide Time 17:30)

```

Complex Numbers GPU

struct cuComplex {
    float r;
    float i;
};

__device__ float magnitude(struct cuComplex a){
    return ((a.r * a.r) + (a.i * a.i));
}

__device__ void add(struct cuComplex a, struct cuComplex b, struct cuComplex *
    res){
    res->r = a.r + b.r;
    res->i = a.i + b.i;
}

__device__ void mul(struct cuComplex a, struct cuComplex b, struct cuComplex *
    res){
    res->r = (a.r * b.r) - (a.i * b.i);
    res->i = (a.r * b.i) + (a.i * b.r);
}

```

GPU Architectures and Programming Soumyajit Dey, Assistant Professor

Now what about the corresponding code in the GPU. So again we will have assumption that we have some device functions here. Now if you remember for in terms of GPU kernels we made a classification there is a device function as well as global function. So the global function is a kernel that can be called from a host side code and a device function is a GPU kernel which can be called from some other function that is already executing in the GPU. So just for your remembrance let us just traverse back to this functional declarations.

(Refer Slide Time 18:10)

Function declaration Keywords

```
__global__
void vectorAdd(float* A, float* B, float* C, int n)
```

Table: CUDA Keywords for functions and their scope

Keywords and Functions	Executed on the	Only callable from the
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

So here we had device type function which execute on device and they are callable only from device functions. Whereas we have global functions which executes on device but they are callable from host side functions. So here as you can see we define some device site functions so one for magnitude one for addition one for structure. Now these are just like our earlier programs which were basically the c programs which represented doing this computation in the complex plane in terms of magnitude addition and multiplication.

So they are just very similar here and I mean because they are like standard c functions there is no hint of parallelism or anything here.

(Refer Slide Time 19:13)

Julia Function GPU

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    struct cuComplex c,a,r1,r2;
    c.r=-0.8;c.i=0.154;
    a.r=jx; a.i=jy;
    int i = 0;
    for (i=0; i<200; i++) {
        //a = a*a + c;
        mul(a,a,&r1);
        add(r1,c,&r2);
        if (magnitude(r2) > 1000)
            return 0; // return 0 if it is not in set
        a.r = r2.r;
        a.i = r2.i;
    }
    return 1; // return 1 if point is in set
}
```

We also have another device function which is named Julia. Now this is the function that is going to do the computation that is the membership of the Julia set and that is also quite similar as you can see that you do the computation of the positions on the with respect to the shift of origin and scale first with respect to jx and jy and then you use those value to initialize a complex data type a you make your simple choice of c I mean in this complex constant values with respect to both the imaginary and real parts here.

And you define and you define these different types of I mean complex types I mean were just calling cu complex where the CUDA equivalent part. And so you have this complex constant c which has been defined and you have the input a you have this original loop here I mean this just like the normal c code. Essentially you are doing the multiplication then the addition and then you are doing the magnitude testing whether it is converging or not.

And if you are able to execute this code 200 times without ever returning out then you say that this is going to be the member of this Julia set. So this is just like the normal CPU side code. But then then the question is why do I start saying that yes it is a GPU, GPU site code. The reason is this is a device side function which will execute on the GPU and it will be called from some other GPU kernel.

So now let us look at the other GPU kernel. So this is it so this is a global function that means this is the GPU kernel which is going to be launched from some host side code. What does the GPU kernel do? The GPU kernel first identifies the x and y values. So that is essentially it can find out based on the block id and the $blockIdx.x$ and $blockIdx.y$. Now to get an idea that how is this block defined.

(Refer Slide Time 21:20)

Host Program

```
int main(void) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;

    cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() );
    dim3 grid(DIM,DIM);
    kernel<<<grid,1>>>(dev_bitmap);

    cudaMemcpy(bitmap.get_ptr(),dev_bitmap,bitmap.image_size(),
              cudaMemcpyDeviceToHost);
    bitmap.display_and_exit();
    cudaFree(dev_bitmap);
}
```

GPU Architectures and Programming Soumyajit Dey, Assistant P

Let us go further to the host program so this is the host program. As you can see in the main I have the CPU bitmap a I mean that function like earlier which is providing me the bitmap definition and then I do a CUDA Malloc on the GPU side memory. On the GPU side memory I am trying to transfer the bitmap and then I design a grid here of type dimension 3.

And then in this grid definition as you can see when the kernel is called.

The kernel is called with this block and number of threads per block as one. So what is a block here a block is just a containing this dimension, dimension number of blocks. So essentially that would mean for each position I just have a block Id and blockIdx and blockIdy and every block has got just one thread. So that is why with comma I have a one here. These things will become more clear as we progress with a course and then we have a CUDA Mem copy comment.

Once the kernel has executed we can assume that all the set computational has been done and accordingly the bitmap has been painted and then the bitmap display function is called and after that we have a CUDA Free for the device bitmap.

(Refer Slide Time 22:50)

CUDA Kernel GPU

```
--global__ void kernel( unsigned char *ptr) {  
// map from threadIdx/BlockIdx to pixel position  
int x = blockIdx.x;  
int y = blockIdx.y;  
int offset = x+y*gridDim.x;  
  
int juliaValue = julia(x,y);  
ptr[offset*4 + 0] = 255 * juliaValue; // red if 1 , black if 0  
ptr[offset*4 + 1] = 0;  
ptr[offset*4 + 2] = 0;  
ptr[offset*4 + 3] = 255;  
}
```



GPU Architectures and Programming

Soumavir Dev Assistant Professor

So coming back here we have this CUDA kernel whom I have provided with these many number of threads. 1 thread per block and then number of blocks we have defined is dimensions square. So using the block ID I am getting a direct map to the pixel I hope that is clear. So we go back to this nice picture here. So essentially now we are saying that we are defining we are launching these many threads as is the number of pixel and I am also saying that we are launching these many blocks that means essentially we are saying that we are launching these block with only one thread per block.

So once that launch is being done every x, y value of course will get the corresponding block Idx dot x and block Idx dot y and using them I first do the computation of offset. That is nothing but I mean I have the x axis + the number the y value that is the number of rows that would be getting multiplied by the dimension of the grid. Now the dimension of the grid has been set as you can see the dimension of the grid is set as the original DIM dimension value that we have discussed earlier in that graphic slide.

So now for each thread in this kernel I have got the idea that this thread has to decide for the Julia set membership position of 1 x, y coordinate. So for that this Julia function will now be called and this Julia function is just like the original Julia function for the C code is just a GPU equivalent one but it is a device function which means this is already a kernel. This kernel has been launched with dimension times dimension that means a number of threads where each thread is suppose to look up to the membership of 1 element, 1 pixel.

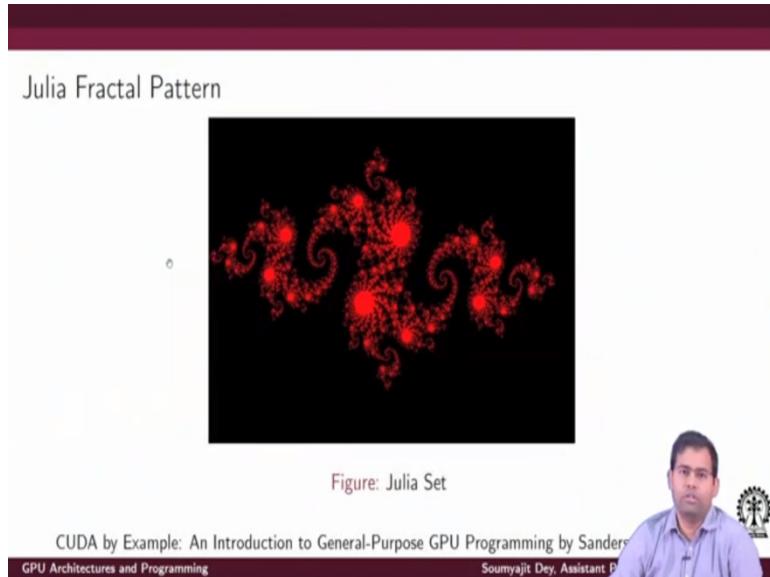
So since this is already a kernel and it is going to launch a C code which will do the membership testing that is why this Julia function is now being called a function which is a device side function. So it has got this device keyword tagged with it. So once being passed with the parameter x and y it should return back the Julia value and just like earlier we will be using this Julia values to assign the corresponding suitable channels here to this ptr memory locations.

And with that when this kernel execution finishes we have got the bitmap colored properly with respect to the Julia set computation. And finally the bitmap display image is able to display properly. So again where does it get different in terms of the CUDA program and original GPU side program. Like we have discussed earlier the original CPU side program had this cascade of loops.

But now when we start talking about the Julia set computation we just do not have this cascade of loops. Instead of that we have this kernel launch and the kernel is launching that many dimension times dimension number of threads. And those number of threads essentially are executing parallel in the GPU and while executing parallelly in the GPU each of the thread is responsible for doing membership decision for each position here.

And for each position it is doing the it is calling the device function Julia. And if the call returns to one then it is doing a proper coloring of the bitmap here. Of course it is a one then it is writing the red channel otherwise it is writing 0 here and so everything is black with all that when this code is executed the host program finishes we will get a nice pattern here.

(Refer Slide Time 27:04)



As we have said earlier that what pattern you get is also a function of what is the coefficient that you choose. This was our choice of coefficient that we made for the complex constant c for that coefficient choice, you would get this kind of pattern. Now this part of the code I will like to mention that this has been adapted from an introduction to general purpose GPU programming by Sanders et al.

And we used that reference to write this program execute them in our environment and reproduce this picture. So this we thought would be a nice example after matrix multiplication. In both the examples we are trying to show you how iterative job that the CPU does by having multiple hierarchies of loops can be replace by suitable kernel launches. Where you launch that many number of threads and each thread does some specific job on some specific data element.

None of these codes are optimized with respect to the GPU's architectural features. But all it does, it exploits the parallelism on based on the job at hand. First we have the matrix multiplication program as we discussed there is lot of parallelism because I can compute each of the matrix into these in parallel. Similarly in the Julia thread Julia fractal case I am trying to take a membership decision about each of the pixels in the display and that also can be done in parallel.

And we thought this would be a nice example because this is also an example of a graphics program through which you are trying to see that how GPU acceleration can provide you good real time graphics performance, although we are going to deal more about general purpose

graphics processing units but graphics program were the first work loads that were identified for graphic processing units. So with that we like to end this lecture. Thank you.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 03
Lecture No # 13
Multi-dimensional mapping of dataspace; Synchronization

Hi everybody. We will be starting with the fourth topic in this lecture series focusing on this idea of multidimensional mapping of the data space for a CUDA program and also the related concepts of synchronization among parallel CUDA threads.

(Refer Slide Time: 00:47)

Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

So just to get back into the overall perspective, we have discussed this idea of basic computer architecture and then we have migrated to more detailed ideas on GPU architectures. We have introduced the basic syntax of CUDA programming and from this point we are trying to get into some more finer details of CUDA programming like how to handle complex data structures, how to synchronize threads which are operating in parallel etc.

(Refer Slide Time: 01:19)

Multi dimensional block

In general

- ▶ a grid is a 3-D array of blocks
- ▶ a block is a 3-D array of threads
- ▶ specified by C struct type `dim3`
- ▶ unused dimensions are set to 1

So that is what we will focus on here. If you remember our lectures on basic CUDA programs and examples of matrix multiplication and similar kind of programs that we discussed about, there was this notion of kernels getting launched and as the launch parameters we were supplying certain parameters which is the number of blocks and number of threads inside the block.

So that is how we define them and we also said that this is something we will take up as a future topic. This is where we will discuss it finally. So when we say that CUDA kernels is getting launched, essentially we launch a set of threads packed into a finer element known as block. We launch multiple blocks of threads and this entire arrangement of threads is called a grid.

So in general when I am talking about the overall idea of CUDA threads that are getting launched, I am launching a grid of threads which essentially is nothing but a 3D array of blocks. Inside a grid, I have entities known as thread blocks or blocks. I can pack a block in 3 dimensions. We will see 3 such examples.

A block is nothing but a 3D array of threads. Overall I have threads arranged as 3D arrays packed in data structures called blocks and the blocks are again arranged in 3 dimensions into packings which are finally defined as grids. These 3 dimensional definition keys correspond to specific data structure type which is a specific structure defined in CUDA. In the extension of

CUDA semantics, we call these structures dim3. Since I am not really interested in extending the dimensions in this example here, the unused dimensions are set to 1.

(Refer Slide Time: 03:37)

Multi dimensional grid, block

```
dim3 X(ceil(n/256.0), 1, 1);
dim3 Y(256, 1, 1);
vecAddKernel<<<X, Y>>>(..);
vecAddKernel<<<ceil(n/256), 256>>>(..);
//CUDA compiler is smart enough to realise both as equivalent
```

I am trying to define such a 3 dimensional packing of the grid of threads for launching the vector addition kernel. If you remember we are supplying these 2 parameters corresponding to the number of blocks in your grid and the number of thread per block.

Since these are individually all 3D packings. both X and Y correspond to dim3 data types. So you define it like this that X is a packing of blocks where you have this many number of blocks in the X dimension and you do not proceed in the Y and Z dimension so these are set as one.

So essentially you are saying that given some **n**, the total number of elements to add for, you divide it into blocks of size 256. So overall you will have ceiling of $n / 256.0$ blocks. I hope you can see this and so you are essentially trying to define these many number of blocks and these 256 is here because essentially you are defining a block as a collection of 256 threads. Again you are not going to multiple dimensions while you are defining blocks.

So the blocks are also defined by the variable Y whose type is again dim3 and this Y is again also defined in the single dimension that it is going to contain 256 threads. Overall you have **n** number of threads divided into $n/256$ number of blocks and inside each block you are going to have 256 threads. This is the arrangement that I am choosing here right so with this definition of

X and Y when you are launching at VecAdd kernel i.e you are passing this X and Y as launch parameters for the kernel.

So when the kernel is launched, the threads get packed into I would say tightly neat blocks of 256 size and in total this $n / 256$ number of blocks are getting launched. I can write formally like this because this is the usual way you will define it. In every CUDA kernel we will expect as launch parameters 2 elements here each being dim3 tuples. However, since this is the simplistic single dimension kernel I can also directly write like `<<<ceil(n/256.0),256>>>`. This comma actually separates the number of blocks and the number of threads per block definitions.

Since I have only two entries the CUDA compiler would be smart enough to identify that I am really talking about a single dimensional grid containing blocks only in the X dimension and indexed in the X dimension again inside the block. Since, I do not have multiple dimensions, it will automatically assume these kind of definitions as discussed earlier here. And so it will also think that inside each block I have got all the threads packed in each dimension.

(Refer Slide Time: 07:13)

Multi dimensional grid, block

- ▶ $\text{gridDim.x/y/z} \in [1, 2^{16}]$
- ▶ $(\text{blockIdx.x}, \text{blockIdx.y}, \text{blockIdx.z})$ is one block
- ▶ All threads in the block sees the same value of system vars `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
- ▶ $\text{blockIdx.x/y/z} \in [0, \text{gridDim.x/y/z - 1}]$

Now this is the most important part. I believe in the earlier program example, we already discussed that you have a simple CUDA program. There are certain variables like thread ids and block ids which get automatically defined and every thread has got its own local value of the thread id and block id. We will get back to that again from the point of view of all multi-dimensional grid and block here.

So in general when we are defining a 3 dimensional grid containing the block indices in 3 dimensions, essentially the runtime system of CUDA would automatically define this variable. So these are system variables that the program can always access. You do not need to define them or you cannot actually define or initialize them. So the moment you launch the kernel with suitable launch parameters, you have automatically available values of these variables i.e. for this CUDA kernel what is the grid dimension in the X axis, Y axis and Z axis ?

These values can range from 1 to 2^{16} . That is the maximum dimension of the grid allowed in each of the 3 axes. So for each block we will have these 3 block id variables providing me the block's actual id in the 3D collection of 3D packing of blocks. That would mean for every individual CUDA thread, I would have a collection of these 3 variables. I will have their values available so that the thread can transparently see what is the block id in the x dimension y dimension and the z dimension.

And of course for all the threads sitting inside 1 block these values should be same, i.e for all the threads inside 1 block they can all access `blockIdx.x`, `blockIdx.y` and `blockIdx.z` and these combination of values should be exactly same because they are technically inside the same block. So just to summarize, all the threads in the same block would see the same value of these system variables `blockIdx.x`, `blockIdx.y` and `blockIdx.z`.

Again you can understand definitely that these `Idx` variable values should be varying inside this range of grid dimension. Because it is obvious you are essentially defining a grid which is constituting a 3D packing of blocks. The grid is the 3D packing of blocks and each dimension - the grid dimension is ranging from 1 to 2^{16} . So every block is definitely in the x direction or the y direction or the z direction whichever way you go. The blocks id has to be less than the grid's allowed dimension in that x, y or z axis.

(Refer Slide Time: 10:33)

Multi dimensional grid, block

block dimension is limited by total number of threads possible in a block - 1024

- ▶ $(512, 1, 1)$ - ✓
- ▶ $(8, 16, 4)$ - ✓
- ▶ $(32, 16, 2)$ - ✓
- ▶ $(32, 32, 32)$ - ✗

Now there is some other significantly important thing here which is that a blocks dimension is limited by the total number of threads that you are allowed to pack inside a block and this number is 1024. So when you are launching a CUDA kernel the blocks, total number of threads packed into the block cannot be more than 1024. Now of course this is a number that may vary with architectural families which may be in future significantly higher compute capable architectures of the GPU's. This number may change. This is something which comes as a restriction based on the architectural content - the maximum register file size possible and all that.

So right now we can assume that a block has a max size i.e there is a upper bound on the total number of threads that you can pack inside the block and it is 1024. So that automatically restricts your possible block IDs i.e. your possible definitions of a block right. Because as you can see we have given here an example of 4 possible block definitions. In the first definition we are saying that let us pack all the threads in the x dimension.

So you are defining a block of size 512. All the threads are in the x dimension so that is perfectly fine because the total number of threads is less than 1024. Same thing for the next 2 definitions. However if you look at the fourth definition we are saying that let us define a block where you have 32 threads in the x dimension. You are letting the `threadIdx.x` variable go from 0 to 31 and similarly `threadIdx.y` go from 0 to 31 and `threadIdx.z` from 0 to 31.

So if we do that then the total number that is $32 \times 32 \times 32$ the number of threads is of course greater than 1024. This is not allowed.

(Refer Slide Time: 12:48)

Multi dimensional grid, block declaration

Consider the following host side code

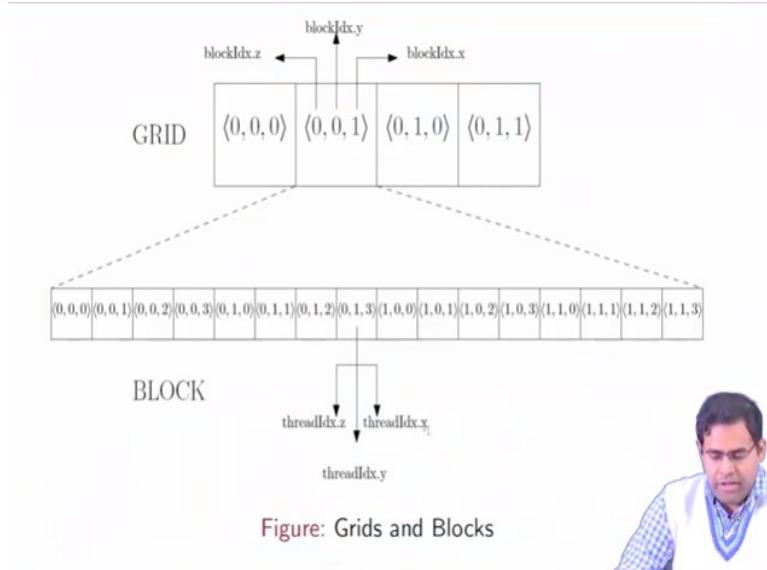
```
dim3 X(2, 2, 1);
dim3 Y(4, 2, 2);
vecAddKernel<<<X, Y>>>(..);
```

The memory layout thus created in device when the kernel is launched is shown next

So as a further example of grid and block declarations, let us consider this following piece of host code now. Why again do I bring in this id of another host code? Because of course we had another example. This was the much simpler example, because technically we had the packing of blocks all in 1 dimension and inside the block the packing of threads also in 1 dimension.

Hence, just to visit a more complex example, let us look at this. So here we are saying that okay let us consider a grid definition where the blocks are packed in 2 dimensions. So in the x dimension, you have 2 blocks and in y dimension you have 2 blocks and also you have inside the block a 3D arrangement of threads. So you have another dim3 type Y with these arrangement (4,2,2) and overall you are making a vecAdd kernel call with launch parameters X and Y.

(Refer Slide Time: 13:55)



So this would create a memory layout in that device and the kernel when it is launched, this is the corresponding memory layout for that. We have a picture here. Now this is something which is very important. We are just trying understand through this picture that what is the impact of this kind of definition of the packing of threads on the variation of the thread id and block id indices.

So since I have defined the grid here as a 2D arrangement, we have as you can see this `blockIdx.z` variable is always 0 right. So that is why this is like more of a hierarchical figure. On the top I just show the arrangement of blocks. As you can see ,we have defined the number of the blocks collection as (2, 2, 1) X. So the first entry is corresponding to the x dimension, the second entry corresponds to the y dimension and the third entry corresponds to the z dimension.

So coming here the way the indexes are arranged here is the opposite. So at the very first MSB i.e. most significant position, I mean the LSB, the least significant at the extreme right I have the x dimensions encoding which is increasing and then I have the y dimension encoding and then I have the z dimension's encoding. Since z is set as 1 here, we are not going to increase the z index.

So it is always 0. Of course the index calculation starts from 0 and they go up to the next dimension in the x or y space -1 right which would mean here for this definition, my thread indexes in the x dimension are going to vary between 0 to 1, y dimension from 0 to 1 and z

dimension has to be fixed at 0. So that is what we have here all the z dimension encodings are set to 0 right.

The x dimension encoding they vary between 0 and 1. I guess you can see 0, 1, again 0 again 1 and as you can see that the y dimension where the encoding is also varying from 0 and 1. But if we were trying to show the arrangement here we are increasing in x first and then we are saying that let us increase in y and we are saying that the blocks are arranged like this one after the another.

But now we need to look deeper. So this is how we will have the blockIdx variables varying for each of the blocks. So we are having 4 blocks and these are the 4 possible triplets or collections of block id variables that we will have for each of these 4 blocks. Now what about the situation inside the block? Of course inside the block I am going to have the threads and I am going to have $4 \times 2 \times 2$ i.e. 16 threads.

How are the threads going to be indexed? We can see that this is again a packing in the 3D space and you are going to have threadIdx.x variable to range from 0 to 3, threadIdx.y to range from 0 to 1 and threadIdx.z again going to range from 0 to 1. That is how we have it here. So we start with the thread with indexes all 0 and then I have the threadIdx.x increasing to 1, 2, 3. Again we will start from 0 , the threadIdx.y increases from 0 to 1.

So I have again 4 consecutive locations where it's the same encoding at the x position as the first 4 , but in the y position, we instead of having a 0 are having a 1. And then again this thing continues in this similar pattern which is just like counting over multiple dimensions instead of now having 0 in the z dimension you just start having 1 in the z dimension and repeat the scheme the again.

So then considering 1 block inside that block all the threads would be having index triplets arrange like this if you specifically pick up one single thread for example let us pick up this thread so these are threads for which as you can see that the system variables would be assuming values like this. So the threadIdx.x variable would assume the value 3 threadIdx.y would assume the value 1 threadIdx.z variable would assume the value 0.

For this thread the block id variables would assume values from here similarly. I would have this arrangement repeating for the next block again, this arrangement repeating for next block. So if I pick up some thread from this block, it would have a specific collection of thread ids x y and z right and for that the block id variables would have these 3 values right. I hope this is clear that how each thread can figure out i.e can see what is the corresponding system variable values for it.

That means if you are going to write a CUDA program, you can actually compute that what is the actual position of single thread among this sea of threads by computing a suitable expression using these variables of `threadIdx.x`, `threadIdx.y`, `threadIdx.z` and `blockIdx.x`, `blockIdx.y`, `blockIdx.z`. Because for every thread this collection of all these 6 parameters is going to be unique. It may happen that I have this triplet of `threadIdx.x`, `threadIdx.y`, `threadIdx.z` same for 2 different threads in the entire kernel.

But of course, they have to be from different blocks. So all though the `threadIdx` collections are same, the `blockIdx` collections have to be different. So in that way I can say that for every thread this collection of the overall 6 parameters we will always be unique.

(Refer Slide Time: 20:40)

	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7
Row 0								
Row 1								
Row 2								
Row 3								
Row 4								
Row 5								
Row 6								
Row 7								

Figure: 2D Matrix



Now let us take an example here that how really is this definition of threads and blocks going to help me? Suppose I am trying to use this packing of threads to define a 2D matrix. So I have an 8×8 matrix right and I use that packing of threads to define the matrix. How does that work?

Well as we can see that I am actually having blocks of size 16 here and overall I have 4 blocks. So the total number of threads that can be packed inside this grid of blocks is 4×16 which is 64.

And here I am trying to define a 2D matrix which is in total going to have 8×8 i.e 64 elements. So I can use this arrangement of threads for storing all the values of the matrix or maybe computing the value of the matrix. So let us say see that how its indexes can be derived for each thread.

(Refer Slide Time: 22:05)

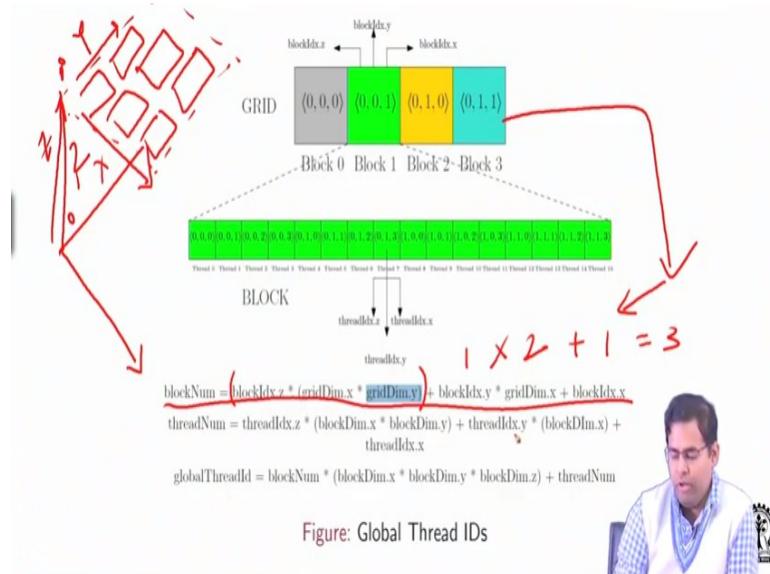


Figure: Global Thread IDs

So first of all for representation purpose we just color coat these blocks in different colors - these 4 blocks in different colors and we just focus on the green block here. We see for one of the specific threads here the `threadIdx.x` parameters are 0,1,3 right and `blockIdx` parameters are also 0, 0, 1. Now just to understand, inside this entire collection of the locations of the 2D matrix, how can this thread figure out exactly what location does it correspond to?

So for that you can see that how these different parameters or the system variables get linked up and how those parameters can be used to compute a position of the thread with respect to the overall collection of all threads. Now for doing that, we observe that there are a few relations that are always going to be true. So what is that? What is the current block number for a given thread?

So I can find that out by evaluating this kind of expression that for this thread, the block number is given by the block id in z dimension multiplied by the dimension of the grid in x and y dimension. Because when I am talking about the block id in z dimension, that means let the value be 2 that means I have already come across a collection of these many that is grid dimension x times the dimensions y number of blocks for 2 values of `blockIdx.z` right.

After that you will have to cover whatever is there in the y dimension so that is we will do a block id x dot y times grid dimension dot x and finally after that we have to enumerate ok what is the remaining number of block id x dot x and that would give you the exact number of the block. So I hope this is clear like we are trying to figure okay given this parameters how do I find out in which block I am right now positioned how can each thread find out its corresponding block number out of this blocks.

So in this representative example as you can see you have 4 blocks for this specific thread which is in block 1 how can I figure out its block number? That the block number is indeed 1? How can we do that? So you can apply this system of equations and try to evaluate. What do you really get here? You are going to get that the block dimension in the z axis is 0. So this part does not work. The block dimension in the y axis is also 0.

So this part of the expression also does not come into play. So you are only concerned about `blockIdx.x` and that gives you 1. Similarly for example, suppose I am talking about block 3 and we are talking about 1 thread in block 3 and how does it figure out that okay it is indeed in block 3. So for this thread the `blockIdx.z` is 0. So this part of the expression cancels out. `blockIdx.y` is 1 So that would get multiplied by the dimension of the grid in the x axis which is 2. Now why is that?

Now let us just review back so as you can see the dimension of the grid in the x axis is 2 and on the y axis is also 2 right. So since this is 1 and this is 0, this part is completely canceling out and the next part for that I get the block id as 1 gets multiplied by the grid dimension which is 2 plus the `blockIdx.x` that is 1. So overall you have $2 \times 1 + 1$ i.e. 3. So just to work it out here.

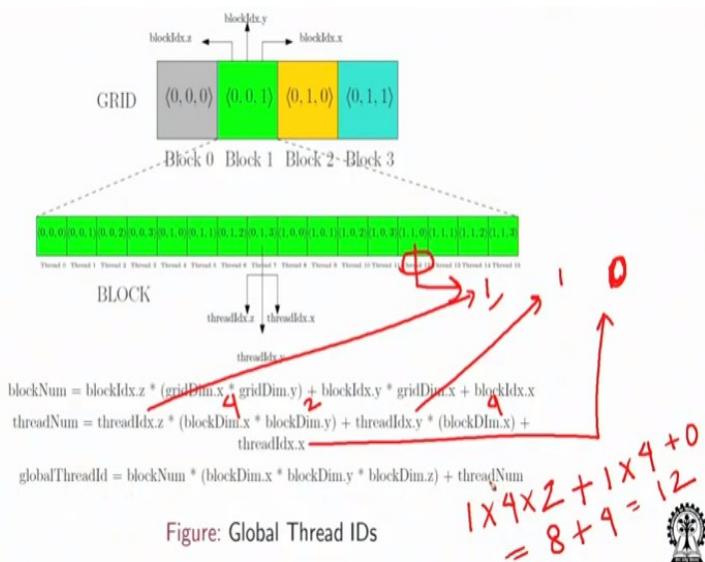
So we will just see that if I am speaking of this block, this value comes as 1 times the grid dimension 2 plus 1 which would be 3 right. So just to explain this expression again for example

in here, we do not have really a case for the first part. How would that have come? Suppose I have got the grid in the actual 3D space. So then you would really be counting first collection of blocks in the z dimension. So you will have some number of blocks in the x dimension in that way.

So suppose in the z dimension you are sitting at position 2. That would mean you have already come across this kind of 0, 1 and 2 layers of blocks. So your blockIdx in the z dimension would get multiplied by these many blocks. Because this is packing in the x direction and the y direction. That is why the blockIdx.z will get multiplied by the grid dimension in x and y axis and then you come to your current dimension position in the z axis. And then you count the number of blocks that are there in the y dimension. You multiply it by x.

So that exhaust your count by all the blocks located in the x dimension for each blockIdx.x value and then finally you would add the blockIdx.y.

(Refer Slide Time: 28:46)



So we will see that it would get better with in terms of examples, once we see some examples of code here. Now in that way we can compute the number of the blocks. In the given thread how do I really compute the exact number of the thread or the global id of the thread with respect to this entire arrangement. So the thread number would be given by the z index of the thread multiplied by the dimension of the block in the x and y axis.

Because actually when I am talking about z index of the thread then I have exhausted the previous blocks. This many blocks i.e. $\text{blockDim.x} \times \text{blockDim.y}$ number of blocks times this threadIdx.z and then you have to compute this term which is threadIdx.y times the blockDim.x and then you have to just add the dimension value in this dimension. Let us take an example to make things much more clear here.

So for example if you consider this thread so the threadIdx.z value is 1 and so basically this is your threadIdx.z and then your threadIdx.y value is again 1 and your threadIdx.z value is 0. The block dimension in the x axis as you can see for here is 2 block dimension in the y axis is also 2 right and this value is also 2. So overall what do you get is 1 times 2 times 2 + so that is 6.

Now if you so if you look at the position here so we are considering this 1, 1 and 0 so this 1 gets multiplied with the block dimension. I believe the block dimensions let me just check that once Sorry so in the x dimension, it is 4 and then y is 2 and z is 2. Sorry so here we just to replace this entity by a 4 and also just check this is also 4 in x dimension is 4 in y dimension and 2 in z dimension. So now these are fine. So this is the threadIdx in the z dimension, the threadIdx in the y direction and here again the block dimension is 4 so that gives you $8 + 4$ i.e 12 and as you can see here the thread id is indeed 12.

Sorry for this delay here. So overall we can see that if you are given the proper thread indexes then the system variables values get automatically initialized and each thread can compute its block dimension and the thread dimensions and then you can figure what are the block id (blockNum) and thread id (threadNum) value. It can compute which block it is in and inside the block which thread it is in. For example if I pick up this thread, the corresponding block dimension is 0, 0 and 1 if we can easily apply this formula. Only the last term would be useful. it can figure out that the block number is 1.

I think for block number we did not have a big example here the block dimensions really small in value but just map this idea of thread number computation here and you can see really see how the block number computed for a more complex example here. And the important thing would be that okay a thread number can be computed for a position of a thread inside this block and a

thread can also find out what is the number of the block in which it is positioned using these two values.

The thread can find its exact position in the global thread id variable i.e. its exact position among all the threads that have been launched which is nothing but the global thread id. It equals to the number of blocks that have been covered in block 1. So in each block you have got how many threads, i.e. $\text{blockDim.x} * \text{blockDim.y} * \text{blockDim.z}$ many threads plus the thread position in the current block. This gives the thread's global id which we call as the global thread id.

So this is the exact position of the thread or exact index value of the thread among all the threads that have been launched by the kernel. Well I think with this we should be ending the lecture here and thank you. We will explain more on this in the next lecture thanks.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 03
Lecture No # 14
Multi-dimensional mapping of dataspace; Synchronization (Contd.)

Hi. Welcome to the lecture on multi-dimensional mapping part 2. So in the last lecture we have been discussing for a given thread how to compute the global thread id, how to compute I) first its block id i.e. it's position in terms of the exact block where it is, II) then the position of the thread inside this block and III) then use this block number and position of the thread values to compute the global thread id i.e. the exact position or index of the thread among the entire packing of threads that is there.

(Refer Slide Time: 01:07)

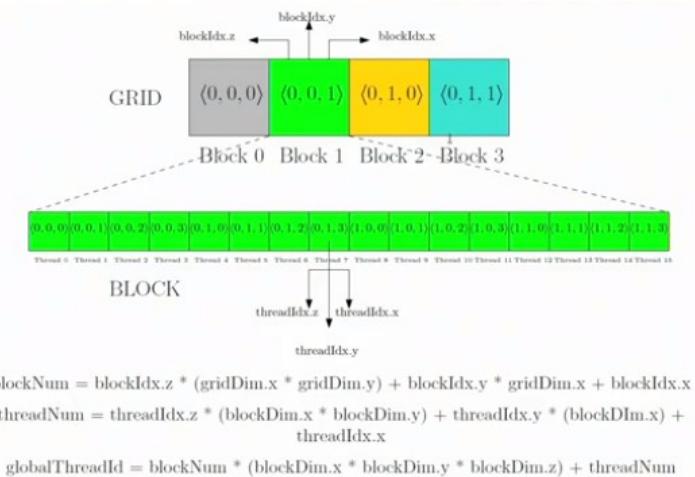


Figure: Global Thread IDs

So this was the example we have been using. Just to recall what we are really trying to do.

(Refer Slide Time: 01:18)

	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7
Row 0								
Row 1								
Row 2								
Row 3						0		
Row 4								
Row 5								
Row 6								
Row 7								

Figure: 2D Matrix



(Refer Slide Time: 01:27)

Multi dimensional grid, block declaration

Consider the following host side code

```
dim3 X(2, 2, 1);
dim3 Y(4, 2, 2);
vecAddKernel<<<X, Y>>>(..);
```

The memory layout thus created in device when the kernel is launched is shown next

We are trying to here design a 2D matrix space and for some reason we are trying to define this 2D matrix space using a collection of threads which are arranged in 3 dimension with respect to the thread block and we have 4 blocks arranged in 2 dimensions. Here in our example. We have colored the blocks in different ways.

(Refer Slide Time: 01:42)

Relations among variables

```
blockNum = blockIdx.z * (gridDim.x * blockDim.y) + blockIdx.y * blockDim.x +
blockIdx.x;
threadNum = threadIdx.z * (blockDim.x * blockDim.y) + threadIdx.y * (blockDim.
x) + threadIdx.x;
globalThreadId = blockNum * (blockDim.x * blockDim.y * blockDim.z) + threadNum
;
```

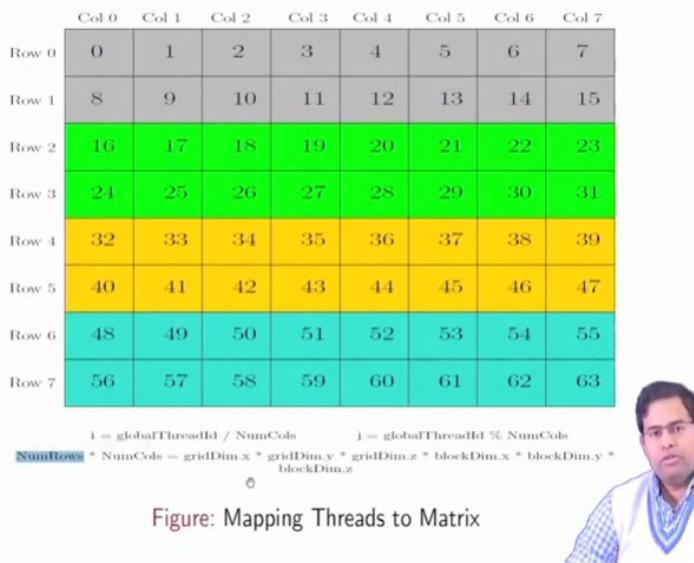
And now we can understand that given this relations, each thread can compute its exact position. This is the relation among the variables that we have already discussed and so inside your program you can have this kind of program statements which a compute thread i.e the CUDA kernel can figure out using this kind of program statements what the global id of the thread is.

So what is very important here is when the kernel is launched and it is executing you have as we have discussed earlier you have single instruction multiple thread right the SIMT model of computation. So that is the same piece of code i.e. kernel and that piece of code is getting executed by all the threads whose packing has been decided by the launch parameters. So using this kind of expression, every thread can compute a unique global thread id.

This is the most important thing with respect to distinguishing the thread with respect to other threads. For every thread, I have the unique combination of block ids and thread ids using which it can compute a unique global thread id. Now this is very important because this is the very parallel program right. Every thread needs to find out what is the unique work it is going to do i.e. what is the unique position in the memory from which it is going to fetch operands what is the unique position of the memory where it is going to store operands based on whatever computation it is going to do that is specified in the code.

So for getting these unique positions in the memory or in terms of some position in a specific data structure I would say for with respect to the programing parlance, the threads need the unique id to be computed and that is basically the global thread id.

(Refer Slide Time: 03:39)



So as we can see if we now apply a earlier discussion of threads and blocks and as we have been discussing that we are coloring the blocks in 4 different colors and each block is now containing these 16 threads, then if we map a 2D matrix into this collections of threads and blocks then this is how they would look like. So as you can see, I have this 4 blocks. So of course this is the 2D picture ,but technically as we know that since CUDA is a C based programming language and the memory access pattern has to be extremely sequential. So finally the arrangement of this locations in this memory is also sequential.

So these locations would be indexed extremely sequentially from 0 upto 63 locations. The first 16 locations would be consumed by block 0 i.e. all the threads with those id's mapping to block 0 and next 16 to block 1, block 2 and block 3 etc. So for every thread here, suppose I am trying to do some matrix operation. So then the thread as to figure out using its combination of thread id and block id as well as grid dimension ,block dimension variables that what is the corresponding position on which it is going to work with respect to the matrix arrangement right.

That means it has to figure out what is the row column index combination for which it really corresponds. We are assuming here that we have launched 64 threads each of the threads are

going to do something about a unique locations of the matrix. So let us say, we are trying to say that the thread which is of id 18 is going to do something about the data located at the second row and the second column.

First of all threads the need to be map itself back to this position that yes I am thread who is suppose to do something about the data located at the second row and second column position, which would mean it has to figure out the corresponding these values. The i and j values that is the row and column values here right. So how can it do that? First of all the thread would use the access expressions which we have discussed earlier i.e how a thread can really compute its thread number, how a thread can compute its block number. It can use these expressions to deliver what is global thread id i.e. what is its exact position among all the collections of threads. So using those relations, this thread should be first able to compute this value 18 i.e. among these global positions which are ranging from 0 to 63, I am the 18th thread. Now it has to use this number and the other values that is the matrix dimension values to compute what is the row and value column on which I am going to work.

It can figure that out just by dividing the global thread id with the number of columns. So if I just divide it by number of columns, then definitely I would get that I correspond to the second row here. And then again if I do a percentile operation, that would give me that what is the column at which I am located which is obvious because as we can see that this percentile with 8 is giving me 0.

So I am located in this column 0, column 1, column 2 like that. So in this way I can keep on doing a percentile and can figure out what column I am corresponding to. I can divide by the total number of columns and I can find out what is the row I am corresponding to. So for 18 I will be able to apply this method computing 18 divide by 8 and of course we are not worried about the remainder. So that gives me a quotient of 2 here so i know that the row number is 2. And of course 18 percentile here with 8 would give me a remainder of 2. So I know that the column number is also 2.

So in that way the thread is able to discover which is the location in the matrix for which it is going to do something over right. Now of course we can see since this is that there is 2D

arrangement another relationship should also hold that I have the total number threads which is given by this expression $\text{NumRows} * \text{NumCols}$, i.e. the dimension of the entire grid that is $\text{gridDim.x} * \text{gridDim.y} * \text{gridDim.z}$ which is the total number of blocks multiplied by the total number of blocks inside the thread that is the block dimension x, y and z dimensions - $\text{blockDim.x} * \text{blockDim.y} * \text{blockDim.z}$ i.e. this gives me the total number of threads which should also be equal to that total number of rows multiplied by total number of columns. So overall I have 2 different ways to find out what is the total number of threads in this case.

(Refer Slide Time: 09:01)

Mapping between kernels and data

The CUDA programming interface provides support for mapping kernels of any dimension (upto 3) to data of any dimension

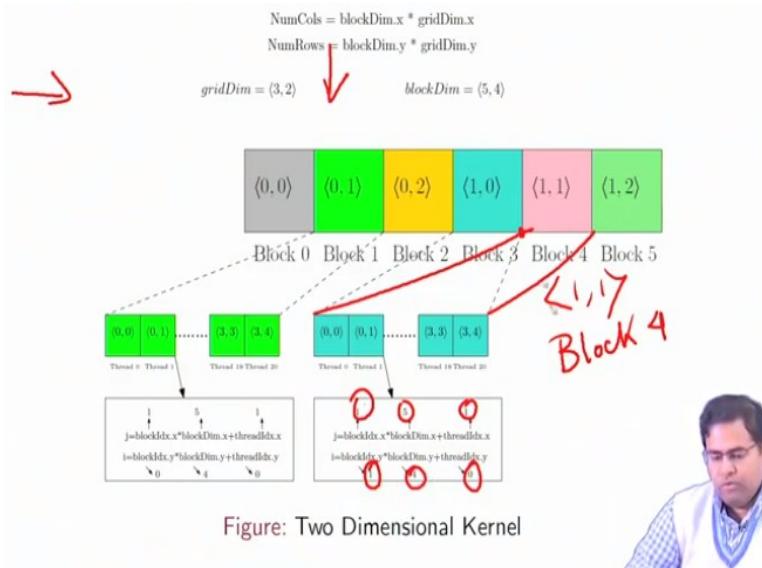
- ▶ Mapping a 3D kernel to 2D kernel results in complex memory access expressions.
- ▶ Makes sense to map 2D kernel to 2D data and 3D kernel to 3D data

Now in general so this is just an example situation. We use this kind of a 2D representation of the data and a matrix example and we use this definition of blocks and grid to figure out that okay if I launch the threads using this kind of launch parameters, then with respect to the actual data collection which may be in 2D space, how can each thread compute what is the data location for which it is going to do some work

But typically how would you like to go about it? So the CUDA programming interface is going to provide you support for mapping kernel in any dimensions to data of any dimension right. You can but of course you are limited by the things that your kernels the number of blocks has to be limited to 3 dimensions and inside the block your thread packing is also limited to 3 dimensions and inside this I can always vary 0, 1 and 2. I can only have the x dimension. I can have 2D packing of blocks with 1D packing of threads, 1D packing of blocks 2D packing of threads All those combinations can come in. But how do you really want to pack the indexes

depends on your choice of the problem. Suppose, you are trying to map a 2D data space using a 3D kernel. That is a bad decision to make, because then what will happen is that your memory access expression are going to be complex. However, it makes sense that if you have a 2D data space, the actual data you are talking about is a 2 dimensional matrix, then you set your launch parameters in such a way that your kernel is fundamentally 2D i.e. you have 2 dimensional blocks and also 2 dimensional arrangement of blocks in the grid. Similarly if your data space is 3D, it may help to design your access expressions of the memory with respect to the global thread ids in such a way that you have a 3D kernel which is mapping nicely to the 3D data.

(Refer Slide Time: 12:03)



Okay this is the small correction here that we have now proceeding further. So in this 2 dimensional kernel example, we will take another different case here. We have already seen what should be a definitive guide line if we are talking about really mapping a 2 dimensional space. We should be using 2 dimensional kernel definitions right. So let us define a grid which is 2 dimensional. This (gridDim) is the dimension of the grid right i.e you are talking about in x dimension we have the blockIdx.x changing from 0, 1 to 2 and in y dimension you are going to have block id values i.e. blockIdx.y values 0 to 1.

And inside each block you have 2D packing of a total of 20 threads (5 * 4). Then this is how your blocks would be arranged. Again we are using a color coding to kind of represent the different blocks. So I am going to have 6 blocks. The issue is see how things change with this

dimensional coding. Since the grid is defined with index (3, 2) so your block id is going to change from (0, 0) upto (1, 2) right.

So you have (0,0), (0,1), (0,2) , since you are going to spread the blockIdx.x variable from 0 to the index 2 and then again you would be increasing the value in the y direction, and then again the blockIdx.y would be 1 and again your blockIdx.x value would range from 0 upto 2. So in that way, you have got this arrangement of 6 blocks. Now if you pick up one example block e.g. this block 1, we can see that since the block dimensions have been defined here as (5, 4) i.e. . a total of 20 threads this (5, 4) will actually define how the threads are going to be indexed.

So since this is (5, 4), the threads would be indexed from (0,0), (0,1) ,.....,(3, 4). So here this is basically the the x dimension and this is the y dimension. Since this is the x dimension, we understand the threadIdx should range from 0 to 4 and since this is the y dimension, the value is 4. So threadIdx.y is going to range from 0 to 3. So overall I would have thread (0,0), thread (0,1) and the final thread with values (3, 4). In total we have 20 threads. I hope this is clear.

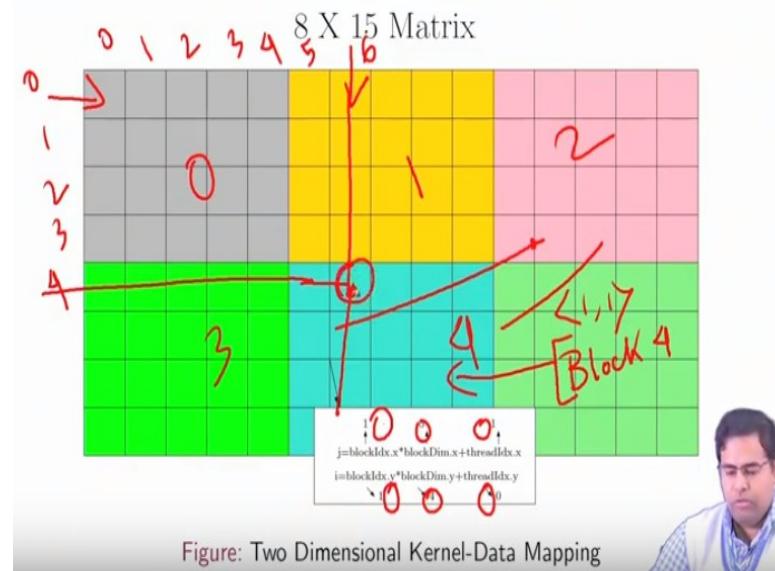
So again, I would just repeat the x dimension comes first, then the y dimension whereas when I am just writing them in order the x dimension is increasing first and then the y dimension is decreasing. I mean that is how you would like to remember it here. So now if I am looking for a specific thread here let us say this thread (0,1). So again let us do a recollection of what would be the thread id and other values.

Since this is a 2 dimensional kernel, I am doing a 2 dimensional kernel mapping to 2 dimensional data space. I am trying to pick over the i and j values i.e. the row and column in the index values here. So the j value is given by the $\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$ which is the thead id in the x direction right. So what are those individual values? As you can see the block id for this is basically 0, 1. So this is the blockIdx.x value i.e. 1

This blockIdx value is not required here. The block dimension is again in the x direction. It is 5 so this is the block dimension so this is 5 here the block id x dot x value is 1 because of this right and what about the thread id x dot x value. So the thread id x dot x value is 1 because here this is 1 and thread id x dot y is 0 which is not again required here. So this is how it is right so using this i can compute what is this j index?

So that gives me the column index here right similarly I can also compute the row index you continuing a 2 dimensional data layout here.

(Refer Slide Time: 16:46)



So coming back here the column is so fine here if we just check it for this specific example only 151 so if you just compute here what do you really get so you have what this. Since that is 6 and also let us do the same for the I value as you can see the block id x dot y not that is we are talking about block 1 here. So block id x dot y is 0 this 0 so in the block dimension in the y direction is this right plus the thread id x in the y direction.

So what is the thread id x in the y direction here? So again you have 0 here. This would map to this $i=0$ the first row. This (threadIdx.x) was computed as 6 so your column index would be 0, 1, 2, 3, 4, 5 so that would be here. Now coming back to for our example, if we pick up some alternate values.

For example let us pick the some entity from block 3 here. Block 3 has as you can see indices 0, 1, 2 and then 3. So the blockIdx.y is now becoming 2 (small correction). So ideally since, blockIdx.x is also 1 and the blockIdx.y is also 1. Essentially we are speaking of in this figure here block 4 right. So then inside block 4, if I pick up for a specific thread, say the first thread a with thread id 1.

So for thread 1 inside block 4, as you can see the thread id starts from the zeroth thread i.e. thread 0 is 00 and thread 1 is 01. So the threadIdx.x is 1, threadIdx.y is 0 and the since this is the fourth block i.e. among the zeroth, first, second, then third block is 10 and fourth block is 11. So I have the blockIdx.x and blockIdx.y both as 1. Then I have the block dimension here right and those we have already defined earlier in the blockDim.x which is 5 and blockDim.y which is 4 right.

So that is what you have. So with this finally I can say that okay here what is the position I am talking about. So j would give me the column position and that is you have $1 * 5 + 1 = 6$ and so the sixth column and which row do you have? It is the fourth row $1 * 4$ and that is what we have. So row 0, 1, 2, 3, 4 so you have the fourth row and then so in terms of the column is sixth.

So 0, 1, 2, 3, 4, 5, 6 right so it should be this position here. So as you can see you have this blocks. You are now talking about the fifth block so the block id's are (1,1) and inside this block you have the block dimension coming to 3 this block dimension for each block you have the block dimension which is 5. So $1 * 5 + 1$ is your j . And if we speak about the block id here in terms of the rows, you are moving block dimension in the y dimension when you are computing the row. So you are in blockIdx.y = 1. That would automatically get multiplied by 4. So then you have already covered the first four rows and then you have your thread id 0. So essentially you will be in the first row of the fourth block here. That is essentially if you compute the rows from 0 zeroth, first second, third, fourth, this is the position you would be talking about.

So just ignore this arrow. If you are talking about block which is the fourth block, so this is the zeroth block first block, second block, third block, fourth block given by (1, 1) and for this block I have got the threadIdx.x value as 1 and threadIdx.y as 0. And with this I am able to compute this is the position which I am talking about. This is the location I am talking about. So in this way you can see that if you are trying to define for a 2D matrix, you are trying to use a definition of grids and blocks in the 2D i.e. you have a 2 dimensional packing of blocks and you also have a 2 dimensional packing of the threads inside the block. So this is the 2 dimensional packing of blocks, this is the 2 dimensional packing of grids inside the block. It nicely maps here to specific locations. And now let us take an example of a 3 dimensional mapping of a kernel on a 3 dimensional data space. So since we are trying to talk about a 3 dimensional mapping but again I

would again repeat one simple thing that why at all we went for this. Because of our initial hypothesis that was a 3D mapping a 3D kernel maps nicely to a 3 data space and a 2D kernel maps nicely to a 2D data space.

So since as we can see that our hypothesis is quite validated because now the excess expression since for this thread which is with the block id 1, 1 and thread id 1, 0 I am able to execute this 2 expressions and compute corresponding i and j values and does not the expression look really nice and simple? Because the j value which gives me the column index, I have a very nice expression because block id multiplied by block dimension plus thread id everything is in the x axis . Similarly for the row index block id multiplied by block dimension plus the thread id again everything in the y axis right. So this feature as you can see it is quite intuitive and it is easy to map because since I have a 2D arrangement of threads and the threads are going to compute on a 2 dimensional data space i.e. inside the data structure, the definition itself is a 2 dimensional arrangements.

So once I map the collection of threads in 2 dimension the access expression for the row and the column indexes automatically map dimension wise. This is the most important thing that we are really caring about here. Since the data structure is having a specific dimensional arrangement and we are packing the threads in an identical dimension arrangement, I have got this kind of very simple access expressions which only involves the x axis variables or only involving the y axis variables. Now we will be soon seeing that this idea maps for more complex data types also

(Refer Slide Time: 28:02)

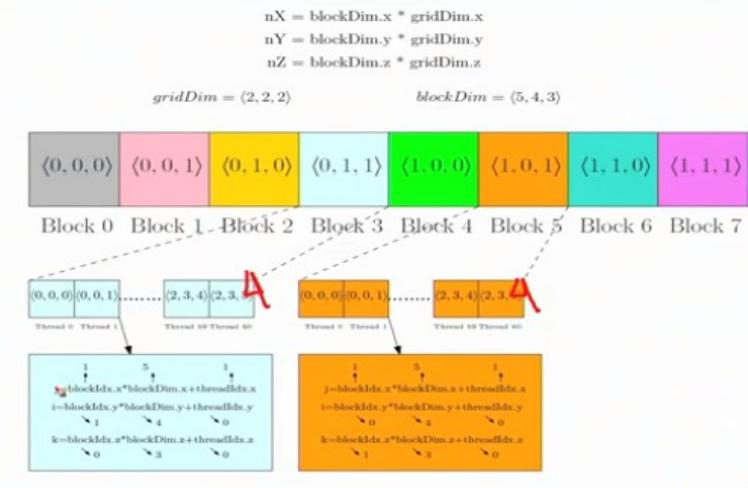


Figure: Three Dimensional Kernel

For example let us talk about the 3 dimensional kernel which is going to work on a 3D matrix. So for this 3 dimensional kernel let us consider 8 blocks so your grid dimensional is (2, 2, 2) arrange in 3D and you have each block containing 60 threads they are mapped in 3 dimensional as 5 x 4 x 3. And so we are having these 8 blocks with their indexes ordered just like we gave the earlier examples.

If you pick up 1 block, inside that again you have this collection of 60 threads with id's from all 0 to 2, 3, 4, 5 Why? Because you have the last thread we will have 2, 1 less than this (3) one less than this (4) one less than this (5). So this should be 4 here Sorry, let me just put a correction here. So last index has to be 4 and so again when I am trying to compute, say pick up 1 thread. For this thread, when I am trying to compute that exactly in which location in the 3D matrix does this thread corresponds to, I can find out the i , j and k . Because this is now a 3D matrix. The positions for the location in the 3D matrix corresponding to this thread. I am saying this thread will compute something for this location in the 3D memory. Again the access expressions are all in x direction i.e. $\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$. Similarly all in y dimension and all in z dimension.

(Refer Slide Time: 30:06)

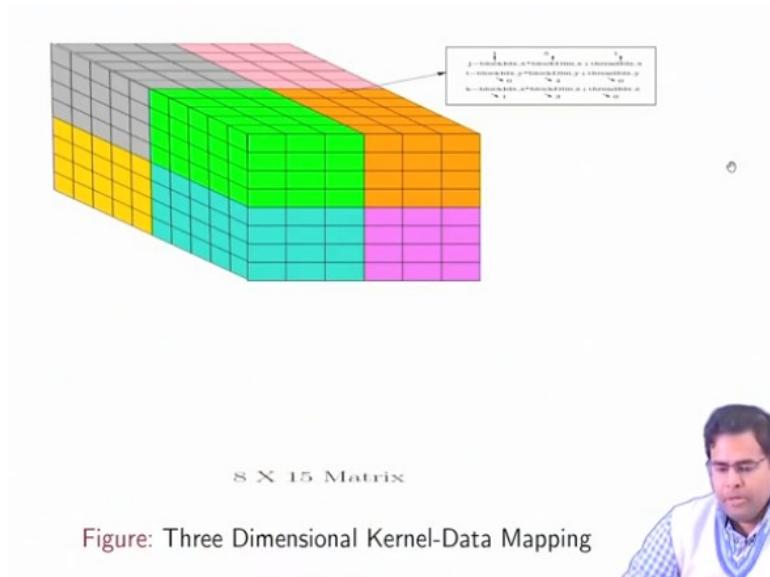


Figure: Three Dimensional Kernel-Data Mapping



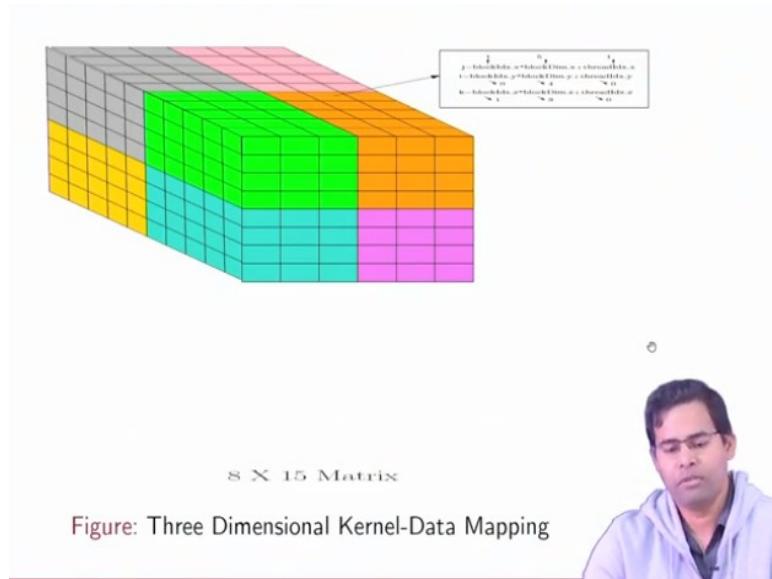
This is how the arrangement would look for a 3D collection here. So you are having 8 blocks now. You have a 2 dimension definition of 2 in the x axis. In the x side you have blocks with values ranging from 0 to 1. Similarly for the y axis and similarly for the z axis. So you have in total 8 blocks and inside each block as we defined that you have got 5 threads packed in the x dimension, 4 threads for the y dimension and 3 for the z dimension.

So you can see you have such arrangements here. In 1 dimension it is a collection of 5, in 1 dimension it is a collection of 3 and in another dimension it is the collection of 4. So this is how you can say that a collection of threads map to a 2D or 3D data space. In general I can have for a complex application, a more complex data structure considering high dimensional data and I can look at dimensionality of the data and accordingly I should be able to decide a suitable dimension of my blocks i.e. how to pack the threads inside the blocks in a multidimensional way and how to pack the blocks inside the grid again in a proper multi-dimensional way, so that I can design a simple expression which can access the location inside the actual data structure and compute using the threads. We will see more into that in future that how this can also create some problems with respect to parallel computation and collaboration among multiple threads. For now let us stop here thank you for listening.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 03
Lecture No # 15
Multi-dimensional mapping of dataspace; Synchronization (Contd.)

(Refer Slide Time: 00:27)



In the last lecture, we have been discussing about multidimensional mapping of data spaces like how to choose suitable ways to map high dimensional data in terms of the kernels, like how to choose a suitable access expression for the different data segments. So those were just some initial thoughts we discussed and we saw that what is advantageous in each case and of course this would be more clear in future when we discuss more examples and provide more assignments with respect to different kind of handling of different kind of high dimensional data spaces and creating what we call as access expressions. That means how does a thread really identify what is the memory element it is going to work on. And just to recap, this was based on a threads knowledge of its block id thread id variables it being able to use those variables to create suitable access expression for different data points it was going to work on.

And as we saw that it is very much a function of how do you define the grid and the blocks that means whether you define a 2D block or 3D block and a 2D grid or 3D grid and accordingly

your access expressions are subject to change. So this is something you can try with more examples and we will also try to show that with examples in terms of assignments later on.

(Refer Slide Time: 01:55)

Synchronization

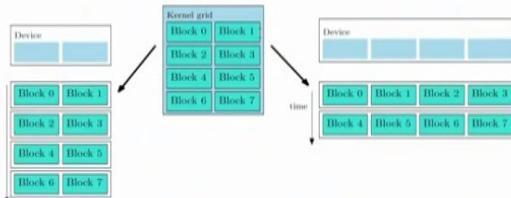


Figure: Mapping Blocks to Hardware



- ▶ Each block can execute in any order relative to other blocks.
- ▶ Lack of synchronization constraints between blocks enables scalability.

Now we will move on to another important topic of synchronization among threads. Now first of all what is synchronization? In general with respect to parallel programming semantics synchronization essentially means some points in the program where we definitely know that okay this is the point where the threads have computed upto and once this point of computation is reached, this is where every thread should reach and together. Or if some thread reaches this point faster and then some other thread reaches this point and the thread which is this point faster should wait for the other threads to come up to this point and then again they can go forward.

So it is basically a primitive. That is why we call it as synchronization point because that is where every thread should stop. Maybe they can collaborate together. They can ensure some sanctity of reads and writes on the data variables and then they can move forward with further computation. I mean of course again this is a high level idea which can only be clarified through some major examples that we will touch upon.

So let us try and understand how synchronization can really be enforced on different threads which are computing in a CUDA program. So as we know that the kernel launches a grid of threads. The grid contains a set of blocks and these blocks contain threads packed internally and these blocks containing are actually getting scheduled inside the SMs.

Now each block can execute in any order relative to other blocks. This is a very important thing. So if you take an example that suppose you have got a device (GPU) and you have got 2 SMs right and may be in SM 1, you have got the blocks 0, 2, 4, 6 scheduled and in SM2 you have blocks 1, 3, 5, 7 scheduled. So overall your kernel has launched this many 0 to 7 block id blocks.

If you have the 2 devices (SMs) there can one there can be possible way in which these devices are processing the blocks. Of course this is not a unique way there can be other ways. Like for example suppose I have 4 devices, I can have a mapping where block 0 and block 4 are mapped to device 0 where device means the SM's. So there can be various possible mappings of blocks in the SM's. Each block can execute in any order relative to other blocks.

That means once I have launched all the threads in a grid there is no relative specific execution order like which thread will execute faster with respect to some other thread. I do not have any control over that, unless I put in some extra primitive. So every block of thread can execute at its own pace. That is the point that we are trying to decide here. Essentially the GPU has got a complex scheduling hardware. There is a 2 level scheduling hardware and it depends very much on how the hardware is going to schedule the blocks. There is some high level knowledge available in open domain about that.

However, every detail is not available till now in open domain. Is very much part of the implementation of the GPU. But what is known in the open domain is that there is a 2 level scheduler. So at some levels, it is decided which blocks are going to be part of which SM and inside the SM's we have the scheduler to decide how to execute the blocks. We will get into more details later on. But for that time being, let us understand that as a programmer you do not have any control over which blocks executes in which order.

You have launched the kernel that would initiate all those set of blocks, that will initiate this entire grid. Then it is part of the architecture's hardware scheduler job to execute them in any order. However certain scheduling notions gets enforced which we will discuss. So the important take away from this slide is that the block can execute in any order relative to other blocks.

And there is lack of synchronization constraint between blocks. That is a good thing because it enables scalability. So as a programmer you cannot enforce synchronization across blocks of code. That means I cannot enforce the block 0 will execute only let us say after block 5 executes or similarly blocks 6 executes only after block 1 executes. I cannot enforce that. This actually is considered a good thing with respect to the hardware, because since this cannot be enforced unless you do some fancy programming.

Again I would repeat if you write a normal vanilla CUDA program you cannot enforce such a thing and so the hardware is at full freedom to execute the blocks depending on the amount of SM or SP that are available that are free. It can schedule them with full freedom and actually provide the maximum throughput that is possible. That is the good thing about not synchronizing across blocks.

(Refer Slide Time: 07:13)

Synchronization

- ▶ Synchronization constraints can be enforced to threads inside a thread block.
- ▶ Threads may co-operate with each other and share data with the help of local memory (more on this later)
- ▶ CUDA construct `__syncthreads()` is used for enforcing synchronization.



However since that is the question, it is of course understood that synchronization can be done among threads blocks inside a thread blocks. That is the very point we are trying to roll in here. So when I am launching the kernel, all this thread blocks are getting launched. They will get mapped to the individual SM's and inside the SM's the order in which the blocks execute or across SM's the order in which blocks execute is beyond the programmer's control.

So I cannot really enforce a synchronization constant among the blocks. However as a programmer what I can do is I can enforce synchronization constants on threads inside the thread

blocks. So inside a thread block, the threads may cooperate with each other and share data with the help of local memory, more on this which we will see later on. First of all let us understand how synchronization among threads can be enforced. So for that you have the CUDA constants `_syncthreads()` which is used for enforcing synchronization Let us see some examples on that.

(Refer Slide Time: 08:21)

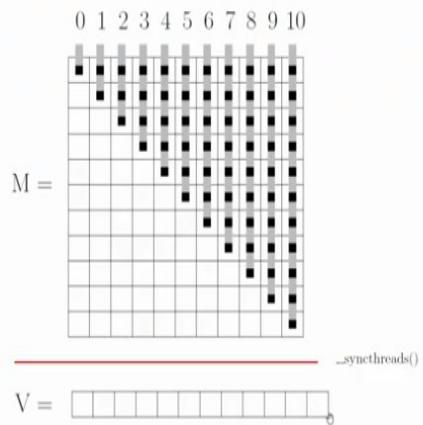


Figure: Input: A 11×11 matrix, Output: A vector of size 12 where each element represents the column sums and the last element represents the sum of the column sums.



For example let us take this sample computation here your input is an 11×11 matrix and you want to output a vector of size 12 where each elements represents the columns sums and the last element represents the sum of all the column sums. So you want the threads here to compute. I mean it should not compute anything. It just reports the value for column 0 here, thread with ID 1 would do a partial sum of thread of this position and the next position.

Basically the position basically the entries in $M[0][1]$ and $M[1][1]$ - the sum of these 2 locations that it should compute and put in here. Thread with id 2 should compute this sum and put it in here and like that. And finally somebody would be actually computing the sum of this entire array and putting it to this location right. So this is what we want to do.

So now we need to understand that why synchronization would be necessary there synchronization would be necessary because we want all the threads to execute the same kernel code. However the amount of partial sum of computation that each thread is going to do is different right. Because if I pick up this thread, the one I am marking here (4) if I pick up this thread I want it to sum only up to this point $M[4][4]$.

So as you can see that although these threads are launched with different ids, the amount of work you want the kernel to make them do is different. That would also mean each of the threads are going to finish their execution at difference points of time. Now if these threads are going to be inside a single thread block, which is the case here, then I need to make the thread which computes it is job earlier, I need to make it wait for the other threads which are going to complete their jobs later on.

Why is that so? Because unless all these intermediate values are ready, I cannot ask some thread to do a summation of all the entries here. Because I want this thread (2) to compute the partial sum of these points and put in here ($V[2]$). I want this thread (4) to compute the sum only upto this point ($M[4][4]$) and then put it here ($V[4]$) like that and only when this is done then I can go and do the computation of summation of all these values and put it here ($V[11]$).

Which means I need to split this computation across different phases. In phase 1, all the partial sum computations should be done and these results should be updated only after that. I would like to have a sink. So that I have to enforce. For that I would like to put this `__syncthreads()` here. Only after the phase 1 is completed for all the participating threads, should some thread go on and work with these values. So if I do not do a `__syncthread()` here, something random can happen. We will see that.

(Refer Slide Time: 11:40)

Synchronization Host Program

```
int main()
{
    int N=11;
    int size_M=N*N;
    int size_V=N+1;

    cudaMemcpy(d_M,M,size_M*sizeof(float),
    cudaMemcpyHostToDevice);
    cudaMemcpy(d_V, V, size_V*sizeof(float),
    cudaMemcpyHostToDevice);
    dim3 grid(1,1,1);
    dim3 block(11,1,1);
    sumTriangle<<<grid,block>>>(d_M,d_V,N);
    cudaMemcpy(V,d_V,size_V*sizeof(float),
    cudaMemcpyDeviceToHost);

}
```

So for example let us look at the code. So this is the host side program for synchronization. So you have 2 cudaMemcpy commands using which essentially you are transferring host to the device, the 2D array M and the 1D array V which is going to be our result. And from the dim3 primitives, you can see for grid and block, you are just launching a single blocks because as you see these entries are all 1 in the grid declaration.

And in the block declaration you can see that you are just launching these 11 threads right. So it is 1 dimensional block and that is the only block that you are launching. With this launch parameters you are launching the kernel sumTriangle and this kernel is going to work on the device side array d_M and d_V. N is the size of the input matrix in each dimension and once this kernel execution is done you want to copy back the content of d_V to the host side 1D array V. So this is your simple host code here. Now let us look at the kernel code.

(Refer Slide Time: 13:00)

Kernel

```
__global__
void sumTriangle(float* M, float* V, int N){

    int j=threadIdx.x;
    float sum=0.0;
    for (int i=0;i<j;i++)
        sum+=M[i*N+j];

    V[j]=sum;
    __syncthreads();
```

So the kernel code is going to tell you what is the per thread activity. Let us have a look what is per thread activity. So the first thing you will do is you will consider what is the thread id. So in this case, since I have just a single block and that block is also a 1D block, I just have to care about the threadIdx.x variable and this variable is going to give me the column of 2D array on which I am going to operate .

So let us pick a sample value of j, let us say I am going to talk about the column with id 5 so that means I am talking about the thread with id 5 starting from 0 right. So now all that is going to

happen is if you look at this picture, I am talking about this thread (5) so it is going to sum up 0, 1, 2, 3, 4, 5 these 6 values starting from 0. So you compute the sum of the quantities here which you access through this access expression M. i is the row number from 0 you go upto $i \leq j$ so you go upto $j - 1$ and so you complete that many rows and then come to the j th column. So in that way with this expression if you look at this expression $i * N + j$ for different values of j you are able to access these different locations. Because fundamentally this is all sequentially located. So with that kind of an access expression, you are able to sum up all those elements in the column with id j . But that is also up to first j elements. That is what you are summing from the zeroth through $j - 1$ i.e. j elements.

So once this is done you store this sum. This thread will store this sum in the location $V[j]$ of the output array. So that means once this computation is done it will store at this location of the output array. After that we have a `__syncthreads()` which we are discussing. Once every thread has computed these partial sums and they have stored these results in the output array, there is a sync thread which means every thread stops here. Why?

Because I want to compute the summation across this array and store that sum at this point only when the job of all these different threads are completed right. That is why I will put that `__sync_threads()` after the partial sum computation and storage in V . And before that is summation starts according across V because I want to ensure that before I do the summation across the entities of V all the entries of V are ready right.

So that is why after computing this V_j in 3 I put the sync thread once all this entries are ready then I get into to ask that one of the threads that okay now you compute the sum across the kernel. So once each thread finishes computing the sum across columns the total sum is computed by the last thread. So essentially how do I choose which thread is going to do this job as you can see that we have launched this many thread that 11 threads we have launched and the last thread.

(Refer Slide Time: 16:47)

Kernel

```
if(j == N-1)
{
    sum = 0.0;
    for(i=0;i<N;i++)
        sum =sum + V[i];
    V[N] = sum;
}
```

Once each thread finishes computing sum across columns, the total sum is computed by the last thread.



So as you can see we so this is the part thread activity for this kernel and only the thread with ID in -1 would enter into this block right only the thread with if n – 1 will enter into this block and then what this thread is going to do is it is summing up the entries of V and storing it into the location Vn. As you can remember that V has got one more space extra space so V starts from V0 and it is upto VN.

So only the thread whose ID is j would be equal to n – 1 so that is essentially the thread which is computing this entry that will get into that if block the other threads are not get into the if block but only that thread would do the summation and then store it up here. Now the question is then what was the requirement of sync thread. The requirement was there because we want the following to be ensured that when this summation is being done on the output array.

All the VI entries in the output array should be ready now which may happen that you run the code without the synch thread but you still get the result. Question is there is no guarantee that you will really get the result you may get the result may not get the result why because without the sync thread then we are assuming that before the last thread starts doing this computation everybody else has finished it is not guaranteed.

Now the question is when do I really not have the guarantee okay if we are working with small number of threads where the threads are all part of the wall then I can have this guarantee but as we have discussed that in a general setting when I have very large number of threads in a block I have got very large number of threads in a block then I cannot guarantee only when the threads

will be part of a single warps then I know that the threads will execute in lock step and then I have a guarantee that okay when the last thread is going to finish by that time the previous threads which are in the same wrap same warps as actually finished.

Bu tin general that may not be true only in the case when I am having multiple warps which are created inside the block right. So just to highlight your example if you are running this code with small number of threads like this the threads will be part of a warp and you may not need this sync thread. But if you are running this code on a big thread block which is going to launch multiple warps then you do not really have a guarantee because then what is going to happen is each of the warps are going to schedule threads in different ways will come to formulizing that later on.

But just to understand that I am trying to remove a notion here that you may run this code on a small setting of a block and see that well yes without the sync thread still things are fine. I am just trying to say that it may or may not be fine it depends on the size of the block and we will see that exactly in which case it is good and which case it is bad. But in general you need to appreciate that when I am running this threads I do not have any guarantee that this thread is going to finish last or this thread is going to finish first it depends on how many threads are lunching.

If I launch this small number of threads we are talking this small number of threads just for a figure representative purpose it may happen that i do not need the synch thread but for a significantly large number of threads I do not have any guarantee. So anyway from a programmers point of view I need to ensure that all this entries in the v are really ready that means all the threads have really done their job before this last thread goes to compute this entry right.

(Refer Slide Time: 21:18)

Synchronization Program Variant I

Modification: Only elements at odd indices are summed.

```
__global__
void sumTriangle(float* M, float* V, int N){

    int j=blockIdx.x;
    float sum=0.0;
    for (int i=0;i<j;i++)
        if(i%2) // Check for odd indices
            sum+=M[i*N+j];

    V[j]=sum;
    __syncthreads();
```

So for that purpose I will need to put this sync thread here fine so that would about this small program and then let us look at some other variants. So suppose I hope just to recall I hope the idea is clear that I just wanted to put this point across that it may not be the case if we are using a very small number of threads or we are launching a small number of block but in general for a significantly large block and in general unless I put this thin thread here I am just trying to summarize here.

And I do not put this thin thread here then there is no guarantee on how fast each of the threads should compute. So to enforce the constant from the programmers point of view I would like to have the sync thread here fine. So looking at other variance of the program so now let us consider the situation that we only sum up that elements that the odd in this s.

(Refer Slide Time: 22:19)

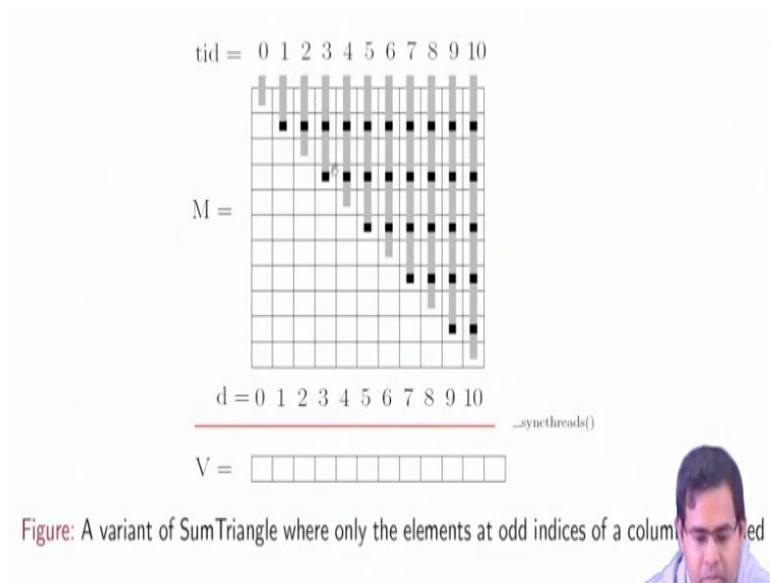


Figure: A variant of SumTriangle where only the elements at odd indices of a column are summed.



So that would mean if I look at the example so I am just summing up the elements at this indices here. So at index zeroth element the third row element have not including the elements are the row number 0 or row number 2 like that right. So again including the first 2 element third first row and third row included here first row and third row is included here first, third and fifth row is included here. Again first, third, fifth and seventh row is included here so we are going in a one hop basis.

If I am trying to do such a variant of the same computation how would really the kernel look like. Well it is simply a matter of putting in a choice statement here so essentially I have the same code so I believe this should be thread id here just a minute sorry for this yes essentially is the same code okay. Let us just review the earlier code once again so this was your original code and if you come here is basically the same but you are doing the sum only for the odd indices.

So you push in a small check if I percentile 2 so if that is 1 that means you get in for the odd index rose and otherwise you do not get in and only in those cases is the sum and in that way everything remains the same right fine.

(Refer Slid Time: 24:19)

Synchronization Program Variant I

Addition still carried out by the last thread.

```
if(j == N)
{
    sum = 0.0;
    for(i=0;i<N;i++)
        sum =sum + V[i];
    V[N+1] = sum;
}
```

So rest of the code is same your addition is still carried by the last thread so it is all the same here. So you have the same thing same setting but as we are explaining every thread is hopping over an element and again we need the synch thread here if I have a significant number of threads all of the threads are not pushed in into a warp. So you do not have any control over which thread is progressing at what speed so after their each of their partial sum computation again I would definitely need the synch thread here to ensure that yes every thread has done the computation properly I mean they are all finished.

And accordingly now we will instruct once every threads reaches this point this is the synchronization point only when they reach here they again restarted for a fresh run. And again the last thread starts summing up this entries to compute the last entry.

(Refer Slide Time: 25:24)

Synchronization Program Variant II

Modification: Consider summing all indices again. But use all threads for final reduction.

```
__global__
void sumTriangle(float* M, float* V, int N){

    int j=threadIdx.x;
    float sum=0.0;
    for (int i=0;i<j;i++)
        sum+=M[i*N+j];

    V[j]=sum;
    __syncthreads();
```

Now consider a different variant of the program again we are summing all indices like the first example like we are not again hopping over the rows we are going to sum all the rows. But now we are trying a modification that instead of using a last thread for the final reduction that means for doing the final summation computation we want to do the final summation computation in a collaborative way by using all the threads.

Why because as you can understand if I give that job to one thread for a significantly large block it may require lot of time but I may want to speed it up by distributing the warp among all the threads because finally why do I leave it for one thread there are many other active threads which are simply sitting there idle right.

(Refer Slide Time: 26:18)

Synchronization Program Variant II

Reduction possible since addition is an associative operation.

```
for(unsigned int s=1;s<N;s*= 2)
{
    if (j %(2*s)==0 && j+s < N)
        V[j]+=V[j+s];
    __syncthreads();
}
}
```

Once each thread finishes computing sum across columns, the total sum is computed by all the threads.



So again the first part of the code is again exactly the same for this piece of code let us first try to understand what is the activity that each of the threads are doing and we can this activity can be actually visualized graphically in a better.

(Refer Slide Time: 26:36)

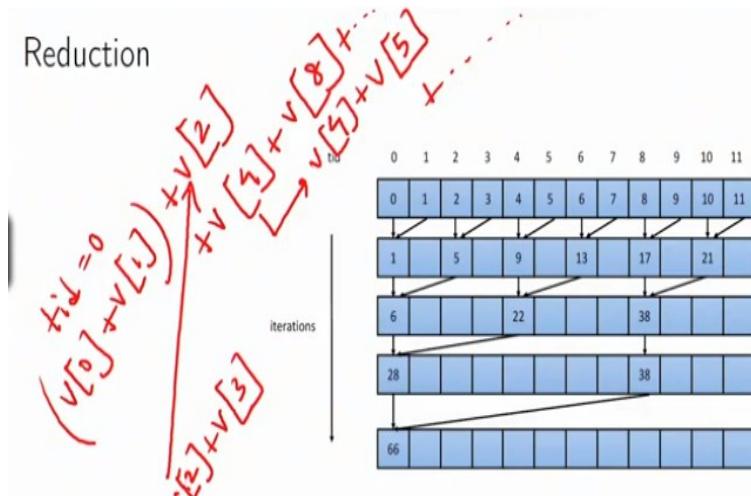


Figure: Reducing an array of 12 elements

So if you look at this example here so we have the thread id's all plotted like this in the X axis and we are showing what each of the threads are showing in each iteration right. So let us just follow 1 or 2 iterations and things will be very clear. So first of all in the first iteration of this loop every thread is going to enter this loop with an $s = 1$ j is the thread id now as you can see in the if condition we have $j + s$ is less than N .

So since j is the thread id and s is 1 so this will be satisfied by all but the last thread right so for all the id 0 to 10 this is fine right is 11 our example here but the first part of the condition is 1 so I am doing essentially a j percentile $2 = 0$. So that is only going to be satisfied by the threads with the even indices right. So all the threads in the even indices is going to do something in the first iteration of the loop and what are they going to do they are simply going to add the entries in the array at the location v_j which $v_j + 1$ right.

So essentially for thread id 0 is going to compute $V_0 + V_1$ and store it in V_0 so that what we have $V_0 + V_1$ and store it in V_0 by thread id 0. Thread id 1 is odd index does not do anything thread id 2 is even index does $V_2 + V_3$ and stores it V_2 and similarly everywhere. So at the end of the first iteration of the loop half of the threads actually collaborated among each other and they have successfully computed the partial sums at tid of 1 is denotes the tid.

So with 1 hop half of the threads are collaboratively computed the partial sums so we can understand how things are going to proceed here. In the next iteration of the loop another half of the threads are going to be active and they are going to increase the tid by double that is $s = 1$ to $s = 2$ and they are going to do the partial sum computation of entries this and this right. So essentially if we go back to the code in the next iteration it gets multiplied by 2.

So we shift from tid 1 to tid of 2 and the thread id 0 will again satisfy the condition but it will get in the loop and compute a $V_0 = V_0 + V_2$ here right. So if you look into the code initially we did $V_0 =$ so if I just look at what thread id 0 is so here it computed $V_0 + V_1$ this value got stored in V_0 and this is how added up with the content of V_2 and then next it will get added up with the content of in the next iteration with before in the next it will get added up with the content of V_8 and in the process the V_2 that is getting.

We already have V_2 to be equal to $V_2 + V_3$ if we look at V_4 I have already have $V_4 + V_5$ and then $V_4 + V_5 +$ actually $V_6 + V_7$ so in that way I have the sum of to V_8 and the other elements will also come in right. So in this way as you can see that the entire computation will make a progress and all the threads collaboratively compute the sum. So just to repeat in the first iteration half of threads are simply adding their own position with the next position.

So in that way I have half of the partial sums stored here in a next iteration I have further half of the threads that is one forth of the threads computing this locations along with the location at a stride of double space that is $S = 2$. In the next iteration we again have a further half of the threads which is actually summing of this location along with the stride at an even double space so $S = 4$ in that way I am able to compute the sum for the entire array.

So you can just write if you are trying to prove the correctness of the program if you can just write at the final location sum is equal to this 2 location sum this is again equal to sum of this 2 locations and this location is again equal to sum of these 2 locations and you can go back like this. And finally you get this expression where this locations is equal to the sum of all the individual locations right.

So that ensures the correctness of the reduction and we can actually prove that this simple optimization gives me the correct result but it actually engages all the half of the threads together in the computation. Unlike engaging 1 thread to do the entire computation. Now observe something very important here after this first iteration I have got all the threads to gives me the value of its own locations data plus the next location right.

Unless all this individual computations are finished none of the thread should be allowed to go to the next iteration because then there will be inconsistency. For example let us say thread 0 will progress to compute this plus this but may be thread 2 has not completed this plus this operation in the previous iteration right that can happen because of the situation. We discussed earlier that in the GPU I do not have a guarantee that threads are progressing together unless they are part of the same world right.

Unless they are part of the same warp threads are not progressing together even if they are part of the same warp based on the conditional execution that warps may diverge and there will be things that we have to discuss that for the timing we can just assume the treads executing inside a block are progressing at their own speed. Since we do not have control over there execution speed because of the hardware scheduling taking care of that I cannot really guarantee that 2 threads doing this partial sum computation are progressing at this same speed.

But we can see that we have the requirement here unless each of the threads are able to do their computation for some specific value of the stride none of the threads should allowed to go to the next level of computation for the next value of the stride I cannot let S progress and all the threads progress to the next iteration of the loop unless all the threads complete the previous iteration of the loop and the way to do that is very simple just put a sync thread inside this loop here.

So once every thread is guarantee to execute one iteration of the loop only then all the threads together go for the next iteration of the loop. So I will just repeat once every thread is guaranteed to compute executing 1 iteration of a loop only then we should let all the threads together progress to the next iteration of the loop and this is how the things should keep on going and the computation would keep on progressing and finally we have all the threads computing together and giving me a final result.

So with this we would like to sum up our explanations on synchronization of course we will have some more examples in future which will involve synchronization but this is just to give you basic introduction of how to and where to put sync thread primitive in the code. So basically we have to understand at what are the point in your kernel where you need to guarantee that all the threads of thread block have actually done their computation before proceeding to the next line of computation.

If there are such algorithmic issues where all the threads are going to use results computed by each other at some point of computation or from starting from that point of computation there as to be sync threads statement for synchronizing the threads inside the stride block. So just to summarize threads inside a thread block can synchronize you have to use the sync thread primitive but the reason here to use it just to sure is that you do not any guarantee that what speed the blocks and the threads inside the blocks are progressing inside the GPU how is the scheduling done. We will investigate more into this from the next lecture thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 04
Lecture No # 16
Warp Scheduling and Divergence

Hi everybody. So in the last lecture we have given a summary of how multi-dimensional data can be processed by a CUDA program and how threads inside a thread block can synchronize amongst each other. While discussing that, we also discussed that in GPU program execution i.e. specifically for CUDA program execution ,we have this concept of warps.

So basically, that would mean which are the threads that are going to execute together in lockstep at least from the programmers point of view if not from the hardware's point of view. So we will from this lecture discuss in more detail with respect to that - how the GPU essentially is going to schedule the threads in packets of size warps and what are the performance issues that can actually occur if the programmer is not aware of the way the program is going to get scheduled.

(Refer Slide Time 01:26)

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

December 3, 2019



Warp Scheduling and Divergence Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

(Refer Slide Time 01:29)

Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6



Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So just to do a summary here. The topic is warp scheduling and divergence and this is our week 5 topic here.

(Refer Slide Time 01:33)

GPU can be viewed as an array of Streaming Multiprocessors (SMs) Each SM has the following elements

- ▶ Registers that can be partitioned among threads of execution
- ▶ Several Caches: Shared memory, Constant, Texture, L1 etc
- ▶ Warp Schedulers (More on this later)
- ▶ Scalar Processors(SPs) for integer and floating-point operations
- ▶ Special Function Units (SFUs) for single-precision floating-point transcendental functions



Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And so doing a summary of overall GPU architecture. If you remember that the GPU can be viewed as an array of streaming multiprocessors where each SM (the streaming multiprocessors) has the following elements. It has got a large register file that can be partitioned among the threads for execution. So it is basically portioned across the SMs i.e. shared across the SPs the scalar processors inside the SMs.

Inside the SM, apart from having a large register file, you also have several types of cache. For example, you have a piece of memory segment which can be partitioned to behave as a shared memory and also a part of it to be as an L1 cache. Refer to our earlier lectures on GPU architecture for this. And also inside the SM you have got separate cache memory area for storing constant values to be used inside the program - the constant cache and also the texture cache.

Also each SM has got a specific piece of hardware which is called the warp scheduler which is going to decide which threads are going to execute when inside the SM. The actual execution units comprised the scalar processors which contain integer ALU and the floating point units as a separate unit i.e. not as a part of the scalar processor. You also have special function units present inside the SM which are responsible for computing the transcendental functions for example trigonometric functions.

(Refer Slide Time 03:07)

Table: CUDA Device Memory Types and Scopes

Variables Declaration	Memory	Scope	Lifetime
Automatic Variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
<code>device shared int SharedVar</code>	Shared	Block	Kernel
<code>device int GlobalVar</code>	Global	Grid	Application
<code>device constant int ConstVar</code>	Constant	Grid	Application

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor



Now let us review the different data types you use inside your CUDA programs and their corresponding scopes that get decided based on the way in which you declare the variable. So as long as your variable is not an array variable it is an automatic variable i.e. its scope is a thread. So it is specifically going to be stored in the register. It is not an array variable. It is going to be stored in a register. As long as the register is available, its scope is part thread and the life time of the variable is the execution of the kernel.

So if you have a local variable V which is going to be used by every thread, then every thread will see its own copy of this variable and with the corresponding mapping in one of the registers. If you have an array variable, for that also the scope is thread. Every thread will see its own version of the array variable if it is declared inside the kernel as a local array variable.

The memory type is local. That means it will be stored in the segment of the DRAM for the execution of this thread. Of course, since it is part of the DRAM, it is local memory in terms of in the parlance of the programmer. But physically it is located far away so the access is slow again for the life time of that kernel. If it is declared as shared, then the place where this variable will be mapped physically is the shared memory segments of the SMs.

The visibility will be the block. That means all the threads inside the same executing thread block will have consistent view of this variable which is defined as a shared variable. The life time is again that of the kernel. That means the moment, the kernel finishes its execution, this variable scope is lost. And then if we declare a variable as a global variable i.e. you declare it simply as a variable type (as a device type variable) without the annotation of shared or something similar.

So that would actually infer it as a global variable. It will be resident in the global memory and then for that the scope is actually grid. That means the variable is live across the entire execution of the application i.e. it is live across multiple kernel instances as operated from the host programs site. If the variable type is defined as constant, then its location is in the specific constant location or constant memory location of the memory hierarchy of the GPU.

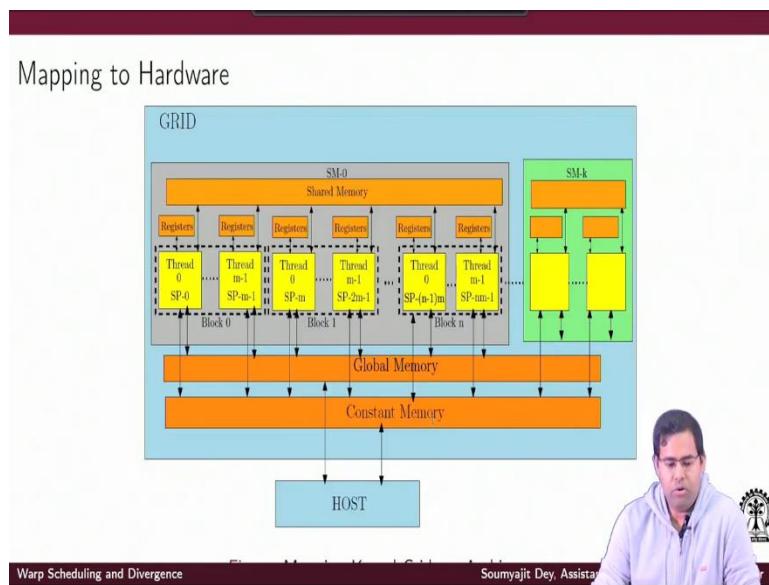
Again the scope is grid and its lifetime is the application that means it is consistently accessible by multiple instances or multiple different kernels. All of them are assumed to be controlled by the same host program. So one application is one host program. One host program can comprise multiple kernel launches. So for scope such as grid, the variable has to be a type global or constant. If its definition should be constant, it should have a constant qualifier.

If nothing is there it infers type as device. Then infer that is a global variable type (device) and if you have the shared shared, then you have this defined as a shared variable. But with shared variables, as we know that the scope is the block, all the threads inside the thread block will have

a consistent view. Lifetime would be kernel and also for normal array and non-array type variables. Depending on whether it is array then memory is locals. If it is non array i.e a single instance, then the memory is register. For both of them the scope is thread.

For this, every thread has a different copy of them located in the local memory or the local memory, essentially with the private memory corresponding to the thread in the DRAM or the corresponding register that the thread has been allocated. And again ,the lifetime is kernel. So these are the different memory types and scopes that we can define for the CUDA device memory side. So as you can see, as long as it is a part of the shared or global or the constant type memory, the scope is specified explicitly with this device annotation here.

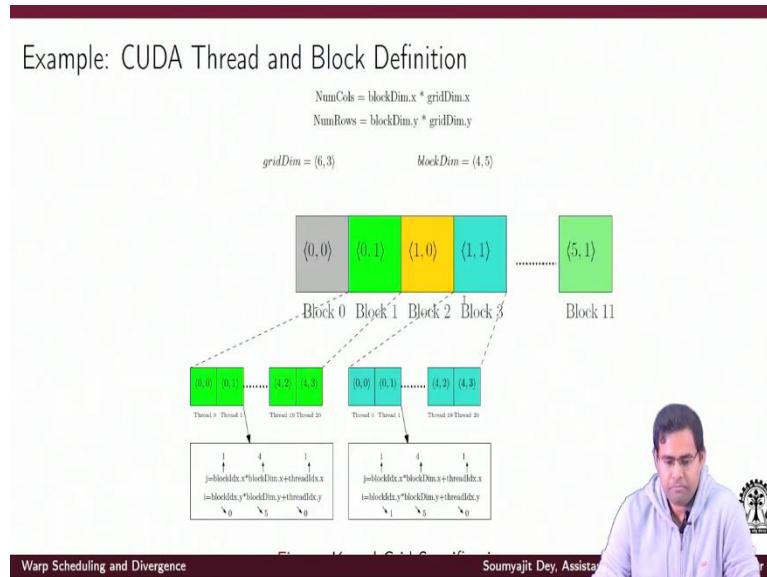
(Refer Slide Time 07:59)



So with this progress, now the question that comes is how really do threads get mapped to the hardware. So this is an example picture that we are trying to give.. So we are trying to say that suppose I have defined threads in a way that I have blocks and each thread block contains **m** number of threads. We are trying to show a sample mapping here where we are trying to show multiple SMs. So these are SMs 0,,SM k. In between we have with the dots. (We are trying to represent there are lot of other SMs in between.) Inside each SM we are trying to give a figure that how the register file is logically partitioned for the SP's i.e. for the thread to do the computation privately, some significant some part of the register is locked for each thread to do their computation, store their automatic variables etc.

And each thread is mapped to each of the SP's . So if I have a thread block, then the threads in the block are getting mapped to one specific SP like this. We are just trying to show a possible mapping. Now of course the question may come that suppose I have more than this **m** number of blocks or let us say I have more blocks than there are SP's available inside an SM. So then what will really happen?.

(Refer Slide Time 09:27)



Before going to those questions, we are trying to keep an example where we try to recall one of the example mappings discussed earlier. So this is one example of 2D mapping here. We are trying to show that - We have this many blocks in the 2 dimensional space and for each block we have again threads which have been mapped into the blocks like this.

(Refer Slide Time 09:54)

Generalized Mapping Scenario

- ▶ Let us consider a scenario for the grid and block dimensions specified above.
- ▶ $gridDim = < 6, 2 >$ and $blockDim = < 5, 4 >$
- ▶ $\#SMs = 6$ $\#SPs$ per $SM = 40$
- ▶ Two Blocks are mapped to one SM at a time.
- ▶ Hardware resources are completely utilized.

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor



So this was one of the earlier examples We are just again here for example purpose and consider a generalized mapping scenario here. So you have got these 20 threads inside this block and you have got total of 20 blocks.. So you have 20 blocks and 20 threads here. So we may consider the mapping scenario where we have a grid set of blocks and the block dimension, lets say 5, 4. So you have 20 blocks with 20 threads per block.

Consider that you have a 6 SMs and the total number of scalar processors per SM is 40. So we are considering a scenario where you have an SM containing 40 SPs and you have thread blocks of size 20. So what does that mean? That would mean that 2 blocks are mapped to 1 SM at a time. Why is that a good thing? Because your hardware resources are completely utilized.

Why is that so? The reason is you have to remember this as a rule of thumb. A block cannot be partitioned while mapping across SM's. A block cannot map half of the block to SM 1 and half of the block to SM 2. That is not allowed. So as long as you have blocks, the thread blocks size which is like you have the scenario is where you have the block dimension and the number of SPs in the SM is a multiple of the block dimension.

You can have an integer number of blocks getting mapped inside the SMs without any SP underutilized. So that is the good mapping. So with that example mapping instance shared in the earlier picture where everything was in symbols, we are considering M number of threads and

these M threads were getting mapped to this. Here, I mean M number of threads per block and we are considering N blocks inside an SM.

So in this example what is happening is we are having 2 blocks. So we are having 2 thread blocks which are getting mapped to 1 SM here. Each thread block is containing 20 threads. Just to recall here, we have discussed there would be 40 SPs per SM and in total for each block, there are 20 threads. So that would mean for the 40 SPs, we can logically partition them across 20 sets of 20 and we can map each block to one of the collections of 20 SPs right.

So I have block (0, 0) mapped to this nice collection of 20 SPs. I have block (0, 1) mapped to this nice collection of 20 SP's like this for SM 0. Overall, if I am considering 6 SMs and then I can go on like this and map the other blocks. Since I have 6 SM's and I have got these many blocks to map. So I should be able to keep on mapping like this- 2 thread blocks per SM.

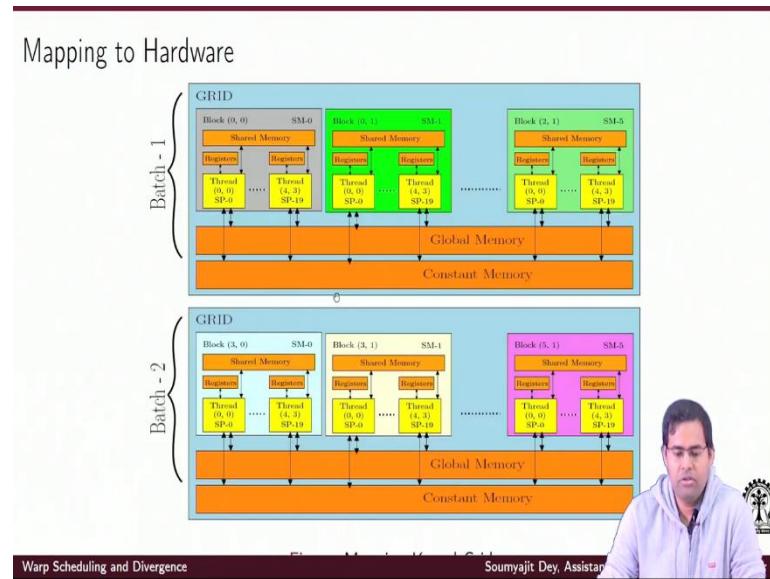
And in the way, I would be having an execution of how many blocks? As you can see 2 blocks are getting executed per SM. I have got 6 SM's so overall, I have 12 blocks executing in parallel inside this entire GPU system. Well what about the other blocks? I had 20 of them. The scheduler is holding them back because there are not enough CUDA codes available execute all of the thread blocks in parallel now which is quite general scenario.

Your kernel dimensions, launch parameter dimensions allow you to launch a large number of threads. Of course, physically you may not have that many cores available across these hierarchy of SMs. But that does not matter. The hardware scheduler takes care of mapping these thread blocks inside these SM's. This is an example packing I am showing with these kinds of mappings.

And as some thread blocks will finish execution, some other thread block actually gets mapped into one of the SM's and in this way the execution will continue. So consider this kind of a mapping in a resource constant scenario. We consider a scenario where the resources of the architecture are limited. Say your grid dimensions is like this. You had these grid dimensions and you had 40 SPs per SM as discussed in the earlier example.

Now you are considering that your grid dimension (6, 2) just like earlier and block dimension was (5, 4) just like earlier. But now instead of considering SPs per SM as 40, let us start considering #SPs per SM is 20. So what happens now? Since your SPs per SM is 20 at a time inside 1 SM, I can execute only 1 thread block. So there will be further serialization for execution of thread blocks.

(Refer Slide Time 15:25)



So that would mean in 1 SM, I have got only one block executing. In another SM, I have got another block executing. So with respect to the earlier example I have got more number of blocks waiting to execute because earlier I was executing more number of blocks per SM. So earlier I was executing with 6 SMs, 12 blocks in parallel. But now with this 6 SMs I am executing 6 blocks in parallel.

So the number of batches per thread block execution becomes large and its a more sequential scenario. So now I would have in this way less number of blocks executing parallel due to lack of hardware resources and more number of batching of thread blocks. So this batching of thread blocks and decision of which block goes to which SM would be different and it will actually be decided by a global hardware scheduler resident inside the GPU.

(Refer Slide Time 16:21)

SM, SP, Block and thread

8

- ▶ thread block max size : 1024 (modern archs 2048)
- ▶ SM can store max 1024 "thread contexts"
- ▶ can have much less than 1024 SPs
- ▶ GTX 970 : 13 SMs : 13 X 1024 thread contexts in parallel
- ▶ GTX 970 : 128 SP per SM

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor



Now some general examples here. When we speak of thread blocks, SPs and SMs, the maximum allowed thread block size is 1024 in modern architectures. It can actually become 2048. It is also the property of the architecture. Now the reason is that an SM can store maximum 1024 thread contexts. So every SM will be able to , rather has to remember what are the threads that are mapping to it even if the thread is executing or not.

So that is the context of the thread. So that context needs to be stored and this limitation on the overall thread block size actually comes from the hardware requirement with respect to storing this maximum number of thread contexts. So of course, the SM can have less than 1024 SPs but it does not matter. As we discussed, I do not really need to execute all the threads in the block in parallel.

The constant of 1024 or 2048 for that matter comes from the SM's ability to remember the context of the thread. It has been mapped a full block or may be multiple blocks. It should be able to remember the context of the threads. The amount of memory available is something finite. In this case it is for 1024 thread contexts, the amount of memory is limited. Then only that many threads can be part of 1 block.

So although because even with that setting although the SM can have less than 1024 SP's that means all the thread inside the block are not executing in parallel. But the SM can remember the context of the thread and actually schedule the threads using a smaller number of SP's by

remembering the context of the thread ok it executed up to this point then some other thread then other some other thread like that we will see some examples.

If we look at an example of GTX 970 then there are 13 SM's. So overall it can actually remember 13×1024 thread contexts in parallel and however it does not mean that many SPs are available. It has got only one 128 per SM but it can remember that every SM can store this many number of thread contexts in parallel and accordingly it can schedule the execution of maximum 1024 threads per block inside the SM's by suitably choosing the threads inside the blocks and packing them together for execution on the SPs. So this is where the scheduler which is inside the SM will come into play.

(Refer Slide Time 19:19)

SM, SP, Block and thread

- ▶ One block in one SM
- ▶ One SM can have multiple blocks

If SM can store max 1024 "thread contexts", and block size is 256, we have 4 blocks per SM.

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor



So 1 block goes in 1 SM. One block cannot be divided across 2 SM or more SMs. 1 SM can have multiple blocks. If an SM stores maximum 1024 thread contexts and block size is 256, we have 4 blocks per SM. So 1024 is giving you the maximum block size that is allowed and of course you are free to actually define a block size that is less than that. But since the SM can store max 1024 thread contexts and if you give a blocks size of smaller size (let's say 256), then when you are going to launch your kernel, the SM can be mapped with 4 thread blocks. Your block size is 256 This is important. Suppose you have chosen to write a program where you have launched a kernel with block size 256. Since the SM can store a maximum of 1024 thread contexts i.e. you can remember the context of these many executing threads while executing a sub part of them being parallel as defined by the number of SPs.

But still since the max is 1024, your block size is a smaller value of 256. The SM can be mapped with 4 thread blocks (256 times 4 is 1024) and this mapping is decided by the high level scheduler who is distributing the blocks across the SMs. So the number of blocks that will be mapped to the SM is defined by the number of threads per block, while remembering the total number of thread contexts that the SM can remember.

(Refer Slide Time 20:56)

GPU HW scheduler

- ▶ The hw scheduler decided which threads to map to a collection of SPs in SIMD fashion :: SIMT model of execution
- ▶ This collection is physically guaranteed to execute in parallel
- ▶ The unit of such collections is "warp"

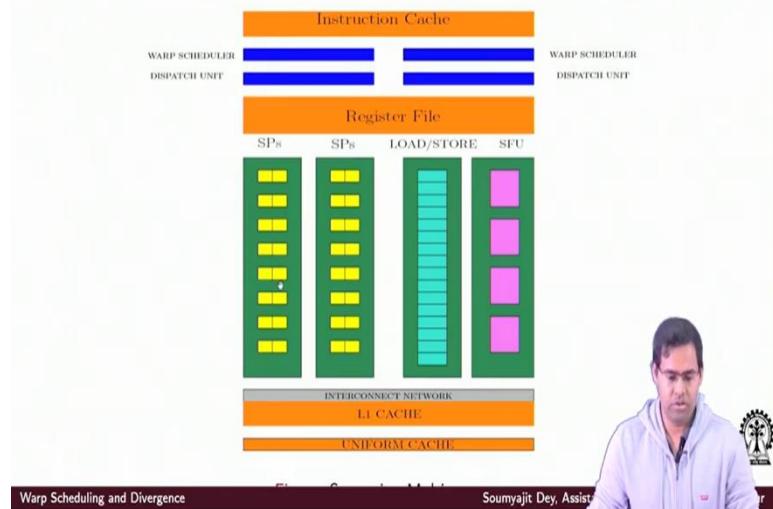
Warp Scheduling and Divergence Soumyajit Dey, Assistant Professor

So this is what the hardware scheduler decides. It decides which thread block will map to a collection of SPs or to map into SM. And inside the SM, we have a secondary hardware scheduler which decides that these are the thread blocks that have been mapped to this SM. Out of these thread blocks which are the physical threads which would be packed together for an execution, this binding of physical threads together perform a lockstep execution which is what we had defined earlier as a warp.

So this is the basic unit of execution inside a GPU's SM. So this is done by the hardware scheduler blocks sitting inside the SM. So the SM will be assigned a set of thread blocks or maybe one thread block. Inside the threads blocks, you have multiple threads which will actually execute in parallel in units of execution i.e warp which will be decided by the GPU's SM hardware scheduler.

(Refer Slide Time 21:56)

SM: A closer look



So this is basically the way we are trying to give the picture. So we have an SM. Inside the SM you have this warp scheduler as a hardware unit deciding which of the threads are going to be dispatched together through the SPs for execution in lockstep.

(Refer Slide Time 22:17)

Warp

- ▶ Warp is a unit of thread Scheduling in SMs.
- ▶ Warp size is implementation specific (typically 32 threads)
- ▶ Warps are executed in an SIMD fashion i.e. the warp scheduler launches warps of threads and each warp typically executes one instruction across parallel threads.

Ex : If a SM has 128 SPs, it can execute 4 Warps at a given time (one Warp has 32 Threads)

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor

So just to summarize the warp is the unit of the thread scheduling inside the SMs. Warp's size is implementation specific. Typically it is still now 32 threads inside a warp. Now this warp is executed in the SIMD fashion i.e the warp scheduler launches a warp of threads. Each warp typically executes one instruction across parallel threads. If a SM has 128 SPs it can execute 4 warps at any given time. One warp has 32 threads.

So a question is which are the thread ids that are the part of the same warp that is decided by the warp scheduler sitting inside the SM? So just to review it here, the warp scheduler decides which threads to pack together for execution in a warp. These are all threads with consecutive thread ids so that the threads in the warp are guaranteed to execute in the same SIMD distribution instruction together.

It executes one SIMD instruction followed by the next SIMD instruction in lockstep like that. Different warps can actually progress through the SM together. Why? Because I have more than 32 SPs. For example I have let's say 128 SP's. So in parallel, 4 warps may execute together. So inside a thread block, I have lot of threads. These threads are packed into packets of 32 by the warp scheduler.

These packets of 32 threads called warps will execute in lockstep together from the programmer's point of view in terms of a instruction. But again, the warps may progress at different speeds. Each warp progressing] ensures that the threads inside the warp progress at the same speed. But I have different multiple warps actually going here decided by the warp scheduler.

So which are the real warps that are executing it depends. That is why when I launch a thread block containing that number of threads as we have discussed earlier, there is no guarantee that all the threads are progressing at the same speed. The threads will be packed into warps as decided by the warps scheduler. And it will decide which packet of thread will progress at which speed. That is why we will require the syncthread or some kind of some synchronization primitive like that to ensure or force synchronization among threads.

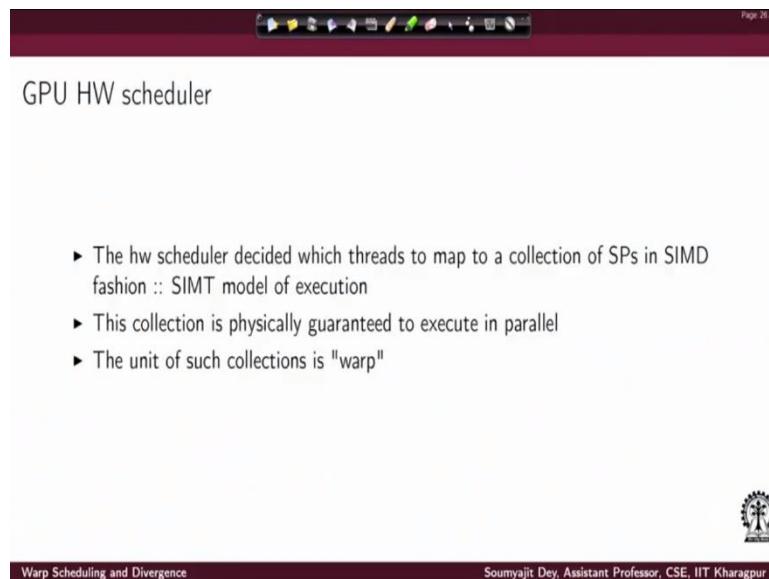
So that they actually have a consistent view of data points on which they want to collaborate and work together. With that we would like to end this lecture and we will resume from this point Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 04
Lecture No # 17
Warp Scheduling and Divergence

Hi, So, welcome to our lecture series on GPU architectures and programming. So in the last assignment we were discussing about GPU hardware schedulers and definition of a warp. So, we will start from that point.

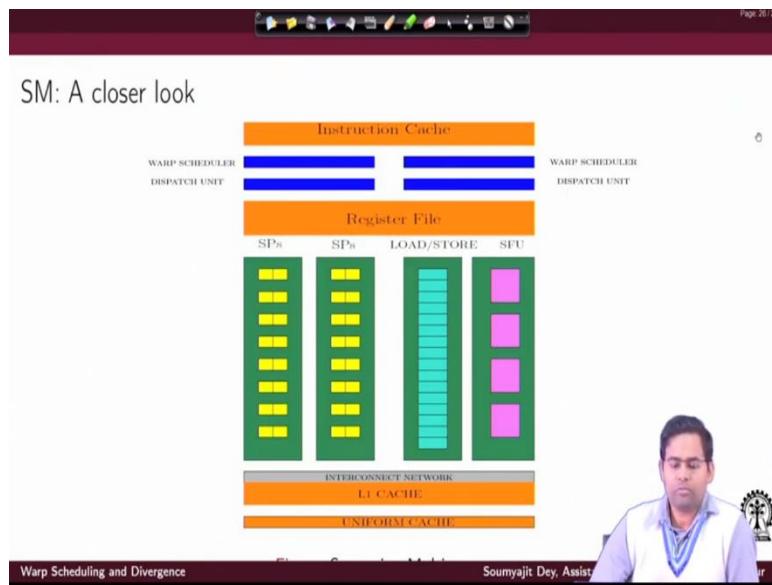
(Refer Slide time 00:40)



So essentially when we map kernels, we launch sets of threads on a GPU hardware. So we have a hierarchy of hardware schedulers which will now come into play. So at the high level there will be 1 component of a hardware scheduler which will decide which thread block gets mapped to which of the SMs. And at the lower level inside each of the SMs there would be another component scheduler which will decide which of the threads of the mapped thread blocks will constitute what we call as a warp.

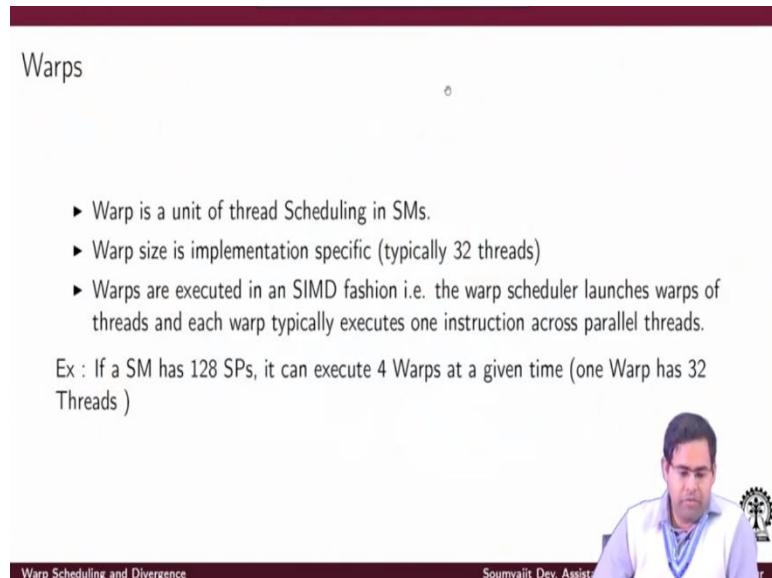
So a warp will essentially be a collection of such threads which execute in parallel as discussed earlier. So this packing of threads from thread blocks to warps is something that will be decided by the lower level scheduler

(Refer Slide Time 01:30)



And if we closely look into an SM we have this kind of warp scheduler sitting inside the SM and there is a high level scheduler which is distributing blocks into SMs for execution.

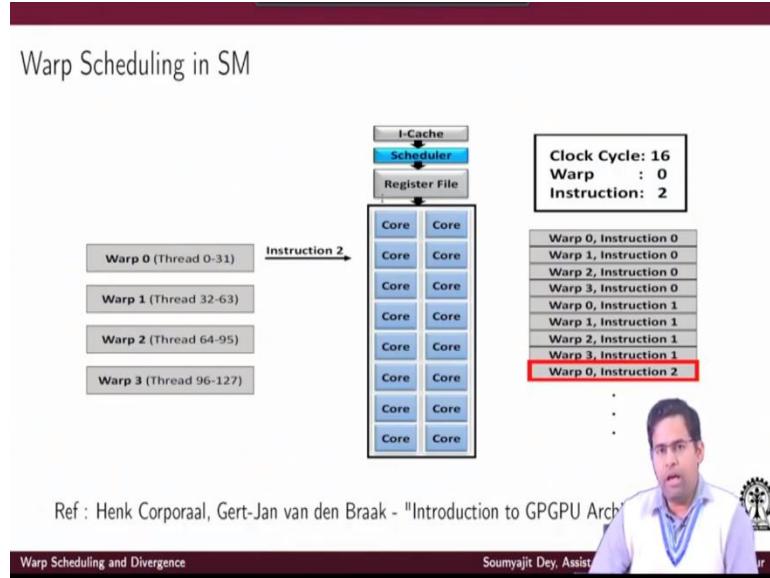
(Refer Slide Time 01:44)



And so just to summarize again what a warp is. It is a collection of 32 threads which will execute using the SIMD instruction sequences and the warps get mapped into the collection of SPs for execution. So if 1 SM has say 128 SPs, then it can execute 4 warps at any given time. Since 1 warp has 32 threads, so at any time in parallel the SM we will have 4 warps proceeding through its SPs in parallel.

(Refer Slide Time 02:15)

Warp Scheduling in SM



So here we have an example scenario where we are trying to provide a situation that we have an SM where as you can see we have a collection of SP cores. So here you have in total 16 SP cores. And consider the situation that you have some warps progressing through this SM. So for example let us see what warp 0 its constituting. Now this packing of threads into the warps is what is going to be done by the scheduler which is sitting inside each of the SM as we discussed. This is the second level scheduler in the hierarchy.

The high level scheduler is dispatching thread blocks to each SM and inside this SM, we have this kind of scheduler who is actually forming these warps. So this low level SM scheduler is actually forming these warps like the form of warp 0, warp 1. So this kind of hiding each warp and it is deciding in each warp which are the threading indexes that should be constituting each of the warps.

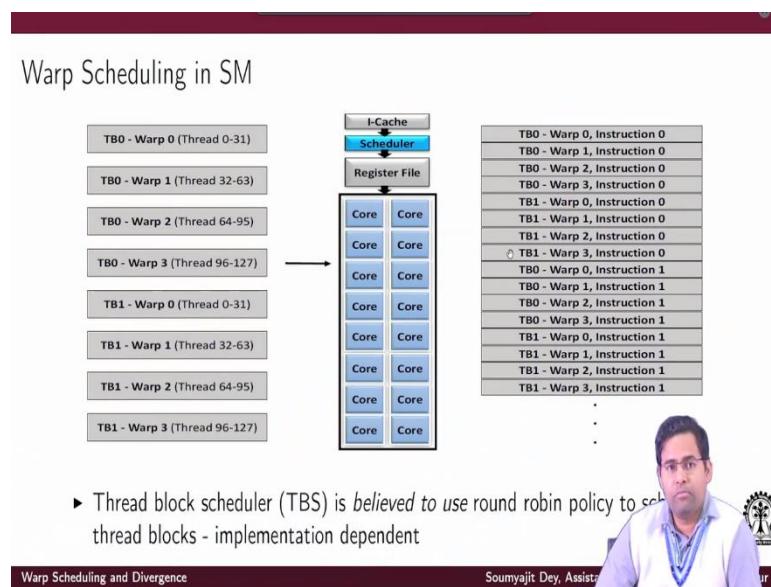
So I have warp 0, warp 1, warp 2, warp 3 and they are containing threads with ID 0 to 31, 32 to 63, 64 to 95 and so on so forth. They are executing the instructions that are part of instruction sequence for that CUDA kernel under question. So just to look into this scenario on the right-hand side, we are trying to say that what is the exact sequence of these warps that are executing. Suppose I have an instruction sequence that is instruction 0 followed by instruction 1, followed by instruction 2.

So in this hypothetical system, I am executing this warp. As you can see that I have a collection of these 16 cores and each warp is containing 32 threads. So assuming everybody takes a single cycle of execution for each of the instructions on the core, there is no floating point delay due to other kinds of complex operations. Each warp would then complete with 2 clock cycles right because there are 32 operations to execute in parallel for the warp while I have only 16 cores that are available.

So the warp would technically take 2 clock cycles. So in that way for instruction 0 to be executed by all the 4 warps as you can see here, each warp is consuming 2 clock cycles. So for these 4 warps, all of them completing instruction 0, I would require 8 clock cycles. Similarly, I would again require 8 clock cycles for instruction 1 and this was the execution precedes here.

So again for warp 0 executing instruction 2 that would start in the 16 clock cycle. This is just an example scenario we are trying to give. So we are trying to say that this is how the warps may get scheduled here on the core and the number of cycles that the warp would take depends on the number of SP's that are available. And also the kind of instruction that the warp is executing.

(Refer Slide Time 05:33)



Now have a relook into the picture when I am considering warps belonging to different thread blocks. I am now considering warp 0 from thread block 0 and then warp 1, warp 2, warp 3 all from thread block 0 and then again warp 0, warp 1, warp 2, warp 3 all from thread block 1 right.

So now I have 4 warps. Here 4 of them belong to thread block 0, 4 of them belong to thread block 1.

And if we map their execution here on an SM with 16 SP cores like earlier, I would have each of the warps executing in two clock cycles. So I would have for example here, instruction 0 getting executed by all the warps - warp 0, warp 1, warp 2, warp 3 belonging to thread block 0. Then instruction 0 is executed by warp 0, warp 1, warp 2, warp 3 belonging to thread block 1. And similarly, with instruction 0 completed by all the thread block warps ,I have instruction 1 again getting executed for warps belonging to thread block 0.

Then again I have instruction 1 getting executed for all the warps that belong to thread block 1. So in this case of course, we are having these assumptions that there is a thread block scheduler which is using a round robin policy to schedule the thread blocks. Now this is what we have as an open domain answer in the academic research papers that people are assuming that the NVIDIA thread block scheduler is following this kind of round robin policy.

But of course, it depends on your implementation. These are the exact hardware scheduling strategies for the thread block schedulers as well as the warp schedulers if not, they are in the open domain. And of course, it can also be implementation dependent, depending on different scenarios. Somebody can implement the GPU hardware in a different way to have a different possible scheduling of thread blocks.

From a programmer's point of view, we will always assume that the warps can proceed at their own speeds. And of course, they have to satisfy the instruction sequence requirement that is enforced by the thread.

(Refer Slide Time 07:57)

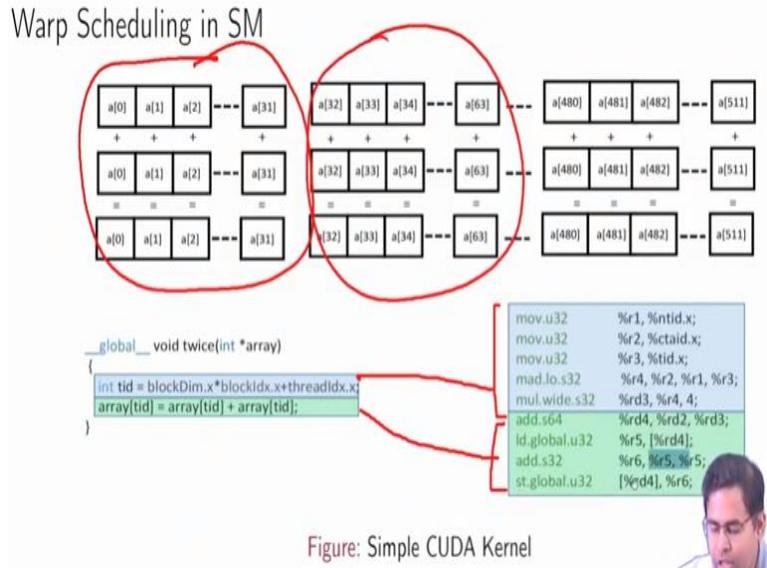


Figure: Simple CUDA Kernel



So just looking into some more examples. Here we have a very simple kernel where we can see that the kernel is 2 line code segments on the left hand side. So we are just computing the thread ID and we are then using the thread ID to access some specific location of an array, and simply adding the value in the location with itself and storing right there. And on the upper part of the figure we show how the warp will execute the operations.

So in warp 0, I would have the locations A[0..31] getting updated sequence of operations. So this is what would get done by warp 0 and then next we would have warp 1, warp 2 and so on and so forth. I am just trying to show how the warps would be packing this addition operation spread over the width of the data space. Now, we take a closer look in to the corresponding PTX instruction for the actual CUDA code we have on the left-hand side.

So on the right hand side we have the PTX instruction. So the initial part of the instructions correspond to the thread ID computation part and the second part of the instruction actually corresponds to the additive operation that has been done. So just take a second and have a closer look in to the instruction sequence here. So essentially as you can see that the first 5 instructions which are highlighted in blue are responsible for doing the thread id calculation here.

And finally in the second highlighted part we are using this value of the thread ID to bring the content of the array at that location corresponding to the thread ID into this register here. So essentially this is your final thread ID here. And then you use the thread ID to load the

corresponding location's value using this load instruction here in r5. And then you are just adding up the content of r5 and storing into r6. And then you do a global store back in to the location for the same thread ID right.

(Refer slide Time 10:45)

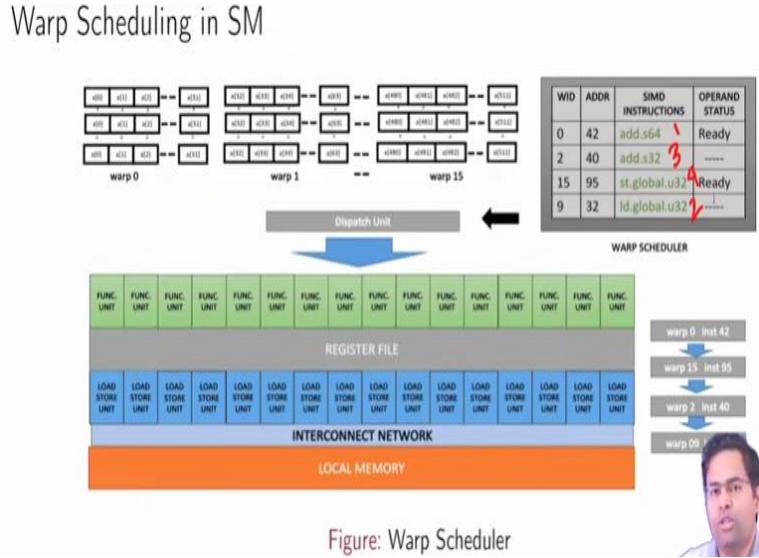


Figure: Warp Scheduler

So if we now have a look into that sequence of instructions that are executing there. As you can see you have an address followed by a load operation followed by another add operation and then a store operation. So for this different packing of warps like warp 0, warp 1, warp 2, the corresponding instruction would be dispatched to the different functional units of the SM. How really would their execution be ordered?

So of course for this the warp scheduler needs the operands for each of the instructions to be ready right. So there would be a mapping table from which the warp scheduler gets to know what is the warp ID which is ready to execute. So for example, this is the sample space. We are showing here that warp 0 is ready to execute from this address 42. And the corresponding SM instruction that needs to be executed i.e. this add is 64 which is basically the first instruction.

So if we just number the instruction for that part, this is the first instruction. If you again have a loop, the next add is the third instruction and between you have the load and then the store. So the original sequence of the instruction is as I have marked in here. And as you can see what we are trying to communicate here is different warps may be in a ready state for different possible

instructions depending on what are the operands available for that instruction i.e. whether they are ready.

And also for that warp, whether the preceding instructions have already executed. So here we have one possible valid execution sequence for this instruction. So the warp 0 would execute instruction with address 42. So basically this is the warp 0 executing the first instruction corresponding to this sum right. This is the first instruction corresponding to sum operation that warp 0 is executing.

Now this is followed by warp 15 going to execute the fourth instruction corresponding to the sum operation. So how can this be possible? Well of course that would mean that warp 15 must have already executed the previous instructions corresponding to the sum operation for each part of the data space. Now in this example, we have warp 15 followed by warp 2. So essentially we are saying that somehow warp 15 made progress at a higher speed and then warp 2 executes its instruction number 3.

That means by this time warp 2 has executed its first and second instruction and then warp 9 would be executing its corresponding instruction which is the second instruction in the sequence. So the idea is very simple. Every warp needs to follow the instruction sequence 1, 2, 3, 4 as has been marked. This is the first instruction, second instruction, third instruction, fourth instruction. This is a situation we are trying to show here.

So warp 0 is going to execute first instruction followed by warp 2 executing the third instruction. This is just an example we are trying to say. So when warp 0 is executed, the first instruction warp 15 is executing the fourth instruction. That would mean that the warp 15 has already executed instruction number 1, 2 and 3. Now the requirement is every warp needs to execute the instruction first, second, third, and fourth in the sequence.

But it is not necessary that warp 0 would execute instructions 1, 2, 3, 4 followed by warp 1 executing instruction want to support like that. Now, here it is necessary that warp 0 executes instruction 1 and then warp 2 execute instruction 1, warp 3 executes instruction 1 like that. They are free to proceed depending on as and when the operands are ready. So in this example, we are trying to say that it is not only the case that the warps would be scheduled following the specific

policy. It is also the situation that the warps required the corresponding operands values to be readily available for making some progress right.

(Refer Slide time 15:32)

Warp Scheduling in SM

- ▶ Issue one “ready-to-go” warp instruction/cycle
- ▶ Use operand score-boarding to prevent hazards
- ▶ Issue selection based on round-robin/age of warp
- ▶ Score-boarding determines if a thread is ready to execute?
- ▶ Scoreboard is a HW implemented table that tracks - instrs fetched, resource availability for fetched instrs (FU and operand), register file modifications by instrs.

Somyajit Dey, Asst. Prof., IIT Kharagpur

So just to summarize, how does really warp scheduling take place in the SM? So the warp scheduler will issue for ready-to-go warps, one warp's instruction per cycle. And it will use an operand score boarding to prevent hazards. So this is the kind of table we are trying to show here just as an example - that what are the warps which are ready to go? Now I mean which ones are ready to go. They actually are being computed by using a score boarding technique. We are not getting into too much of detail of that.

But of course you need to understand that just like the pipeline, here also, I need to prevent hazards. That means I should not be executing a warp before the operands become readily available or I should not issue a warp before the functional units become free right. So these are the potential hazard scenarios which need to be prevented. For that, we will be using the score boarding approach.

Now the issue selection is based on round robin. I mean there can be specific examples. There can be a round robin scheduling of warps as an example we discussed earlier. However that need not be the only system. It can also be based on how long is the warp active for how much time i.e. depending on the age of the warp and of course the warp has to be ready i.e the corresponding operands need to be available and functional units need to be available.

The score boarding determines whether in that way, the thread is ready to execute and essentially it is a hardware implemented table which tracks different scenarios like whether the instruction has really been fetched, whether the resources required by the instruction are really available. That means whether the functional units are available, whether the operands are ready and whatever is the register file that is going to be modified by this instruction - whether that can also be done or not.

So the score board actually takes care of providing a status that whether executing this instruction would be hazard free. If so, then it would say that well this instruction is ready to go with corresponding 2 operands and functional units. And then the warp scheduler will use its implementation algorithm, whether it is round robin or some other technique to decide whether to issue the warp's instruction or not.

So this is how possibly the warp scheduling is executed. Of course some part of it is implementation dependent but these are the current techniques that are in actual implementation in modern GPUs.

(Refer Slide Time 18:17)

Latency Tolerance

- ▶ When threads in one warp execute a long-latency operation (read from global memory), the warp scheduler will dispatch and execute other warps until that operation is finished.
- ▶ Other long latency operations : FP units, Branch instructions
- ▶ After all, all threads in the same control-flow execute same instruction sequence on different data points !
- ▶ A common practice is to launch thread blocks of a size that is a multiple of the warp size to maximally utilize threads.
- ▶ Slow global memory accesses by threads in a warp may be optimized using coalescing (more on this later)



Now the other important thing is that why really do we have this kind of warp-based executions and why do we really have this kind of multiple SPs i.e. why is the presence of SPs in very high number inside the SMs?. The reason is that you want the GPUs to tolerate long latency

operations. So when threads in one warp executes such long latency operation, Eg. a read operation from the global memory, we know that read from the shared memory is much faster with respective read from the global memory which is very slow.

So whenever a warp would require any read operation from the global memory, it has to wait for a long amount of time. Now I have this warp scheduler internally built into the SM. When this warp is waiting for the operands to be available, the warp scheduler can dispatch and execute many other warps until the operands are available for the warp that was waiting.

Also a warp may have to wait for many other long latency operations like floating point operations, branches etc. So overall the issue is that - if a long latency operation is there, then the GPU should be able to hide the corresponding penalty. Then the way to hide the penalty is you always keep your functional units engaged by using some other warp. That is why the warp scheduler has a huge role to play in terms of feeding all the SPs with as many operations possible by dispatching warps.

Whenever some executing warps stalls due to a long latency operation, one has to remember that all threads that you have which are following the same control flow i.e following the same sequence of ifs and same sequence of instructions, they are essentially executing the same instruction sequence. So it does not really matter which threads execute fast or which threads inside another warp executes slow.

As long as they do not have any dependency among each other. Because they are all doing the same instruction sequence computation on possibly different data points in the memory. So also a common practice in this regard is to launch thread blocks of a size that is multiple of the warp size. Now this is something, we have also discussed earlier. If we launch a thread block with size which is multiple of the warp size then I have all warps completely filled up with 32 threads.

Otherwise, if I have a thread block size which is not a multiple of warp size then I will have some warps which are not really full with 32 threads and that would actually transfer to some hardware not getting utilized while executing that warp. And also there is the important question of whenever the threads in a warp are executing a long latency operation like a global memory read.

The warps can be optimized in a such a way i.e. the threads can be reading in an optimized way. Such that the warp will always access consecutive global memory locations so that the fetch can be done in parallel. This is known as global memory coalescing optimization which is something we will take up later on.

(Refer Slide Time 21:43)

Efficient use of thread blocks

Target System Constraints

- ▶ A maximum of 8 blocks and 1024 threads per SM
- ▶ A maximum of 512 threads per block

(Handwritten notes in red ink: 4, 12, 128, 64, 8, 76, 512, 1024, 1029)

Table: Solutions for various block scenarios

Input Block Size	Blocks per SM	Threads per Block	Remarks
8 * 8	12	64	SM execution resources will be underutilized
16 * 16	4	256	Achieves full thread capacity in SMs
32 * 32	1	1024	Exceeds the limit of 512 threads per block



Now the next thing is how to make a good efficient use of thread blocks. So we take a simple situation here considering some target system constants. So you have a maximum of 8 blocks and 1024 threads per SM. And you have a the maximum number of blocks threads allowed inside block or the thread block size, let it be 512. So you are going to have 1024 threads per SM. You can have maximum 8 blocks.

What ever you choose as a thread block size and whatever you choose as a number of threads that you launch, in total we are restricting that 1 SM can handle 1024 threads and 1 SM can handle 8 blocks. It is just as synthetic scenario we are assuming just as an example here. And let us also consider that a thread block size is limited to 512. Now in this situation consider different possible input block sizes for some kernel launch parameters.

Consider the input block size as 8 x 8 and so you have blocks per SM as 12. And then in this case, if your input block size is 8 x 8 and you have blocks per SM as 12, then since you are having blocks per SM as 12 and input block size is 8 cross 8, so you naturally have a straight

forward block as 64. So in this case our observation would be that the execution resources in the SM would be underutilized.

Now the question is why that so? So in this case you are actually using the number of blocks per SM as 12 ,while you have threads per block as 64, Now there would be a problem. Since you have 64 blocks, 64 threads per block. So when the SM would be executing, you have in total a thread block size of 64 and in total you have 12 blocks. So overall that would be your total number of threads that you are essentially launching?

So overall you have launched essentially 768 threads. So ideally I would say that SM could have handled more number of threads. So execution resources are practically underutilized. Now consider another situation where you have got this input block size 16 x 16 2D definitions. Now you are going to launch 4 blocks in the SM and the number of threads that you are allowing per block in naturally 16 x16 which is 256.

So your total number of threads you are launching is essentially a number that SM can maximally handle. So in this case, you are using the full capacity of the SM. Now consider a third scenario. Suppose you are defining a kernel with per block parameters 32 x 32 and you are just launching 1 block. So then as you can see 32 cross 32 is also 1024. So essentially you are defining a thread block of size 1024.

Now in this example we have taken it violates the definition. That I am not going to allow here for this specific example with more than 512 threads per block. So this exceeds the limit and is not allowed. So these are the typical problems that you can have while mapping a kernel to a GPU. You have to know the hardware's limitations. So as we have discussed earlier, that there are such specific limitations and you need to know them to define the corresponding different parameters spaces and corresponding launch parameters for that count.

(Refer Slide time 26:35)

Querying Device Properties

CUDA API provides constructs for obtaining properties of the target GPU.

- ▶ `cudaGetDeviceCount()`: Obtains the number of devices in the system.
- ▶ `cudaGetDeviceProperties()`: Returns the property values of a particular device

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor



Now something about different ways in which a GPU can be queried. So the CUDA library provides you several specific constructs - these kind of constructs using which from your program you can actually figure out what is the hardware configuration of the target GPU. For example in your program you can use this function called `CUDAGetDeviceCount()` to give you the number of devices in the system.

You can use this function called `CUDAGetDeviceProperties` which again would return different possible property values for some particular device in the system. Now why is this important? Because as we discussed earlier your program's kernel launch parameters may be decided based on these different properties that you get. And also when I write my CUDA kernel, I would have the parameters of the kernel which may be certain access expressions in the kernel in terms of variables which would get initialized based on the different CUDA device properties that I can acquire through this kind of library function process.

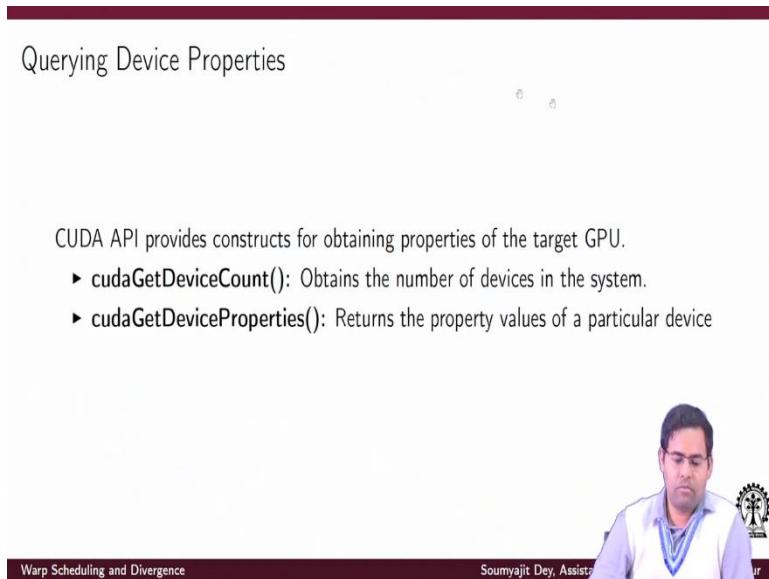
So with this we will conclude this lecture and from the next lecture we will take a deeper look into different querying and different ways in which device properties can be required. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 04
Lecture No # 18
Warp Scheduling and Divergence (Contd.)

Hi welcome to the lecture series on a GPU programming and architecture. So in the last lecture we have been discussing about how to query the GPU device properties and there is a good reason behind that. You can write your programs in a architecture oriented parameterized manner. So you can first run this kind of diagnostic code to figure out what are the different architecture parameters. And accordingly you can decide the launch parameters and some other parameters of your kernel so that it executes optimally in a given architecture with respect to the execution time of the kernel.

(Refer slide Time 01:07)



The screenshot shows a presentation slide with a dark blue header bar. The main title 'Querying Device Properties' is centered in white font. Below the title, there is a large, faint watermark-like image of a person's face. At the bottom of the slide, there is a dark footer bar with white text. From left to right, the footer text reads: 'Warp Scheduling and Divergence', 'Soumyajit Dey, Assistant Professor', and 'IIT Kharagpur'. The slide content itself is a list of CUDA API functions:

- ▶ `cudaGetDeviceCount()`: Obtains the number of devices in the system.
- ▶ `cudaGetDeviceProperties()`: Returns the property values of a particular device

So we were discussing some specific CUDA APIs for target GPU systems. For example if I use this `CUDAGetDeviceCount()` function call, I get to know what is the number of devices in the system. I can use a `CUDAGetDeviceProperties()` called to return different property values for a particular device.

(Refer Slide Time 01:23)

Querying Device Properties

```
int main()
{
    int devCount;
    cudaGetDeviceCount(&devCount);
    for (int i = 0; i < devCount; ++i)
    {
        cudaDeviceProp devp;
        cudaGetDeviceProperties(&devp, i);
        printDevProp(devp);
    }
    return 0;
}
```



Now we will see some highlights of an example program. So consider this scenario that you have a small host side code from which you are launching this `CUDAGetDeviceCount()` function call and it would be returning the device count in the variable that has been passed here right. And then you are going to use this variable initialized from this call.

You are going to use it to upper bound the number of iterations for the subsequent loop and then using this loop - the first thing you do is you initialize type a variable `devp` of type `CUDAdeviceProp` and then for each index value `i` of this loop from `i=0..devCount`, you are making queries per device right. So you are now calling the function `CUDAGetDeviceProperties()`, passing a structure of type `CUDAdeviceProp` (the name of it is `devp` here in this code) and passing it the device id - device id 0, device id 1 up to the device `devCount` which is the total number of devices up to the last device.

So in that way for every device id you have the corresponding device information getting stored in the CUDA structure `devp` of type `CUDAdeviceProp` and then you are going to print using this specific function `printDevProp` property right.

(Refer Slide Time 03:03)

Querying Device Properties

```
void printDevProp(cudaDeviceProp devProp)
{
    printf("Major revision number: %d\n",devProp.major);
    printf("Minor revision number: %d\n",devProp.minor);
    printf("Name: %s\n",devProp.name);
    printf("Total global memory: %u\n",devProp.totalGlobalMem);
    printf("Total shared memory per block: %u\n", devProp.sharedMemPerBlock);
    printf("Total registers per block: %d\n", devProp.regsPerBlock);
    printf("Warp size: %d\n",devProp.warpSize);
    printf("Maximum memory pitch: %u\n",devProp.memPitch);
    printf("Maximum threads per block: %d\n",devProp.maxThreadsPerBlock);
    for (int i = 0; i < 3; ++i)
        printf("Maximum dimension %d of block: %d\n",i,devProp.maxThreadsDim[i]);
    for (int i = 0; i < 3; ++i)
        printf("Maximum dimension %d of grid: %d\n", i, devProp.maxGridSize[i]);
}
```

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor



So this is something which we have customized for our purpose - the printDevProp function. So essentially it is being passed this device properties structure right. And so essentially in this case it is expecting that it would be given a CUDADeviceProperty type structure and that would be having several other fields which will be printing through this printf statement. So I am not going to discuss these printf statements in details. It does not make much sense.

(Refer Slide Time 03:36)

Querying Device Properties

```
printf("Clock rate: %d\n",devProp.clockRate);
printf("Total constant memory: %u\n", devProp.totalConstMem);
printf("Texture alignment: %u\n", devProp.textureAlignment);
printf("Concurrent copy and execution: %s\n", (devProp.deviceOverlap ? "Yes"
    : "No"));
printf("Number of multiprocessors: %d\n",devProp.multiProcessorCount);
return;
}
```

Rather than that let us look at what really the code will output. And of course the slides will be with you and you can have a look into how is essentially these different printf statements are organized.

(Refer Slide Time 03:49)

Example: Tesla K40m Characteristics

```
Major revision number: 3
Minor revision number: 5
Name: Tesla K40m
Total global memory: 3405643776
Total shared memory per block: 49152
Total registers per block: 65536
Warp size: 32
Maximum memory pitch: 2147483647
Maximum threads per block: 1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 2147483647
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Clock rate: 745000
Total constant memory: 65536
Texture alignment: 512
Concurrent copy and execution: Yes
Number of multiprocessors: 15
```

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor



And so overall if I run that code, these are the different device properties I can extract. So I can actually extract the different revision numbers of the architecture. So that is basically the device properties major and minor field. So this major and minor field is telling me what is the major revision number and the minor revision number of the architecture - the name of the architecture family. So that is the name of the device family and then I have the total global memory size available for each device.

So for this device I have a total memory size like this and then there are several other memory segments Eg: these are total global memory. What is the amount of shared memory per block? What is the amount of registers per block? What is the warp size, maximum memory pitch? What is the maximum number of threads allowed per block? And for each of the block definitions what is the maximum dimension of the block in each of the 3 possible dimensions.

And finally lifting from blocks to grids what is the allowed dimension of the grid in the 3 possible dimensions x y and z. So these are the possible values that get printed and of course the GPU's clock rate. The total amount of constant memory i.e. the texture memory and whether the GPU supports certain other properties like concurrent execution of kernel? What are the total number of SM in the GPU and all that? So if we execute the earlier diagnostic code, these are the different messages that can get printed and they can be useful for framing your launch parameters of the kernel.

(Refer Slide Time 05:42)

Control Flow Divergence

- ▶ Threads inside a warp execute the same instruction.
- ▶ How does a warp handle if statements / branch instructions?
- ▶ The GPU is not capable of running both the if else blocks at the same time.



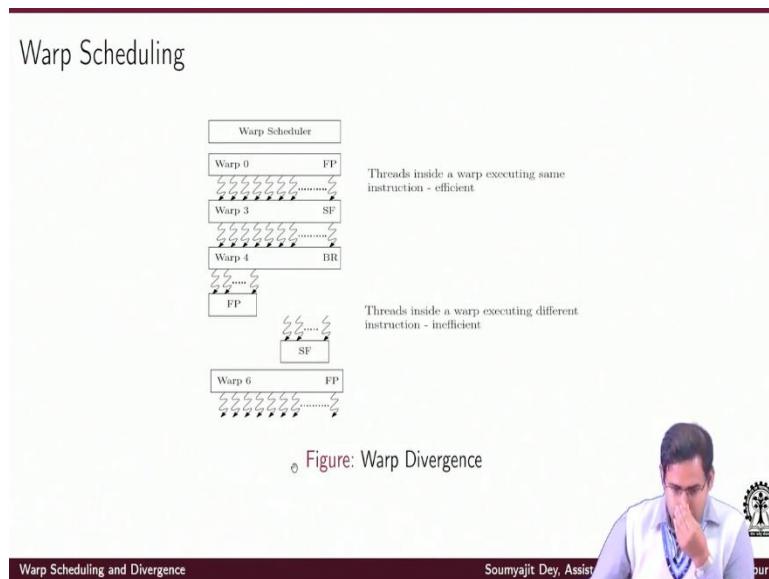
Now so that is about how you can extract the device properties. Since we are actually discussing about scheduling the warps, there is another major issue which impedes the performance of a kernel which is called control flow divergence. Till now we have been just discussing that when you launch the kernel you are essentially launching multiple thread blocks. Each of the thread blocks progressed at its own speed. Inside a thread block I have a collection of warps that are decided by the internal schedulers and the warps will also proceed at their own speed. Since each of the SMs are able to execute a lot of warps together in parallel, it can tolerate the latencies of long latency operations and in that way a GPU is able to make progress with lot of threads in parallel. The progress of each thread may be slow due to latency of operations. But overall the GPU would be giving you high throughput because it is completing a lot of operations in parallel. However, this question that how many operations really the GPU is executing in parallel depends on something very important which is the flow of control inside the kernel.

We have been assuming the threads can make progress in parallel inside the warp but that really need not always be the case. For example, we know that by definition of a warp that the threads inside the warp execute the same instruction. At any point of time they are executing from a programmer's point of view and are executing the same instruction. What if there is a branch statement which the warp is executing and some of the threads in the warp satisfying the condition of the branch while some of the threads in the block do not satisfy the condition of the branch?

The GPU is not capable of running both the if and else blocks at the same time. That is not how it is designed that i) it will make progress with the warp or ii) it will form the warp into 2 parallel lines. So that some of the threads progress with the if part of the code and some of the threads in the same warp would progress with the else part of the code. If you just think logically, that also does not make sense because if you allow the hardware to fork a warp and still progress in parallel there can be further subsequent forks right.

So how will really the hardware able to handle so many numbers of forks of a warp and make it progress? It does not make sense in terms of implementation complexity right. So we are not getting into that detail right now. But rather we will try to understand how really a GPU handles such divergence of control flow that comes from branch executions.

(Refer Slide Time 08:32)



So suppose you have a warp scheduler and it is executing these warps and as long as the thread inside the warp are executing the same instruction it is a very efficient situation. The problem comes as we discussed that threads inside a warp start executing different instructions. So then the number of threads which are executing in parallel will reduce and we will have less amount of throughput than required. This phenomenon which is known as warp divergence which leads to performance loss from the perspective of the GPU's throughput.

(Refer Slide Time 09:12)

Divergent Code 1

Consider the following kernel code

```
__global__
void divergence(float *M)
{
    /*P1:*/ int tid=blockIdx.x*blockDim.x+threadIdx.x;
    /*P2:*/ if(tid%2)
    /*P3:*/     M[j]+=2;
    else
    /*P4:*/     M[j]-=2;
    /*P5:*/ M[j]*=2;
}
```

Half the threads of a warp execute the addition instruction while the other half execute the subtraction instruction.



Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor

So as an example, let us consider the divergent code. So this is a kernel which is a very simple kernel. We are just labeling each of the statements in this kernel with levels P1, P2, P3, P4, P5. There is a reason for that. So the program statement with label P1 just computes the global thread id for that specific kernel - for that specific thread in the kernel. The program statement with label P2 decides whether that thread will actually satisfy or not satisfy the if block.

So essentially all the thread ids which are even would satisfy it and they would get in to do the execution of the statement P3. The thread ids which are odd indices would actually get in and execute the statement P4. And after that, both the threads together would execute the statement P5. So overall here, we have a scenario that since the if condition is $tid \% 2$, half of the threads of a warp would execute the addition instruction while the other half of the threads in the same warp would execute the subtraction instruction.

And as we have decided, it is not the case that the warp would actually split, and half of the instructions execute one instruction and half of the threads execute the other instruction in parallel. That is never going to happen in the GPU.

(Refer Slide Time 10:38)

The Hardware's Job

The GPU has hardware support for handling divergent branch instructions in code.

- ▶ The PTX Assembler maintains internal masks, a branch synchronization stack and special markers
- ▶ The PTX Assembler sets a **branch synchronization marker** first for the divergent *if* statement that pushes the active mask on a stack inside each SIMD thread
- ▶ Depending on the value of the mask relevant threads execute instructions,
- ▶ Once the instructions in the *if* block are finished, the active mask is popped from the stack, flipped and pushed back.

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor



So how really does the hardware handle this scenario? So the GPU does the following whenever such a branch comes. So if we look at the corresponding PTX code for this kind of kernel we would see that the PTX assembler maintains internal masks or essentially the execution of all such instructions what we call as predicated instructions. So it maintains the mask - which is a bit pattern that would decide which instruction would really be executed and the status of the instructions are actually stored in something called a branch synchronization stack.

So the PTX assembler sets the branch in synchronization marker first for the divergent statement (the *if* statement) and pushes the mask on the branch synchronization stack for each of the SIMD threads. It will decide that whether the mask value would actually decide which thread is executing or not executing. Depending on the value of this mask, the warp scheduler simply makes progress with the instruction.

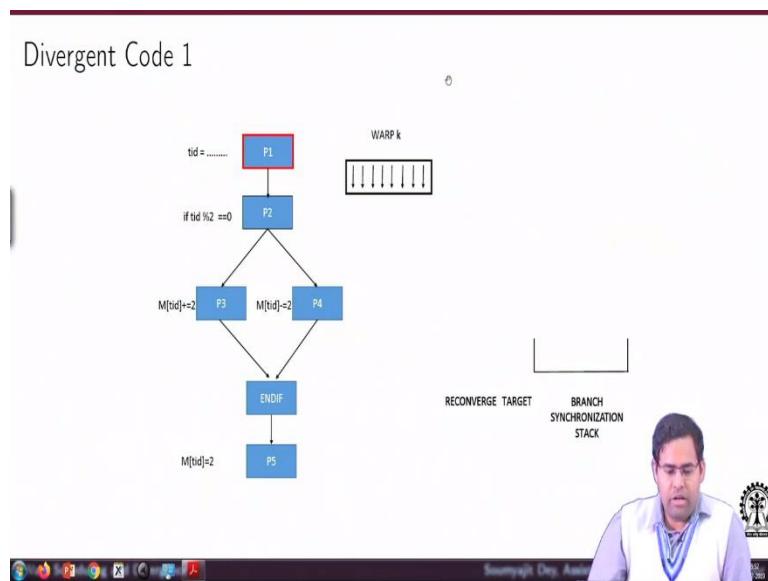
And once the instructions for the *if* block are finished, then the active mask is popped out from the stack and it is flipped. So when it is getting flipped, essentially the mask changes. So whichever 1 entries will now change to 0 and whichever were earlier 0 entries would now get changed to 1. So essentially, the other instructions, the other thread ids which we are now stalled would now start executing. Essentially once the *if* block finishes the *else* block starts execution.

Well it is too verbose a description possible. So we would make it much more alive through a running example in this case. Just a brief summary in this case would be that when the

corresponding assembly code is generated a suitable marker is decided. And at the run time when this predicated instruction, instructions with markers are getting processed - a separate data structure a branch synchronization stack is set up.

And which instruction would make progress and which instruction would not make progress is actually encoded as a 1 0 pattern and stored in the branch instruction stack. So when the if block progresses the threads having a 1 in that stack would make progress. Once the if block ends I actually bring out this 1 0 pattern, flip it and if I flip the pattern now essentially then in terms of the hardware it would know that what ever is corresponding to one in the pattern would actually be the threads in the else block and now they would make progress.

(Refer Slide Time 13:28)



Let us look at it rigorously from with a example here So this is what we called a control flow graph of the corresponding program we had earlier. So we are making good use of the labels here. So just have a re look into the different labels P1, P2, P3, P4, and P5 for the kernel. So P1 is for the thread id computation, P2 for if, P3 for the addition then after the else I have the P4 for the subtraction operation and finally the common operation has got P5.

So here I have P1 followed by P2 the branch that is a if condition (`if tid % 2 == 0`) and then I have a fork here to P3 or P4. So some threads would go in here and some threads would go in

here and once this if condition ends, I have the thread synchronization together and then I have all of them executing P5 i.e. this operation. So I believe this actually should be *=2 here right.

So now let us try and find out how the warp progresses with the help of this kind of a branch synchronization stack and the predicated execution of the masks that we have been discussing earlier. So let us say these are the threads which are going to execute for the kth warp and they are right now executing the instruction label P1. Here, all of them compute the thread id.

Then all of them comes to the branch instruction. So when all of them comes to the branch instruction, it will actually see that the branch condition is satisfied by the threads with even. So those who actually satisfy would be considered as 1 and the thread id which do not satisfy is considered as 0 and in this way the active mask is created - the 1 0 pattern, and this initialized active mask is now stored into the branch synchronization stack.

Essentially, the location of the 1 decides which threads make progress and this is how the hardware takes care of making progress with the warp. It simply looks into the active mask accordingly the scene as we discussed that this actually signified which of the predicated instruction would make progress. Since each of the even thread ids are having 1, those are the thread ids that would progress. As you can see that now the I have half of the threads making progress i.e. the tids which are even.

So they would all execute P3. Once the execution of P3 is done, then this mask bits are complemented. So P3 is done. I have complementation of mask bits. So this pattern is going to change to the reverse pattern. With this reverse pattern the hardware will again start executing the mask instructions following this mask. So it should essentially use this sequence to compute which of the threads will make progress.

These are the alternate threads which are making progress. After this, all the threads are supposed to converge together and start executing together. So that would mean from the branch branch synchronization stack, the mask is popped and since there is nothing here so the hardware warp scheduler would simply make progress with the entire warp right. So the question which is obvious here is why do I at all need a stack kind of thing? Because here I was simply making progress with the warp.

Since there was a branch instruction I actually computed which are the threads that satisfy the branch from a hardware implementation perspective. I represented that with a 10 bit pattern. Whichever has got 1 i.e. which tid was 1, corresponding to that I set some mask, made some of the predicated instructions active and I made them flow. Corresponding threads made progress. And after those threads executed P3, in terms of the hardware I just reverse the bit pattern.

So then I could get the other thread ids to make progress with the alternate flow of instructions or alternate flow of instructions which was in P4 and finally I actually reached the synchronization point. And at that point I just set the mask and I just made progress with all the threads right. So that is the simple thing. Consider the scenario where you have this kind of cascaded ids which is so common in a general program. How would you then handle it? For this, I would actually require this kind of branch synchronization stack.

(Refer Slide Time 18:37)

Divergent Code 2

Let us consider an example that has nested if/else statements.

```
__global__
void divergence(float *M)
{
    /*P1*/    int tid=blockIdx.x*blockDim.x+threadIdx.x;
    /*P2*/    if(tid%2==0)
    {
        /*P3*/        if(tid%3==0)
        /*P4*/            M[tid]+=3;
        else
        /*P5*/            M[tid]-=3;
    }
    else
    {
        /*P6*/        if(tid%3==0)
        /*P7*/            M[tid]-=3;
        else
        /*P8*/            M[tid]+=3;
    }
    /*P9*/    M[tid]*=6;
```

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor

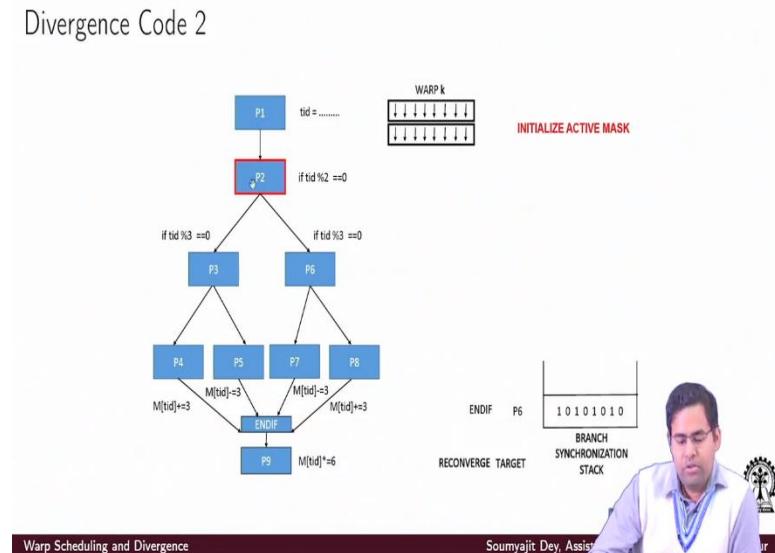


So we consider this second example of a divergent code. Now this is where we are actually considering nested if else statements and we are trying to figure out how this nested if else statements would actually be handled by our divergence handling scheme in terms of that branch synchronization stack. So just to look into it very minutely, first we compute the tid. If the tid is even we get into a branch inside this branch. We also check whether the even tids are divisible by 3.

So only those tids which are overall divisible by 6 would get into this part and do an addition by 3. Otherwise the even tids which are not divisible by 3 would get into the else block and subtract 3. For all the odd tids, if they are divisible by 3 you have something. If they are not divisible by 3 you have something else. And finally everything converges to location P9. So this is a more complex program. The complexity is just in terms of the nesting of if else statements.

Have a relook into the original program. We had a single level of if elses. Here in this program we are considering the nesting of if else statements.

(Refer Slide Time 19:58)



Look in to the control flow graph here. So you have positions P1 for tid computation, P2 for the first branch, P3 for the next branch which is only satisfied for everything which is % 2 and also percentile 3 i.e tids which are divisible by 6. Then you execute P4 otherwise you execute P5. When both are done you execute P6, then you execute P7 and then P8. Finally everything would synchronize and you go to P9 right.

So this is the structure of the control flow graph. Now let us look into our original idea of warp execution. So your warp starts. You initialize the active mask so at P2 you compute the mask which is basically the sequence of 1 0 1 0 like that because $\text{tid } \% 2 == 0$. So all the even thread ids are satisfying the mask. So you make progress here with all the even tids and you are here.

And then what you do is again since you are facing a nested if which is the situation which we were discussing, you have already taken care of the first if. Based on that if, you are pushed in a mask. Once inside the if part of the logic you are facing the second if. So for this second if you have this condition. So first you push the old mask into the stack. What is the old mask? The old mask simply signifies what is your original thread of computation?

The old mask signifies execution of the if part of the first nest. Now you are actually going to come into the if part of the second nesting. So how to do that? So you have pushed the old mask once again and then you are applying this condition on the old mask. So essentially you are trying to figure out which of the tids are going to progress and execute P4. So figuring that out would mean applying this condition of the if on the old mask and finding out which are the threads that are really going to execute here.

Now as you can see from the condition, only the threads which are divisible by 6 in terms of tid will execute P4 and that is what it is. So it is the old mask and you apply the condition here. So that would give you all 0s and only 1s at the 6 positions of the tid - 0, 1, 2, 3, 4, 5, 6 only. So only this thread id will proceed and execute P4. That is all that's happening. After that you are supposed to execute P5. How will that happen.

So all you need to do is you have to complement the element bits of this entry in this step right. So you just compliment the entries because this was your old mask. This is your old mask's 2nd instance. This is the second instance because of the nesting. First you apply your mask on this instance. And figure out who will execute the second if and then you just reverse this pattern. With the reverse pattern, you execute the else inside the if.

So this is the progress of these 3 threads here. Once this is done, you will just pop the stack. Why? Because this segment of P3, P4, P5 execution is done. So you are essentially done with the if part of the first level if. Now you are going to process the else part of the first level if. How do you do that? Now you are using the older logic. So all you are going to do is you are going to complement the relevant bits. So, this was your original bit pattern on the old mask. You complement the relevant bits.

Once you complement the relevant bits, you have the other half of the warps progressing to P6 where again you have another mask to execute. So again following the old algorithm you push the old mask. So this is the old mask corresponding to the reverse pattern here. You are taking care of the other threads that are executing inside this else. So for this old mask, again you will apply this condition $\text{tid} \% 3 == 0$

If you apply this condition, you get only this thread which will execute here. Why? Because it has to be an odd thread id 0, 1, 2, 3 but still it has to be $\text{tid} \% 3 == 0$. So it will only be made by thread id 3 because the next thing $\text{tid} \% 3 == 0$ that is 6 which is not odd. So it would definitely be a 0 here. So you will have only one thread progressing and then the rest of the threads needs to progress with the else part. For that again you have to complement this mask.

So this mask is complemented and accordingly the hardware decides which other threads would progress. Once this is done, you have got all the diverging threads reconverging back here in this ENDIF and that is signified by popping the stack. So after this, there would be another synchronization. So the first time you popped the stack to identify that this sequence of second level ifs have converged.

But then you figure that the first level if has also converged right. So that is signified by popping the stack again. I hope this is clear. There were two pops - the first pop signifying this flow. Let me mark the flows here. So the first pop signifies here that this part of the flow has converged and then you will pop the stack again and empty it because that would signify that from the high level the entire flow has converged. So you pop the stack completely.

So overall, this is the idea that in terms of the hardware and the PTX predicated instructions, how GPUs would really handle this divergence of warps and then reconverge back the threads in the warp. One important thing you notice here is that whenever the warp diverges you have performance loss. Why? Because whenever the warp diverges you have less number of threads executing per warp. Because whenever I have a divergence half of the threads are not executed.

The threads which satisfy the if condition would progress and then subsequently at some other point of time the threads that is satisfying the else condition would progress. So essentially I have the warp making progress with some of the threads executing which satisfied the if

condition. And then at some point of time other threads are executing which satisfies the else. So whenever there is a divergence in your execution of a CUDA program you have performance loss.

So divergence is not a good behavior for your program and there is a lot of research which go on in developing tools and techniques which identify whether your program can exhibit divergence, whether it may be possible to overcome it by an alternate program and things like that. But a divergence is a serious issue which as a programmer you need to be aware of. So that you can understand the way the program is going to behave in terms of architectural timing.

(Refer Slide time 28:13)

Programming tips

- ▶ GPU programmer has to be aware of hardware imposed restrictions - threads/SM, blocks/SM, threads/blocks, threads/warps
- ▶ The only safe way to synchronize threads from different blocks is to terminate kernel and make a fresh launch at the target synchronization point



In this lecture we would like to summarize with some specific programming tip that as a GPU programmer you have to be architecture aware i.e. you need to be aware of the hardware imposed restrictions. The hardware imposes restrictions in terms of how many threads can be mapped into an SM. How many blocks can be mapped into an SM? How many threads are allowed per block and how many threads execute inside a warp? These are the figures which actually help you to extract performance in programs. And some things are very important. With respect to our earlier coverage of synchronization of threads it would be this. As we have discussed that synchronization primitives of GPU programs work inside thread block.

So suppose you are trying to synchronize or have collaborative execution among threads across multiple blocks. That is not allowed. So the only safe way to synchronize threads from different

blocks would be to terminate the kernel right at that point. That would mean that all the computation get done at that point and then you will make a fresh launch of a kernel at that point. So all the threads would get launched again with a consistent view of the memory. So with this we will be ending this lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 04
Lecture No # 19
Memory Access Coalescing

In the last lecture, we have been discussing about how GPU warps are formed and how they are scheduled inside the SMs. In this lecture we will be focusing on one of the important topics relevant to optimizing GPU programs and that is memory access coalescing.

(Refer Slide Time 00:51)

Recap: Memory Spaces

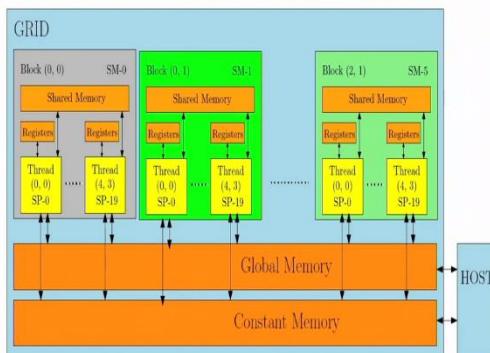


Figure: Global Memory Accesses



So before getting into the detail of that, let us first do a small recap on the hierarchy in the GPU memory - How the different memory segment are organized? and how really the global memory is accessed by the computing threads inside an SM? So as we have discussed earlier when a kernel launches, essentially a grid of thread blocks are launched. Multiple blocks can reside inside one SM as decided by the high level scheduler of the GPU and it depends on the size of the blocks.

One thread block cannot be split across the SM and so when the set of blocks get mapped to set of SMs (in this example we are just showing a single block mapping in to single SM), essentially I have a set of threads inside the blocks which are computing which are being packed

by the low level scheduler of the SM into the units of computing called warps. And each of these warps perform progress. They actually progress through the instruction sequence automatically.

All the threads inside the warp are executing in lock step with respect to the instructions. So here we had just shown in this earlier picture that actually how each of the SPs i.e. the scalar processors are getting mapped to the threads and how they are executing. And of course we will have scenarios where we have more number of threads rather than the number of SPs. So it would require another low level scheduling of the threads on the SPs and that is how things are done as discussed earlier.

Now whenever the threads while executing an instruction, they will require access to the global memory. So they would definitely require 2 kinds of access. They would need to bring operands from memory and they would need to write back updated variable values to the memory. So this essentially should be happening through the hierarchy of the GPU - the memory hierarchy of the GPU.

So finally all the variables have their locations in the global memory and also while you have written the program, there are certain constant factors. You may have mapped them also into the constant memory for faster access right. So when a thread is asking for a variable's value it may possibly be residing in the L1 cache or the L2 cache or the shared memory part or if you are lucky it may be residing in the register file segment allocated for the thread.

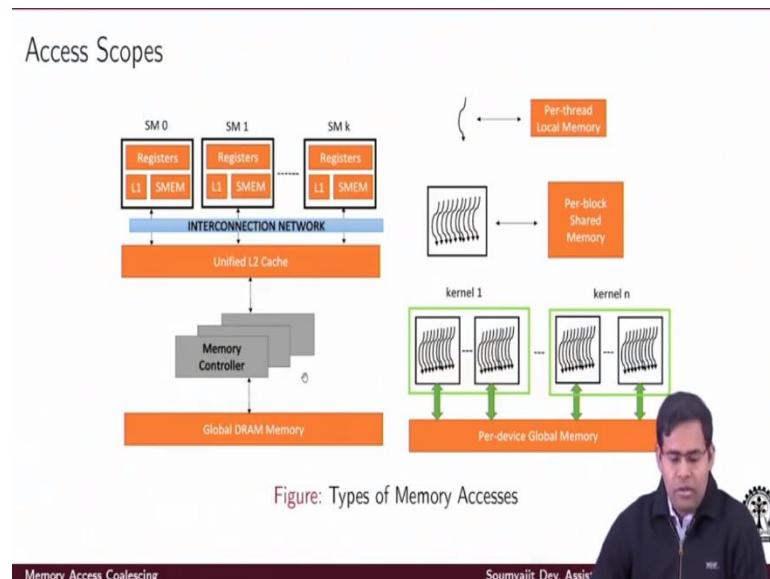
Otherwise, the thread has to access this value from the global memory. Now of course by standard computer architectures techniques as we know a registered access is first, shared memory access is bit slow with respect to the register. But it is the global memory access which is a lot slower because essentially this is outside the SM units i.e. outside the entire GPUs compute environment. It is in the DRAM chip of the GPU.

So accessing the global memory is an order of magnitude slower. So if you are writing a program which will be doing a lot of access to the global memory, essentially your program performance will be low. It will not execute with high throughput. If you can somehow read write the program

so that effectively the number of global memory accesses are getting reduced, then your program will perform much faster.

So, essentially the important thing here is how can I reduce the number of times the global memory is going to be accessed by a program? Now this question leads to a lot of memory related optimizations, some of which we will be touching throughout this lecture sequence here. Now, of course you can understand that if we are going to reduce the number of accesses made by the program to the global memory we need to somehow optimize the accesses and make data resident in the shared memory and the register file, the shared memory or L1 cache whichever way it is getting configured as discussed earlier. And so we have to optimize the program in a way at source level.

(Refer Slide Time 05:21)



So first of all lets redo the scope of computation for the different threads. So you have got these different SMs. Each of the SMs have got their own as we discussed earlier. Each of the SM's have got their segment of register which are dedicated for them. Each of the SM's have got the memory partitions - the internal memory partitioned into the L1 and the shared memory segment. And then through their interconnection network you have access to unified L2 cache which accesses the bank global memory to the memory controller.

So for each thread with doing its computation it has got its own separate private memory segment allocated in the global memory i.e. allocated in the DRAM and is known as local

memory for the thread. For every thread block you scan, as a programmer you define an amount of shared memory which is available to a thread block for doing its computation.

So since it is shared memory, all the threads executing the block will have a consistent view of the variables in the shared memory which again means that thread i and thread j belonging to the same thread block are executing and when thread i makes an update on a variable it is immediately visible to the any other thread j this updated value, since it is in the shared memory.

And of course you can have multiple kernels i.e. in case you are executing concurrent kernels which is a feature supported by a modern version of CUDA. So that would actually mean multiple kernels executing together and they can have their part's device access to the global memory.

(Refer Slide Time 07:05)

Memory Access Types

Latency of accesses differ for different memory spaces

- ▶ Global Memory (accessible by all threads) is the slowest
- ▶ Shared Memory (accessible by threads in a block) is very fast.
- ▶ Registers (accessible by one thread) is the fastest.

Memory Access Coalescing

Soumyajit Dey, Assist

So now coming to the important question of the latency of this accesses. Now, they definitely differ for different memories spaces as we discussed earlier. So this is a nice summary here that the global memory which is accessible by all the threads is the slowest. Definitely, it is the DRAM. The shared memory which is divided into each of the different SMs is then accessible by the threads inside the thread block. It is an order of magnitude faster than what is referred to the global memory. However if your variable is resident in the registers, then its access time will be the fastest. So this is the order which is standard like any computer architecture.

(Refer Slide Time 07:44)

Warp Requests to Memory

- ▶ The GPU coalesces global memory loads and stores requested by a warp of threads into global memory transactions.¹
- ▶ A warp typically requests 32 aligned 4 byte words in one global memory transaction.
- ▶ Reducing number of global memory transactions by warps is one of the keys for optimizing execution time
- ▶ Efficient memory access expressions must be designed by the user for the warps.



Memory Access Coalescing

Soumyajit Dey, Assistant Professor, IIT Kharagpur

Now the question is how really is the global memory accessed by the threads. We have already discussed that the threads are packed together in what we call as warps. So when these warps are executing the GPU, whenever the threads inside a warp are looking for data to be assessed from the global memory, the GPU tries to gather the accesses together and form specific accesses specific in the sense that a warp's threads try to merge the accesses together into global memory accesses. This is known as coalescing.

So GPU coalesces the global memory loads and stores depending on whether threads inside a warp are looking for data sitting consecutively in the memory or not. Summarizing, I would just say that the GPU is coalescing the global memory loads and stores. Now these loads and stores requests are put up by the threads inside the warp.

And so if I have multiple threads inside the warps which are looking access global memory locations which are consecutive. They would be packed together into a single global memory transaction. So of course you have to define what are global memory transactions? What are the data points that can be brought together by a single global memory transaction and all that? So suppose you have a warp where all the threads i.e. you have 32 threads and by 32 threads I mean you have the warp executing the load instruction or a store instruction.

And in general let me call it a transaction. So essentially you have got 32 memory requests issued in parallel right. If these requests are actually going for consecutive positions in the data

space, that means you have 32 aligned 4 byte words which have been requested. Now this is exactly what is defined as a global memory transaction. So a global memory transaction is essentially a transaction in which from the global memory you can bring 32×4 number of consecutive bytes from the global memory.

So that is the width of a global memory transaction. When you are accessing the DRAM chip, this is how the hardware hierarchy is optimized. The hardware designers know that the global memory access is slow. So he is trying to provide the program an optimization that whenever you access, since your accesses are slow, I am allowing you to access lot of data points in parallel by giving you a wide bus.

What is the width of the bus? It is a 32×4 bytes. That would mean if in a warp you have 32 consecutive thread ids and they are making a load request for 32 consecutive memory locations. Essentially, you have to do a single global memory transaction. So this is the most fundamental and important thing here. Since your global memory access width is 32×4 bytes, a warp is looking to access 32 consecutive memory locations.

So essentially, we are defining each memory element as a 4 byte warp (consider let say integers). So, the warp can access its corresponding instruction corresponding data points from a memory using a single global memory transaction. Now as we discussed earlier, since this global memory transactions are slow overall objective here is to reduce the number of transactions by different warps which are executing across the SMs.

So for that what is required is that you should be able to write the program with a very efficient memory access expression. So whenever you are accessing an array position in the memory you have an access to an expression. So just to give an example, suppose you are trying to access M which is an array and you have a expression here which is telling that for the current values of i j and k this is the location you want to access and you put it in some variable a and then the access expression would be this - $i * j + k$.

So as a programmer you need to define very nice access expressions. You have to define them keeping in mind a global memory optimization so that overall when your program executes, you have lesser number of global transactions. The warps then do not need to wait so much for the

memory operations to be done (both load and stores). And so that overall progress is much faster for that kernel.

(Refer Slide Time 12:50)

The slide has a title 'Coalescing Examples' at the top left. Below it is a code snippet:

```
__global__ void memory_access(float* a)
{
    int tid= blockDim.x * blockIdx.x + threadIdx.x;
    a[tid] = a[tid] + 1;
}
```

To the right of the code is a diagram labeled 'warp 0' showing a 4x8 grid of memory locations labeled 'global memory'. The grid is organized into four rows and eight columns, with indices from 0 to 7 along the top row. The first column is labeled 'global memory'.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]
A[16]	A[17]	A[18]	A[19]	A[20]	A[21]	A[22]	A[23]
A[24]	A[25]	A[26]	A[27]	A[28]	A[29]	A[30]	A[31]

Below the grid is a small video frame of a person speaking, with the name 'Soumyajit Dey, Assistant' displayed.

Let us look at some example to make this idea very clear here. So consider the execution of this very simple program. So we have a small kernel here which is just a memory access kernel. So you are first computing the thread id using this expression - standard expression of the block dimension multiplied by block id + the thread id everything is in the x direction. So considering a single dimensional kernel, single dimensional blocks, single dimensional threads, I mean single dimensional arrangement of blocks and single dimensional arrangement of threads inside the block.

So this is your tid and all that you are doing is for every tid you are accessing the corresponding tids in this location in the array A. So lets say this is your warp 0. For this warp 0, you have got the consecutive thread ids and we are just taking an example here. So you have 8 threads executing inside warp 0. So when this line of the code is executed by all the threads inside the same warp what really happens? What are the memory elements that are accessed?

As you can see by looking into the access expression which says that you just access the memory location corresponding to the tid. So warp 0 contains tids 0 1 2 3 4 5 and like that. So warp 0 is accessing memory location A[0],A[1],A[2],A[3],A[4],A[5],A[6],A[7] etc. So considering that

this is going to be a global memory transaction, we are not drawing here 32 thread ids just to make the points very clear.

So essentially what are you really doing? You can have a single global memory transaction for this read operation. Here it is the simplest example. Again we are assuming that the global memory transaction width is 32 i.e. 8×4 bytes. We are not drawing 32 tids and all that here. So essentially I have one global memory transaction which will suffice for doing the read operation for all the thread ids here.

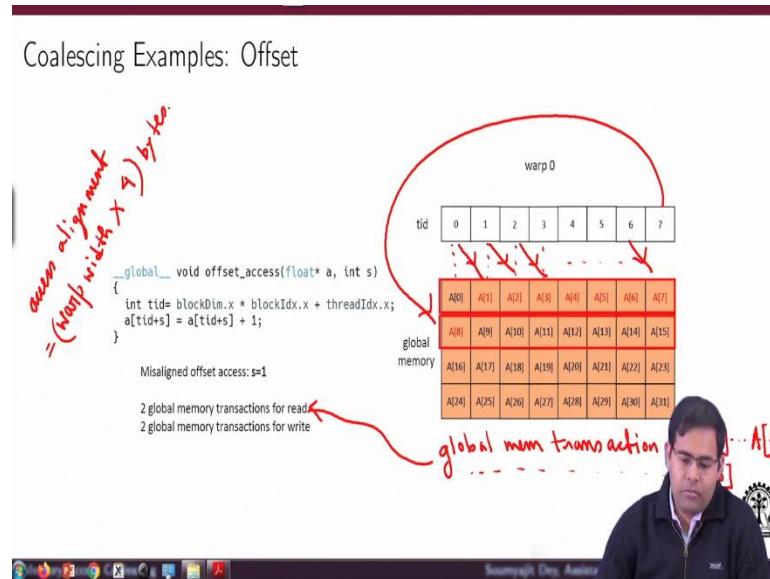
Why? Let me just repeat again. Because as we discussed that the global memory access is quite wide. Assuming that in this case the global memory access is defined as 8×4 bytes i.e. 32 bytes I can bring in parallel. So essentially for this specific case I can bring in all the data points required for executing this line. I can do the corresponding load operations here by a single global memory transaction.

So essentially, the global memory transactions would be defined based on the width of the warp. As we have discussed already since standard warp width is 32, the global memory transaction can be done in such a way that you can bring in 32×4 consecutive bytes from a memory together right. So assuming here for our simplistic example, you can bring all of them together in a single transaction.

If I just want to generalize this all then what I would like to do is put 31 here. So this would go up to 31 this next would start from 32 and this would go up to 63 and like that. We are just taking a simplistic example here. We are just considering that the warp width is 8 and similarly the global memory transaction is also 8×4 bytes right. So here we can see that since all the access are consecutive location in the memory for the threads executing inside a warp, I can have 1 transaction for doing the read and then there will be an execution of the increment operation and then again there will be one transaction for updating all of this location by another write operation for the global memory, through which all this location will be updated in parallel. Because again the location are consecutive. So what happens if the program is a bit different? So just continuing with the animation for this program with warp 0 executing and bringing everything together with 1 global memory read followed by one updating everything here in one

global memory write. Then again bringing together all the data points to one global memory read and again updating everything from this using one global memory write. Similarly for warp 2 read and write and that is how this will go on.

(Refer Slide Time 17:56)



Now consider a small difference here in the program. So let us look into the earlier program once again. So the earlier program was this $a[tid]$ is $a[tid] + 1$. In this program we are providing an extra parameter here accessing with offset and the offset value is s . So you compute using normal notion the tid and then what you do is you do the access for $a[tid + s]$ as location.

So what are you doing ? You are accessing the $tid + s$ as location, adding it up and then updating again the $a[tid + s]$ as location for the array a right. So that is what you are doing. Now let's see how it goes. So in terms of the reads what is happening? Consider the s value as 1. Now that would mean if I again start looking into warp 0 with thread id 0, I am doing an access of $a[1]$ location. So earlier the access was this, but now the access has kind of changed.

So you continue like this $tid = 7$ is looking for $A[8]$. So overall with this change what is really happening? You will have the first transaction for locations $A[1]$ to $A[7]$ and in the next transaction you just bring $A[8]$. So this together gives you 2 transactions. I hope the point is clear. Here again I have got 8 threads inside the warp they are looking for 8 data points to load to update and to write back. However the problem in the access here is offset, in the sense that the access is not properly aligned here right.

So when the access would happen for this memory, although these are consecutive locations but fundamentally if you look into our definition of the access what did we really want? There has to be 32 aligned 4 byte warps right. So for this case, by warp definition there has to be 8×4 bytes aligned access. But the alignment is wrong here. Essentially here what is happening is that you have got an access of consecutive locations but the access is starting from A[1].

So entire thing is not aligned with the warp. So that would actually lead to 2 global memory accesses here. I hope this is clear. So essentially the DRAM bandwidth is utilized with respective to the warps width. If the warps width is 8 (in this case), so the accesses would be like so. When I am defining that the access will be aligned at the position, it is basically so that the first row can be done through a global memory transaction.

So as you can see in this picture the memory organization is shown with respect to this access alignment right. So the moment, I put in offset the alignment is lost. So in the first access, I can only get things from A[1] to A[7] because if I am looking for data for A[0] to A[7] they all come inside the aligned access but A[8] cannot be aligned i.e. cannot be brought in with A7 in one access because when I bring A[7], the aligned access is done in terms the warp width times four number of bytes.

So it will actually bring in data from A[0] to A[7] from which actually A[1] to A[7] is only useful for me. So for the next thing, I will require another access right. That is why when I go for A[8] there is a different access here. Overall I have two different global memory transaction for the read operations and again I would have 2 different global memory transactions for doing the write operations.

So just the introduction of this offset may be required depending on the program that you want or if you have a desirable property for the program. If this is what you really want as your access pattern, then you have the following problem. So essentially the number of global memory transactions earlier for this entire operation of executing this line of code every warp was spending 2 global memory access.

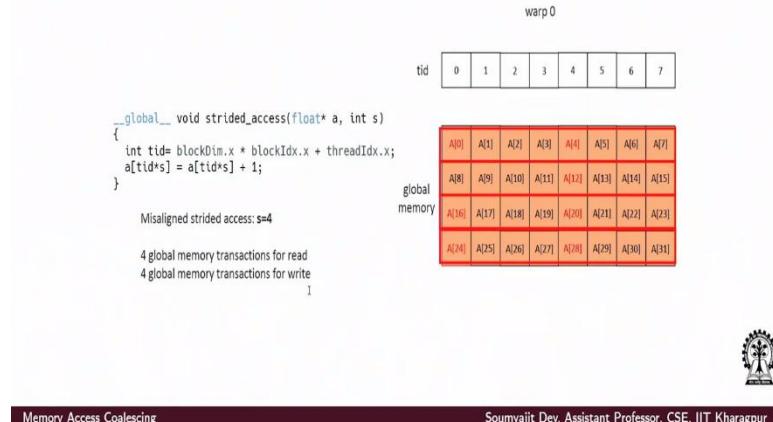
But now instead of 2 you are spending 4 global memory accesses right. So immediately, you have lot of slow down in terms of execution time of each of the warps. So I hope this point of access alignment now becomes clear with this example of offset. So just to summarize again we showed the earlier example where all the accesses are nicely aligned because we have the warp width multiplied by 4 bytes.

So that is the definition of the global memory transaction and that is essentially one of the important points here that the warp would request 32 aligned 4 byte warps. So each of the transactions would happen in spaces of these 128 bytes considering a warp width of 32. But whatever is the warp width in your system multiply that by the requirement of the integer storage that is 4 bytes.

So that is basically the unit of global memory transaction. All the global memory transactions are essentially aligned with that amount of width here. So whenever I am doing this transaction, I am getting everything in a single transaction of the global memory. Whatever is the requirement here from A[0] to A[7] because they have a nicely aligned access here. However whenever I have a offset based code that becomes a problem here right.

Simply because the accesses have to be aligned with the warp width. So here in this case for this kind of arrangement, I have the first global memory access for the warp 0 that cannot form a global memory transaction starting from here and looking up to this. They have to be aligned at the warp width multiplied by 4 number of bytes. That is why it will have to make two transactions the first transaction will bring in this many data, out of which this many would be useful. Again the first transaction would bring this many data out of which only this many would be useful. The second transaction would actually bring in this entire row out of which 8 would be useful. So 2 global memory transactions. I hope that provides the summary of this thing. But overall what we can see immediately is the performance loss here and this is how it will keep on going on. Now consider a different type of access pattern. So in my earlier access pattern it was all with respective to offset. I am doing a strided access here. (**Refer Slide Time 26:54**)

Coalescing Examples: Strided



Memory Access Coalescing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now what is defined as a strided access? So this is another kind of access pattern which we often see in our programs. Essentially all I am doing is there is that there is another integer parameter s and whenever you are accessing the locations in the memory, you are multiplying the thread id with s . We can see such patterns in very simple programs that we write. So this is one of the patterns i.e. a strided access. What would it really mean?

Consider again our simplistic example with warp width 8 and global memory accesses happening in 8×4 byte width. Of course those accesses are aligned with this width. So consider that value of s is 2. Now if value of s is 2, I can see that for warp 0, the total number of accesses are actually spread across 2 rows here in our picture. So for the thread ids 0 to 7. I am going to actually access location $A[0]$. I do not want to access location $A[1]$ because thread id 1 is going to access $A[2]$. Then thread id 2 is going to access $A[4]$, thread id 3 is going to access $A[6]$ and like that it continues.

So in this case also if s is equal to 2, the number of global memory transaction required for executing these instructions required 1 read and 1 write with respect to the program. But it will actually translate to two global memory transactions for the read and the two global memory transactions for the write operation. But what happens if I increase this? Consider s is equal to 4. Now things are further catastrophic.

For warp 0 again considering 8 threads inside the warp look at tid 0. It is looking for accessing A[0]. What about tid 1? Since s is equal to 4, it is going to access A[4]. So overall now the access is spread across this entire memory block right. So for each of the strides, each of them are accessing a position shifted by 3 from the earlier thread . So you are essentially accessing A[0], A[4], A[8], A[12], A[16] ... A[28].

So essentially since the accesses are aligned with the restriction of warp width times 4 number of bytes. So essentially for covering this entire segment of the global memory you would need how many? You would need 4 global memory transactions for the read and 4 global memory transactions for the write. So as you can see this is from our performance perspective for the program this is going to be an impediment I would say.

Because whenever you are going for a strided access you need more number of global memory transactions for both read and write and strided access is something that we often do in our normal programs. So that is not something that we can do away with. So we have to figure out ways in which the program may be written in an alternate way such that I can create a equivalent program but the accesses are intelligently managed.

So these are the important performance impeding factors whenever we are writing code for the GPU. And you have to write programs where you have a variant of the program which actually minimizes this access or may be it helps in coalescing the accesses. That means the program is such that the warp actually is looking for accessing consecutive memory locations which can fit inside a single memory transaction. This is known as memory access coalescing. With coalesced access you can reduce the number of global memory transactions and that is going to be a very important performance factor for your GPU programs. With this we end this lecture. Thank you.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 04
Lecture No # 20
Memory Access Coalescing (Contd.)

Hi. Welcome to this lecture on GPU architectures and programming. In the last lecture, we have been providing you with an introduction on how GPU memory accesses are managed and what is the impact of the way in which it is accessed with respect to performance of your program and how these performance factors changed with different variants possible for access expressions inside your program.

And of course whenever we are talking about performance, you need a metric through which you can actually measure the performance of your program. So even if my metric is execution time how do I really measure the execution time of a GPU program? For measuring execution time of programs one standard way is to profile the execution of the program. So what is profiling of a program?

Essentially there are profiling softwares through which you can monitor the execution i.e. it will simply monitor some hardware counters present in your architecture and you can actually timestamp the execution of the program. I can either use different profilers for profiling the execution of my program. Or I can use modern programming languages which provide a lot of profiling primitives.

So I do not need a separate profiler. I can gather the execution time instants of different segments of the program, which will essentially make use of the hardware performance counters present in the architecture. So these programming primitives will actually tell me when some part of the program actually started and when some part of the program actually ended. And by just measuring the difference, I can simply point out what is the exact execution time for that segment of the code for that architecture.

Now of course, we can see that this is an important scenario. Because, finally we want efficient code to be delivered and efficiency of the code is a function of lot of things in terms of what is the memory architecture design of the system, how many threads are executing, what is the global memory transaction width and all the parameters that were discussed earlier. So let us first figure out how a CUDA program can be profiled using programming primitives.

(Refer Slide Time: 02:53)

Profiling

- ▶ Profiling can be performed using the CUDA event API.
- ▶ CUDA events are of type `cudaEvent_t`
- ▶ Events are created using `cudaEventCreate()` and destroyed using `cudaEventDestroy()`
- ▶ Events can record timestamps using `cudaEventRecord()`
- ▶ The time elapsed between two recorded events is done using `cudaEventElapsedTime()`

So in CUDA the event API provides you with profiling primitives and there are several types of CUDA events they are available under this type `cudaEvent_t`. Now events can be created using a function `cudaEventCreate()` and created events can be destroyed from its scope. I can just destroy the event using the other function `cudaEventDestroy()`. These event variables can be used for recording timestamps by using other primitive known as `cudaEventRecord()`.

Now of course for getting the difference between these recorded times, again there is a API function which is called `cudaEventElapsedTime()`. **(Refer Slide Time: 03:47)**

Driver Code: Offset Access

```
cudaEvent_t startEvent, stopEvent;
float ms;
int blockSize = 1024;
int n = nMB*1024*1024/sizeof(float); //nMB=128
cudaMalloc(&d_a, n * sizeof(float));
for (int i = 0; i <= 32; i++)
{
    cudaMemset(d_a, 0.0, n * sizeof(float));
    cudaEventRecord(startEvent);
    offset_access<<n/blockSize,blockSize>>(d_a, i);
    cudaEventRecord(stopEvent);
    cudaEventSynchronize(stopEvent);
    cudaEventElapsedTime(&ms, startEvent, stopEvent);
    printf("%d, %fn", i, 2*nMB/ms);
}
```

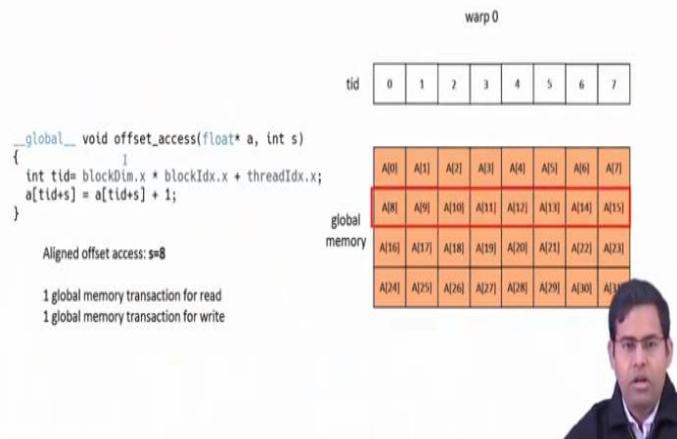
Source:

<https://devblogs.nvidia.com/how-access-global-memory-efficiently-cud/>

So we will just see some example program where these things can be put into good use. So consider the offset access example program that we looked into earlier. So just to remember what it really was.

(Refer Slide Time: 04:02)

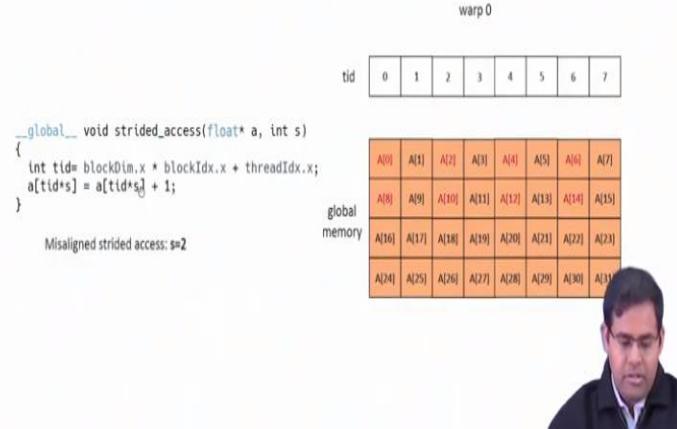
Coalescing Examples: Offset



Let me just walk back to that earlier program on offset access. This was our program. We are trying to run this small piece of code with an offset parameter s . So all we want to do is that I want to check what is the execution time of this kernel on some specific GPU and from our knowledge we have understood that if I keep on changing the value of this, definitely the performance is going to change. Depending on whether I am doing offset access, there should be some pattern of change in the performance.

(Refer Slide Time: 04:43)

Coalescing Examples: Strided



I can be doing strided access which is the other variant of the program. So this is strided access here. There would be some other kind of impact on performance. We want to see that how such things can really be measured. So as we have discussed this CUDA event API is providing me with these functions `cudaEventDestroy()` and `cudaEventRecord()` and finally I can measure the elapsed time using the `cudaEventElapsed()` function.

So let us now have a look into sample program here. So the first thing we are doing is that we are defining 2 variables whose type is `cudaEvent_t`. The variables are `start_event` and `stop_event` and I am going to use them to record timings at 2 different instances which is of course the start and end time. So here we are writing a small driver program for the host side. So essentially the **for** loop through which the loop is going to run for these many times, in every iteration of the loop what am I really doing is that I am putting in `cudaEventRecord()`.

So what is that? If you look into our earlier instances, that this `cudaEventRecord()` function can be called to make one of the CUDA event type variables to start recording the timestamp. That is all we have done. So we have defined this CUDA event type variables and using the CUDA event record primitive, I am starting the execution time from this time point. Immediately after this starting point for recording I am launching the kernel offset access with the parameter N.

So essentially I am launching N number of threads and it is been divided into N / block number of blocks with each block containing block size number of threads. So the offset code is being launched and it is being passed with this parameter d_A. And now if you look into the earlier code of offset access, this second parameter essentially is the offset value.

So in this function, I am going to launch this kernel multiple number of times inside the loop. In every iteration I am changing the offset values. So essentially I am trying to launch this function multiple times, each time with a different offset immediately after the kernel execution finishes and the GPU is notifying the host that it has finished. I have this CUDA event record with the stop event.

So this function cudaEventRecord() will start recording the timing with the start event i.e. essentially from this time (before the kernel is launched). Then this counter stop is providing me the time point for when the kernel is going to finish right. And then I would need a cudaEventSynchronize() for the stop event, because essentially these are all asynchronous calls and finally I need to synchronize this point.

So essentially using the start event monitor, I have started the recording of time and at this point using the stop event monitor i.e. the CUDA event I had started recording the time from this point. So essentially from these two, if I provide these 2 different monitors to the other call cudaEventElapsedTime(). We can see that monitoring starts from these 2 different points i.e. they are doing the time event recording from these 2 points.

This function is provided with this 2 event monitors which have been recording all the events starting from these 2 time points. This function will actually go through whatever is there in their statistics, because they are doing a recording of events that have been going on from different start times. So this function will take care of those differences in start times and record that difference in the first argument variable.

So I hope the point is clear. I can use this data structure to monitor several kinds of architecture phenomena. But in this case, I am only interested in the difference of the start times of these recorders. That specific information can be provided to me by using this function which is doing the work of actually looking into the recording events that have corresponding start times for

these two monitors and providing me just with whatever I am interested. In this case, this is the total elapsed time between the kernel launch and the kernel's execution finish.

(Refer Slide Time: 09:42)

Driver Code: Strided Access

```
cudaEvent_t startEvent, stopEvent;
float ms;
int blockSize = 1024;
int n = nMB*1024*1024/sizeof(float); //nMB=128
cudaMalloc(&d_a, n * 33 * sizeof(float));
for (int i = 0; i <= 32; i++)
{
    cudaMemset(d_a, 0.0, n * sizeof(float));
    cudaEventRecord(startEvent);
    offset_access<<n/blockSize,blockSize>>(d_a, i);
    cudaEventRecord(stopEvent);
    cudaEventSynchronize(stopEvent);
    cudaEventElapsedTime(&ms, startEvent, stopEvent);
    printf("%d, %fn", i, 2*nMB/ms);
}
```

Strided

Now the similar thing I can also do for a strided access. I will just change the code from offset access to strided access. So one thing you have to take care is that the function call should also be changed to strided. This is an error in our part. As you can see for strided accesses, this is our kernel with s being the stride. (Refer Slide Time: 10:52)

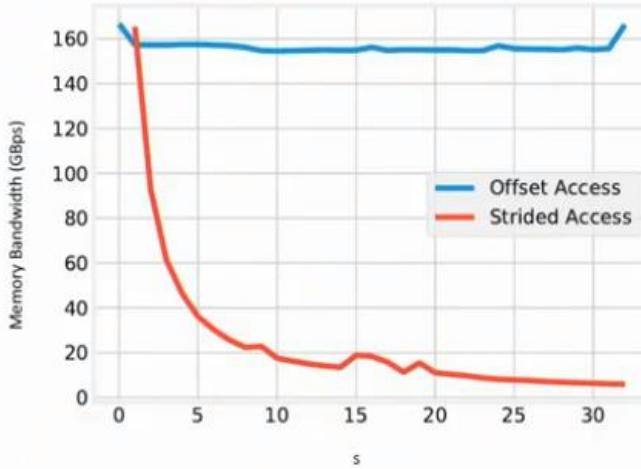


Figure: Memory Bandwidth Plot

So I can do a similar thing for strided access. I just need to change this name so then as you can see that using these 2 programs we are able to figure out how the total execution time of the kernel changes with the change of the offset parameter and with the change of the stride

parameter. We have a nice plot essentially where we are changing the S parameter and we are trying to figure out what is the total memory bandwidth consumed by the system i.e. essentially the elapsed time of the kernel.

When I am changing the stride of the kernel or I am changing the offset, we have got these 2 curves. Just take a minute to look minutely into these curves and figure out why they behave like the way they have been plotted. So let us first figure out what happens in the offset curve. For that let us first go back to the code offset access. So this is how it is. Considering an offset of 0, i.e. when you do not have any offset, for every execution of the kernel we have 1 global memory read followed by 1 write.

Consider an offset value of one. So now you have 2 global memory reads and 2 global memory writes. What happens with the offset value of 2? So with offset value of 2 you will have the data points A[2] upto data points A[9]. Then you will have data points from A[10] upto data points A[17], then data points from A[18] upto data points A[25]. This is how things will go on so essentially whether my offset value is 1 or 2 in both cases the global memory reads will require 2 transactions followed by global memory writes requiring 2 transactions.

Again, I would have the same thing with offset value 3, I am accessing from A[11] up to A[18] but again what is the number of transactions? 2 global memory transactions for read and 2 global memory transactions for write. So in this way as you can see with the increase of S from 0 to 1, I have a bandwidth reduction but after that things remains same with offset values 2, 3, 4.

But what happens when the offset value is same as the warp i.e. when s is 8. What will happen is that for warp 0 instead of accessing from A[0] to A[7] you would be accessing A[8] to A[15]. So again everything is nicely coalesced and you are back to 1 global memory transaction for read and 1 global memory transaction for write. So essentially with respect to offset what is the situation? When you have no offset for this program, you have the best performance. Whenever you have some offset your performance decreases, but it will decrease until the point where the offset is aligned with the warp so that again you have nicely coalesced global memory transactions and that will again increase the bandwidth.

So now with this idea if we go back and see that the total performance reduces first but that is just for the first value from $s=0$ to $s=1$ and then this is all specific to the program. It remains the same. Then again when it will get coalesced access when it gets byte aligned. We are doing all our runs on Tesla K40 GPU. And of course as we know the standard warp size is 32 so immediately at 32, things will be getting byte aligned nicely. The global memory is again nicely coalesced as the offset 0 case. So I have a reduction but then the reduction does not change. I just doubled up the usage for all the threads for that specific program line then again the performance is back to what it was.

But that is not the case for strided accesses. Let us go back to strided code once again. As you can see when you are running the strided code when your stride is 0 i.e. there is no stride so you have again for every instance of this operation 1 global read and 1 global memory write. What happens with stride? So this is the stride which is changing from 1 to 2. Now the number of global memory reads and writes have doubled up. The number of global memory reads and writes have further doubled up with respect to $s = 4$.

So if I keep on increasing the stride factor the global memory transaction the number of transaction is doubling up right. So this would actually lead to drastic reduction in the memory bandwidth provided by the DRAM to this kernel. So this will lead to huge performance loss and if I look at the effective memory bandwidth that I am getting, it is reducing drastically. So the more I increase the stride, the memory accesses are getting even more scattered.

So there would be at some point I will lose all kinds of coalescing and in the worst case, I will have for all thread id's inside the warp, I am doing separate global memory transactions. And that would be the worst case. So I hope that is understood because nothing can be worse than that. I am executing a warp. The warp as 32 threads, each of the threads are requesting locations in the memory which are far apart and they are so much apart that they cannot be brought in a single transaction.

So that is the worst case. After that if I go on increasing the stride it really does not matter because essentially then by that time for each access I am doing a separate global transaction. That would mean for every instance of execution of the warp for some load or store instruction,

I am going to do 32 separate global memory transactions. So that is the worst case performance and that would give me the saturation point in this curve - the lower saturation point in this curve right.

(Refer Slide Time: 18:28)

Using Shared Memory

- ▶ Applications typically require different threads to access the same data over and over again (data reuse)
- ▶ Redundant global memory accesses can be avoided by loading data into shared memory.

Now of course I can use better arrangements of data points for whatever computation I am doing to reduce the number of global memory access. I need to do in such pathological cases. So overall the recommendation is that whenever you are writing your program you need to make good use of your memory segments. You have to understand the GPU's memory hierarchy and accordingly you should write the code so that the access patterns are nice with respect to coalescing.

I mean this is where the role of shared memory is coming because as we have discussed earlier many times inside each SM you have a portion of the memory which can be configured as shared memory or L1 cache. If you tune your algorithm and write the code in such a way that you are doing lesser number of global memory accesses and doing some smart usage of shared memory it can increase the performance of your code drastically.

So although in general the property of applications is that they typically require different threads to access the same data over and over again. So that is what we know from our principle of locality or data reuse. I have a principle of locality which is spatial as well as temporal. So

different threads may need access to the same data or threads may need access to data points which are located in neighbouring regions of the memory.

So the primary thing is since different threads would need access to the same data, if I keep the data in the memory at some point some other thread will require the data. And also if I bring in chunks which is actually done by wide transactions that we always do, the basic philosophy is that whenever we are bringing in the data to the memory, you are bringing data in chunks. It may so happen that in future with a high probability you would be actually reading the nearby memory elements which are already available to you in the cache.

So the shared memory can be configured as partly as cache and shared memory, and the good thing about the shared memory is that all the threads inside the blocks have a consistent view to the memory. So in many cases based on this idea that different threads can actually access the same data over and over again at different time points , I can reduce the global memory accesses by avoiding multiple loads and stores from the global memory.

But rather I put the data in the shared memory and keep it there for collaborative access by other threads at different points of time. So I make one thread responsible for bringing the data and putting it into the shared memory using a shared data type and then those data points can be used by some other data point or by some other threads at different points of their computation.

Now, if this can be done then as we can understand every thread, whenever it is doing a memory reference, if it can get it from the shared memory, then there is an order of magnitude reduction in the access time with respect to the global memory access. And that is quite useful as an optimization. This is one of the primary motivations of keeping a shared memory segment in the GPU.

It allows threads inside the block to collaborate among each other. The data elements brought in by threads in the block can be used by other threads in the block at different points of execution by accessing them from the shared memory, if they are actually made to be resident in the shared memory by the programmer. Now this is something important the hardware does not even decide what should be or should not be in a shared memory. The hardware decides this for the cache by using standard cache principles.

I mean using the cache's read and write policies but for the shared memory access, it is the programmer who has to decide by giving suitable data types i.e. which data point should be resident in a shared memory and which data points should not be resident in the shared memory.

(Refer Slide Time: 22:51)

Using Shared Memory

- ▶ Each SM typically has 64KB of on-chip memory that can be partitioned between L1 cache and shared memory.
- ▶ Settings are typically 48KB shared memory / 16KB L1 cache, and 16KB shared memory / 48KB L1 cache. By default the 48KB shared memory setting is used.
- ▶ This can be configured during runtime API from the host for all kernels using `cudaDeviceSetCacheConfig()` or on a per-kernel basis using `cudaFuncSetCacheConfig()`



Now lets further go deeper into shared memory. How is the shared memory structured? This is something we have discussed again. Just a small recap. So each SM typically has 64 kilobytes of on chip memory which can be partitioned between the L1 cache and the shared memory. Now the setting is typically like this - you have 64 kilobytes. Out of that total 48 is configured as shared memory and rest 16 kilobyte configured is L1 cache.

But it can also be the reverse by doing suitable programming setting. You can make it like 16 KB shared and 48KB L1 cache. As I told earlier, that if it is L1 cache, then it is managed by the hardware just like the cache memory is managed. But if it is defined as the shared memory then the programmer has to manage it through suitable type definitions for the variables in the code.

Now this can be configured during the run time API i.e. given the memory block which is on chip in the SM, whether it should be a shared memory or how much of its should be shared memory and how much of it is should be L1 cache. Essentially it is going to be 2 options that can be actually can be configured using the CUDA API. Now suppose you want to configure it

permanently from the host for all the kernels so the host can technically launch multiple kernels one after another asynchronously.

But if I want a specific setting of shared memory and L1 cache available for all the kernels I can do that using this function `cudaDeviceSetCacheConfig`. So this is the function which will take care of doing the configuration of the shared memory. You can just look into the manuals online for NVIDIA and figure out what are the parameters for this. I can do it in a per kernel basis i.e. I have a host code for 1 kernel. I can configure the shared memory and the L1 cache in one way and then for the next kernel I can configure in a different way.

Of course it depends on what is the property of the kernel, how many threads are getting launched, how the threads are being arranged, whether it is actually useful to have more shared memory or whether it is useful to have more amount of L1 cache. It depends on the programmer's perspective. He has to figure out whether having more shared memory actually helps that specific kernel. Then you would actually like to use this function to configure that memory segment with more amount of shared memory or depending on if you want fast access to decrease the latency and you do not have really control over what really would be used along the variables by which thread, you can actually have more amount of cache.

Just before concluding here I will also like to put in one important point which I have seen earlier. GPUs typically have less amount of L1 cache with respect to CPU's. The primary reason is that first of all I have the shared memory i.e. one segment that I have. So I can afford to have some L1 cache. But the other reason is that in this case of the smaller L1 cache, I am trying to hide the latency of L1 cache by using more threads. So when I compare GPU computing and CPU computing, although I have multi-core, multi threaded CPU cores in a CPU, the number of threads executing parallelly in the CPU are much less with respect to GPU. So objective there is to execute the threads fast enough. One important impediment in terms of executing a thread faster enough is this memory operation.

So I have good motivation for using large caches so that there is a high probability that I can find whatever data is required in the L1 cache and that would reduce the memory access penalty and that will give you better performance. When I am using GPU essentially what I am doing is that

using more, I am actually using lot of threads in parallel. I am launching more number of threads in parallel.

So I really do not care if some of the threads are suffering due to non-availability of data in the L1 cache, because technically I have too many other threads to execute. I have many warps waiting. I have launched more number of warps than I have actual available physical space. So I can just steal that work and execute some other work whenever this work is stalled due to some high latency operation.

So this is how things are managed. They are different philosophies. Since in this case I have more number of threads to manage, if some threads are waiting due to non-availability of data that is fine with me. In this case I can afford to have smaller amount of L1 cache. So with this we like to end this lecture and maybe in the next lecture, we will see a good program example through which we show how the usage of shared memory can be a nice optimization with respect to performance. Thank you.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 05
Lecture No # 21
Memory Access Coalescing (Contd.)

(Refer Slide Time: 00:34)

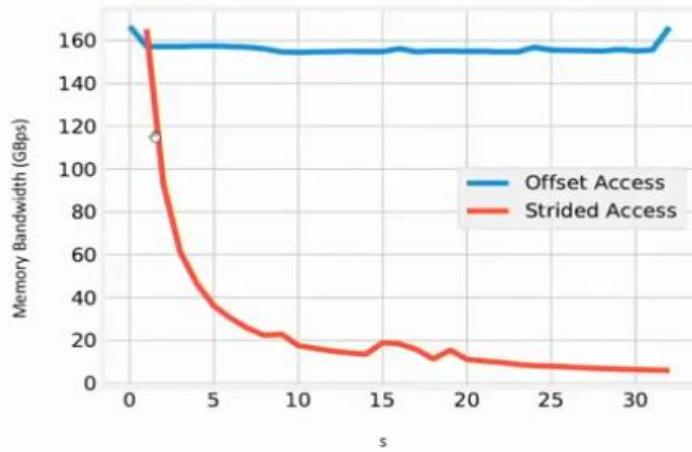


Figure: Memory Bandwidth Plot

Hi. Welcome to the class of GPU architectures and programming. In the last lecture we have been discussing a bit on this usage of shared memory i.e. how the shared memory is configured and in general how global memory transactions are really happening. If you remember the curve we were showing that based on the different ways I am doing access of an array, whether it is offset based access or strided access, we could see that the memory bandwidth that was exploited

i.e. the number of global memory transactions per second that was

Using Shared Memory

- ▶ Each SM typically has 64KB of on-chip memory that can be partitioned between L1 cache and shared memory.
- ▶ Settings are typically 48KB shared memory / 16KB L1 cache, and 16KB shared memory / 48KB L1 cache. By default the 48KB shared memory setting is used.
- ▶ This can be configured during runtime API from the host for all kernels using `cudaDeviceSetCacheConfig()` or on a per-kernel basis using `cudaFuncSetCacheConfig()`



was really getting affected. So after that we also pointed out how to actually configure shared memory and the L1 cache what are the related CUDA comments for doing that.

(Refer Slide Time: 01:20)

Recap: Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int N){
    int i=blockIdx.y*blockDim.y+threadIdx.y;
    int j=blockIdx.x*blockDim.x+threadIdx.x;
    if ((i<N) && (j<N)) {
        float Pvalue = 0.0;
        for (int k = 0; k < N; ++k) {
            Pvalue += d_M[i*N+k]*d_N[k*N+j];
        }
        d_P[i*N+j] = Pvalue;
    }
}
```

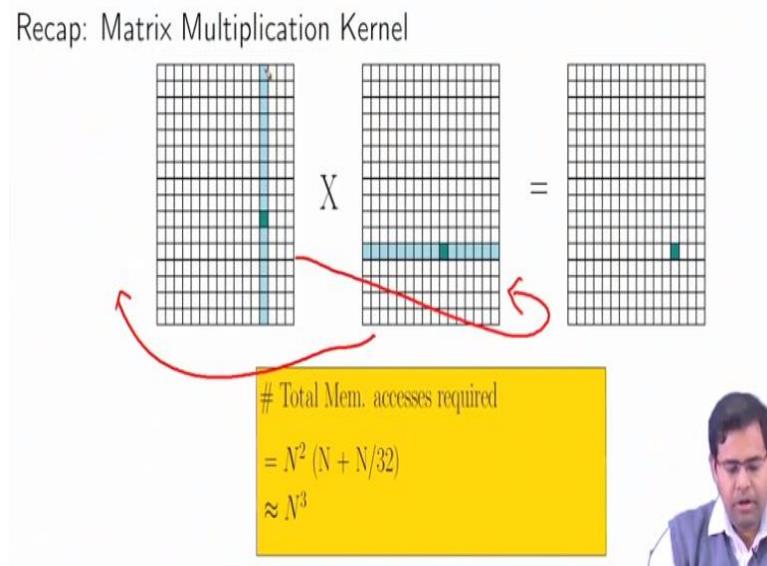
So now we will see how I can exploit this shared memory that is present in each of this symmetric multiprocessors i.e. streaming multi processors and in each of the streaming multi processors how they can actually help in accelerating parallel algorithms by suitable programming techniques. So just as a recap let us consider the common popular matrix multiplication kernel.

So this is the usual code of matrix multiplication kernel which you might recall from our earlier examples. So all that we are doing is there is a if statement which is basically checking whether the i and j i.e. the location of the row and column we are trying to access in the matrix whether they are inside the relevant bounds. By the way, the row and column positions were figuring out again by using the `blockIdx.x` and `threadIdx.x` and `blockIdx.y` and `threadIdx.y` variables.

So for each thread its unique combinations of block id's and thread id's would able to find out a unique i and j . Then you should check whether the i and j are inside the memory range of the matrix and then each thread will get inside this for loop. So as you can see the for loop is running from 0 to $N - 1$. So essentially we are talking about multiplying $2 N \times N$ dimensional matrices.

So the thread would essentially multiply the elements of 1 row of matrix 1 with the elements of the corresponding j th column of matrix 2 and it will essentially sum it up, store it in `Pvalue` and this value would finally be transferred to the target matrix location of the output matrix location which is `d_P` here right.

(Refer Slide Time: 03:24)



So in terms of the figure, this is the per thread activity. So the thread I have launched for accessing i th row and j th column, I would actually prefer this picture in the flipped way basically. So if you look into the access pattern here, all that we are doing is we are figuring out an i and j . So i is the row index and j is the column index. So just to match with it just consider the figure in a flipped way.

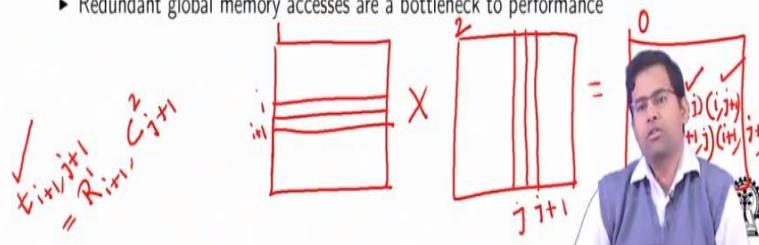
So just think that this is matrix 1 and the next 1 is matrix 2. So essentially the thread will navigate through the i th row and the j th column, pair wise multiply the elements and finally figure out the final output value for the i th row and the j th column of the output matrix. This is the usual way we do matrix multiplication. So what is the total number of threads that are getting launched for computing the output. So definitely the total number of threads used has to equal to the number of elements in the matrix, because each threads is responsible for computing 1 element in the output matrix right.

(Refer Slide Time: 04:56)

Recap Matrix Multiplication Kernel

$$\begin{aligned} t_{i,j} &= R_i^T, C_j \\ t_{i,j+1} &= R_i^T, C_{j+1} \\ t_{i+1,j} &= R_{i+1}, C_j \end{aligned}$$

- ▶ Number of threads launched is equal to the number of elements in the matrix
- ▶ The same row and column is accessed multiple times by different threads.
- ▶ Redundant global memory accesses are a bottleneck to performance



But observe something important here. With respect to the memory accesses performed by the threads. So the same row and column is accessed multiple times by different threads. Why would we say that. So if we draw up the simple figure here. So for computing element i, j , I would access the i th row entirely and also the j th column entirely. So if I consider some thread i, j so essentially what is it doing?

It is essentially accessing row i . Let me say row $i1$ for matrix 1 and this is matrix 2, and this is matrix output and column j of matrix 2. Now let us look at entity $i, j + 1$. So for that, I would have another thread. So what is the row and column that this thread would be accessing. So $i, j + 1$ has to be sitting in the same i th row side by side with i, j . So the thread which is responsible for computing the i th row, $j + 1$ th column element of output matrix . I call it $t_{i,j+1}$.

So essentially what it is going to access? It is again going to access the i th row of the first matrix and the $j + 1$ th column of the second matrix right. Now let us consider some other entity. Let us consider the point $i + 1, j$. So what are the different memory locations that the thread should compute for this element or will be accessing? So let us call this thread $t_{i+1, j}$. So essentially as we can see the first one accessed i th row and j th column. The second one has accessed i th and $j + 1$ th column .

And for this so I am essentially doing an access of $i + 1$ th row and the j th column. So let us again take another example just for completeness. I have then considered 4 nearby locations here. So $i + 1$ th row and $j + 1$ th column. So then I would be having this new thread which has been launched for $i + 1$ th row and $j + 1$ th column. That would be accessing frp, first matrix, $i + 1$ th row and for the second matrix the $j + 1$ the column. Now see what is really happening. So what is the total number of row's and columns that we have accessed ?.

We have accessed 4 total number of rows and columns that is to be specific 2 rows and 2 columns. But as we can see there is lot of common access. For example, the i th row has been accessed separately by 2 threads who have been computing for this position as well as this position. So the i th row has been accessed by both this threads similarly the $i + 1$ th row has been accessed by this thread and this thread right.

So I would say there is a performance loss. Why? Because 1 thread has already accessed the data from the global memory. Now again I am asking another thread to access the same data from the same location of the global memory. Why? For doing the computation of a different position. Now of course, one way we do that is that some of this will saved in the cache. We are talking about GPU where the caches are really small right.

So considering that we are looking into performance, our programming for very large matrices for saving some redundant i.e. duplicated accesses from the global memory, this will be a huge performance issue. Why? Because every global memory access is extremely costly, an order of magnitude costly. So the warps which will essentially access this global memory would be stalled for a lot of time right.

So if I am trying to do the matrix multiplication in a nice way, this will definitely not be the good algorithm for doing that. So again just to give an example, we can see the i th row is getting accessed how many times over all in the computation. Of course for in total, in the output matrix there would be N elements in the i th row. Using this naive algorithm, I am bringing elements from the i th row into the memory n number of times separately by each of the threads.

So that does not make sense. It would have been much better if data brought in from the global memory would have been useful for the computation by some other thread. As you can see that each thread is bringing its own row and own column from the global memory. But if it was the case that whatever the threads $t_{i,j}$ brings can also be used by the threads $t_{i,j+1}$. Now that would have been much nicer right.

Because then both of them bringing the data of the same location multiple times does not really make sense. They may not be doing the computation at the same time - the $t_{i,j}$ and $t_{i+1,j}$ or $t_{i,j+1}$. Threads $t_{i,j}$ and $t_{i,j+1}$ may not be active together at the same time. So in that case, although since they are not active together in the same time where will the data be resident?

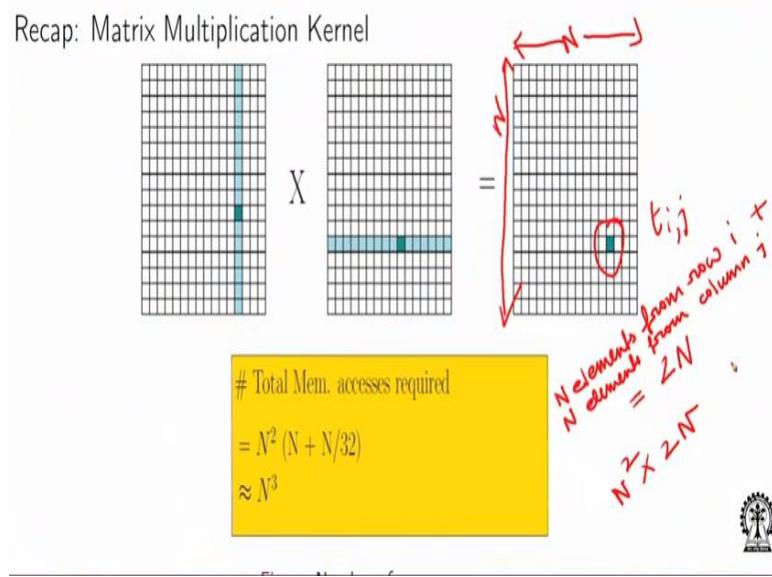
So that also is an important thing. So essentially for this purpose we can exploit the shared memory that is located in the same streaming multi-processor. As we know that the shared memory is offering a consistent view to all the threads inside the block. Hence threads operating inside the block can actually collaborate among each other. Let me put it in other way around. The threads operating inside the block can bring in data in such a coordinated manner that the data required by one of the threads in the block is also useful to some other thread inside the block, if they are brought from the global memory and they are made resident in shared memory location.

Why? Then each of the threads inside the block would be accessing the data from the shared memory location and that access will be much more faster. So the point to be noted here is that we should try to minimize the re-fetching of data multiple times through global memory loads by different threads for doing the similar activity on different data points, but using the same locations from the global memory. The alternate option is make the threads working inside the

block collaborate among each other by bringing the data from the global memory and putting the data in the shared memory.

Why? Because as we have already discussed that the shared memory view is consistent for all the threads inside a block. So they can actually use the data for doing their own part of the matrix multiplication computation and while doing that instead of doing global memory loads they will be actually loading data from the shared memory which is much faster. Because the shared memory is on chip and it is located inside the SM and is much nearer to the scalar processors or the SP cores.

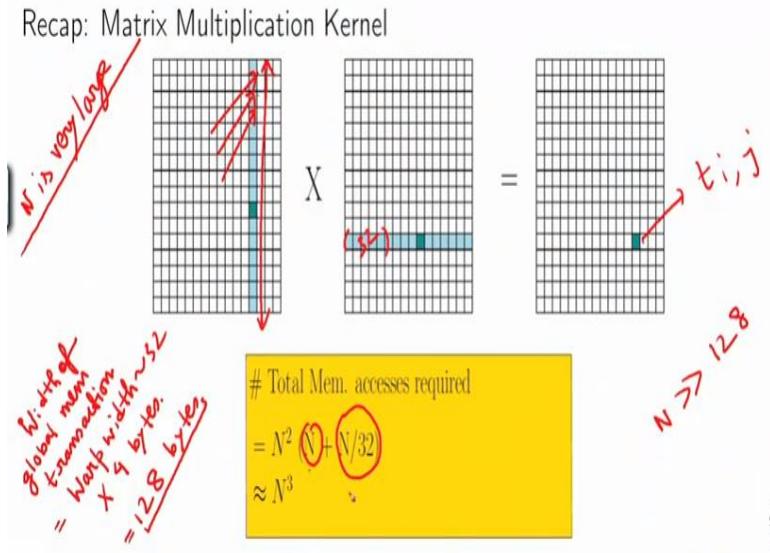
(Refer Slide Time: 15:34)



So just consider the picture here. What would be the number of memory access required overall? If I am doing a nice matrix multiplication computation i.e. if I do not consider the global memory coalesced accesses then essentially what is going on for doing computation for each location in the output matrix? I need access to 1 column and 1 row of data right.

So I need access to N elements from a row and I need access to N elements from column so that is $N+N=2*N$. The thread which is doing this computation would require access to $2N$ number of elements from the global memory. So N elements from a row. So this is the total number of memory elements that 1 thread would access and overall how many threads do I have? So overall I have N^2 number of threads. So this is the total. However the number of elements accessed is not equal to the total number of global memory transactions that would be going on right.

(Refer Slide Time: 17:51)



Just recall the knowledge of memory access coalescing. Whenever I am doing access for a column, the number of global memory accesses I am considering for that N is very large. So since N is very large, the access of any column location and the access of the previous or the next column location is never going to be coalesced. That means they will never be brought from the global memory by a single global memory transaction. Why is it so? Let us recall.

What is the width of the global memory transaction? As we have discussed earlier, since the warp we are considering for most GPUs is 32, the global memory transactions would essentially bring in 128 consecutive bytes in 1 byte aligned access. Again, I would just repeat the per byte aligned accesses we discussed earlier. So as long as this N is going to be large than 128 bytes , these accesses of these consecutive locations in the column will all be separate global memory transactions.

So for computing for 1 location here, for N of the column elements, I would land up with N global memory transactions. However that will not be the case for the row. Why? Because in the row all these elements in the final DRAM. They are sitting side by side right. Because we all know that we are following a row major setting in the DRAM. So all of these elements are going to sit side by side. They are consecutive elements separated by 4 bytes considering an integer matrix right.

So out of this N , the consecutive 8 elements would be brought by a single transaction. So assuming that these accesses are all byte aligned i.e. N is divisible by 32, the number of accesses we would have is $N / 32$. Now of course if N is not divisible, essentially it would be an $N / 32$ and then you take the remainder and do one more access or something like that. I mean we can also assume here for simple purpose it is divisible. I have got these many accesses for 1 element.

So the total number of accesses is N^2 times this. Now of course this is the actual figure i.e. the total number of global memory transactions would be this considering 32 as small. If I am considering a very large matrix, a significantly large matrix, I can approximate this as N^3 but of course that we will hold for a very large N .

Now we would like to exploit 2 things here. First of all the fact that consecutive access in the row are going to be coalesced and the other fact that we have discussed earlier that I can make use of the shared memory by threads inside the block collaborating among each other and in that case many of the global memory loads would be replaced by the shared memory loads. And that would provide a lot of acceleration in terms of the execution of the code.

(Refer Slide Time: 22:23)

Matrix Multiplication Kernel using Tiling

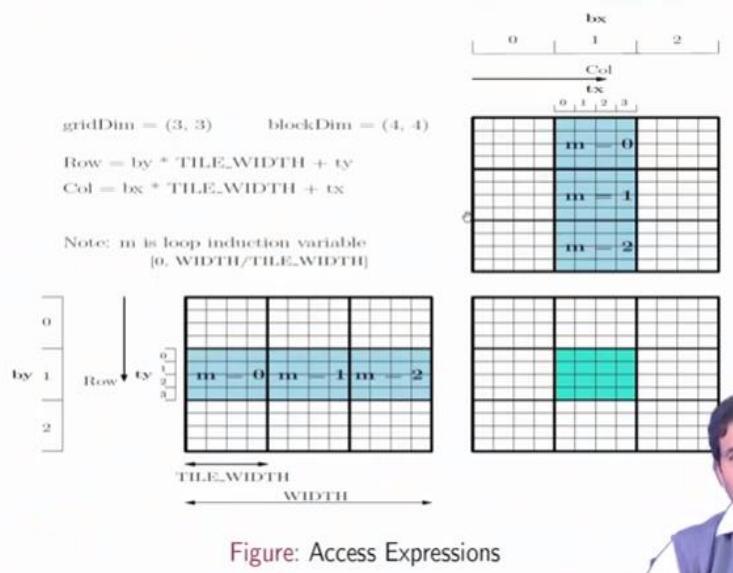
An alternative strategy is to use shared memory for reducing global memory traffic

- ▶ Partition the data into subsets called tiles so that each tile fits into shared memory
- ▶ Threads in a block collaboratively load tiles into shared memory before they use the elements for the dot-product calculation

So if I do this optimization then what we get is an algorithm for doing matrix multiplication using tiling. So in summary, we would say that the strategy we are going to use shared memory which will reduce the global memory traffic. So what we will do is we will partition the matrix data space into subsets which are known as tiles in such a way that each tile fits into the shared memory.

So considering that I can fit each tile into the shared memory. That is how we will design the tile size. So we were discussing earlier the threads inside the block will then collaboratively load tiles into the shared memory before they use these elements for doing the dot product calculation here.

(Refer Slide Time: 23:18)



So that would lead to a kind of much more complex algorithm. We will talk about that algorithm in the next lecture. For the time being let us get comfortable with the idea that the shared memory can be used collaboratively between the threads inside the block. And I am also going to use another important parameter here which is that whenever I am doing global memory loads I am always able to do a coalesced access of the elements in the row. With this we will end this current lecture. In the next lecture we will go deeper into the algorithm. Thank you.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 05
Lecture No # 22
Memory Access Coalescing (Contd.)

Hi Welcome to the course on GPU architectures and programming. In the previous lecture, we have been discussing about our simple idea of matrix multiplication using a GPU and how that can be accelerated by using some specific constructs like the shared memory based optimization and also using the idea of global memory coalescing.

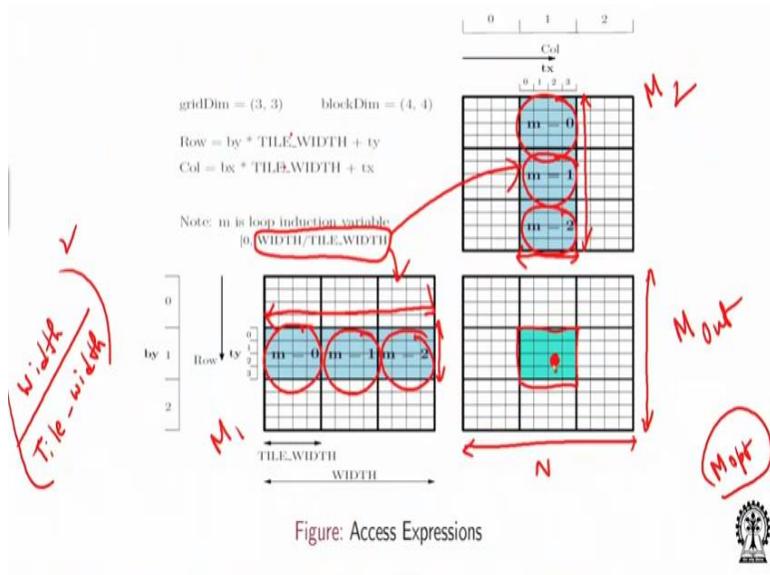
(Refer Slide Time: 00:5)

Matrix Multiplication Kernel using Tiling

An alternative strategy is to use shared memory for reducing global memory traffic

- ▶ Partition the data into subsets called tiles so that each tile fits into shared memory
- ▶ Threads in a block collaboratively load tiles into shared memory before they use the elements for the dot-product calculation

(Refer Slide Time: 01:04)



So based on that let us look into this example. Now first thing we will do is we will define a notion of tile. Essentially a tile will be same as a block size. So we will be considering 2 dimensional blocks. That is why we will define the grid i.e. the number of threads I am going to launch is still N^2 which is the dimension of the matrix but the launch parameters I will be doing will be carefully selected.

So what I will do is that I will define as we have discussed something called as tile which is essentially equal to the block size. So in 2 dimensions I will decide the size of the tile based on the size of the shared memory. We will soon see that how can be done. For now, let us consider that the tile size is the block size and this is an example picture. We are showing that grid is defined as a collection of the 3×3 that is 9 number of blocks distributed in 2 dimensions and inside each block, I have a 2 dimensional arrangement of threads in 4×4 .

This is an example where essentially I am holding N as 12. This is of course the simplistic example. As you see that it maps nicely with the original idea that we are considering 2 dimension matrices. So we will definitely consider the blocks in 2D and also the grid. Threads also inside the block are in 2D i.e. the blocks are also packed in 2D because that would give me a very nice access expression.

Now how do I figure out some specific row and some specific column for the given thread? Again that would be quite easy. So essentially here we are setting the block dimension in the x

and y axis both equal to the TILE_WIDTH variable. Now of course it will depend on the dimension of the matrix here. So the TILE_WIDTH variable is going to give me the block dimension in the y direction and similarly in the x direction.

So if you replace this blockDim.y and this blockDim.x variables, that gives me the usual access expressions for the row and column for a specific location in the matrix. Now how do I figure out that in which tile I am located? So I want a loop to decide on which tile to load or which tiles to access. So essentially the basic principle inside this computation is as follows. Threads computing inside a tile would do collaborative loading and computation.

So earlier in our naïve algorithm let us say any 1 thread the i,j thread was bringing data from the i^{th} row and the j^{th} column, multiplying and accumulating them to compute the i,j^{th} location. But now we are saying: No, we will do something different. Instead of this entire tile or the corresponding block, whatever the threads are there, they would do collaborative access and collaborative computation. And when all the threads computing for this block finish, they are done with computing all the elements for this tile in the output matrix. Now for doing that what is the region of access required in the 2 input matrices? So of course as you can see that for a single row, it would have been a for a single location. It would have been a single row and single column. But a region would actually spread for all the locations inside this shared output tile. I would need access in M1 in this entire region right.

And similarly here for M2, I would need access in this entire region. But the issue is not still fixed. How am I going to do some optimization and decrease the number of loads that are going to happen in the global memory i.e. the key thing they are going to do right. Just another point we would like to make here. Let us suppose somehow we figured out how to optimize computation of things inside this output tile.

But then let us call it some method. I have some method to optimize. If I am using this method, how many times am I going to call this optimized method? How do you figure out that? Earlier I was computing for each location and that I was doing M^2 number of times. But now of course if I have think that if I am using this optimize method for doing computation for one tile here, then the number of times I am going to call this method would be the total number of tiles here.

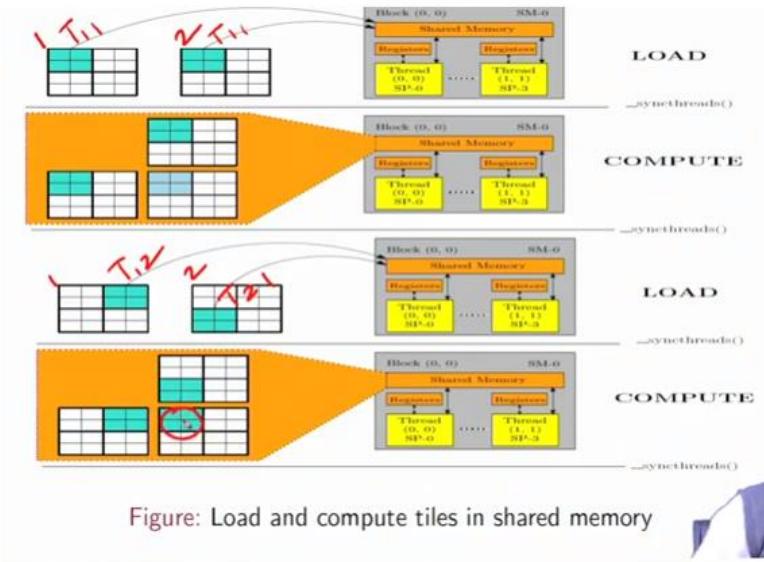
So what is the total number of tiles here? $(\text{WIDTH}/\text{TILE_WIDTH})^2$, because as we can see in this example that since my width will be divided by the tile size, considering width and length both same here. So I need to access that many tiles here on the x side and the y side. You multiply them and so you have to access these many tiles right. So this M_{opt} would be accessed this many times.

Now let us try and look into the functioning of this optimized method. So first of all, I have broken down the matrix into this kind of coarse grain distribution of blocks which we are now calling them as tiles. Now observe that for computing this output tile, I need a loop which should iterate through M_1 and access 3 tiles because this is the region which needs to be accessed. This region can be nicely broken down into 2 types.

As per this picture in general I can say that it can be nicely broken down into this $(\text{WIDTH}/\text{TILE_WIDTH})$ number of tiles. So for computing this output tile, I need to access this $(\text{WIDTH}/\text{TILE_WIDTH})$ number of tiles from input matrix 1 and similarly I need to access this $(\text{WIDTH}/\text{TILE_WIDTH})$ number of tiles from the input matrix 2 right. However, I have to figure out what is the location of each of these tiles as as I have to load these tiles into the memory and then I have to do some computation right.

So I can say that all the threads which are computing for this block - the output tile, the first thing they have to do is that they have to load. Let us say I set the loop iterated variable name as 0 and I load tile 1 here and tile 1 here and then I do some computation using them. Then I load the next tile from here and the next tile from here. I do some computation using them. Then I load the next tile from here and the next tile from here and again I do some computation using them. And finally all these threads inside this block, they are ready with the output. So I am loading in tile sizes and I am doing some computation using the data. Then again I am loading in tile sizes, I am doing some computation using the data and that is how I am going on. Assume that for the time being that this is going to work.

(Refer Slide Time: 10:14)



So if this idea is really going to work then I would essentially have a sequence of loads followed by computes. again loads followed by computes right. So essentially I am using a set of threads in the same blocks to load to tiles, so I have effectively loaded this tile and this tile and did some computation with them. And in that way I am making this progress. So I have loaded these 2 tiles and made some computation.

Then again, I have loaded the next tile. This one and this one and they have been highlighted here. And next tile this one and this one and again I have done some computation. So this picture is show using smaller size. Let us say I am showing instead of having this kind of 3×3 number of total tiles in this picture for representation purpose, I have 2×3 number of total tiles. Sso I am trying to show that suppose that this method works. Then essentially the threads which are doing some computation for this region, they are going to collaboratively load data from this tile of matrix 1 and this tile of matrix 2 to do the computation.

So this is the load space and using the loaded data in the shared memory they will do some computation. Then again is the next load phase. So this is matrix 1 and this is matrix 2. So they are loading the 2 tiles - T_{11} and T_{12} . They have put them into the shared memory and did some computation then they are loading T_{12} and they are loading tile T_{21} . Then again they are doing some computation and with that, the final result is ready.

But the question is what is the good thing here? How does this really help? Still we are assuming that this sequence of load computes are going to work. We will see that why it works.

(Refer Slide Time: 12:26)

Matrix Multiplication Kernel using Tiling

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {

    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
```

So suppose that works. Then we would write the kernel like this. So inside the kernel I would define 2 shared memory locations each which are of the TILE_WIDTH x TILE_WIDTH. Why? Because assuming that I am going to do this kind of load compute, I am loading these 2 types from these 2 different input matrices into the shared memory. The compute phase will be done in the shared memory and we will see the advantage of it.

So the 2 tiles have to be loaded by the set of threads who are working for this part of the computation. So the set of threads are going to first load this tile from matrix 1 and this tile from matrix 2 into the shared memory. So I define a share memory location each of the size that should be able to hold 2 of this tiles together. Then I get the block id here and thread id here in this smaller case variables.

(Refer Slide Time: 13:31)

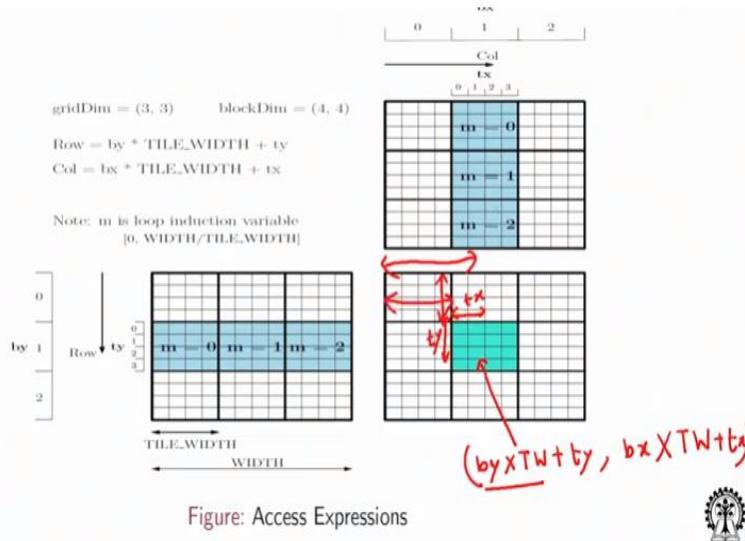
```

int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;
float Pvalue = 0;
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
    Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
        Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
}
d_P[Row*Width + Col] = Pvalue;
}

```

And then I have to perform the actual loads right. So to perform the actual loads, I first figure out the row and the column index for the locations. So essentially what does this mean? I am essentially putting in here two variables - Row and Col. So if you just take the example of what is that? What do I really get out of this blockIdx.y and threadIdx.y of any thread?

(Refer Slide Time: 14:09)



So any of the threads that are trying to figure out what is the location for this, it is going to load $\text{blockIdx.y} * \text{TILE_WIDTH} + \text{threadIdx.y}$, $\text{blockIdx.x} * \text{TILE_WIDTH} + \text{threadIdx.x}$. So what is this block id times the tile width. So you come up to here and then the thread id. Why I am really multiplying the block id with the tile width? Basically that helps me if I multiply this by tile width, I can get to this position here.

So I get to this much and then I shift by the TILE_WIDTH. Similarly, with $bx * TILE_WIDTH$, I get to this much and then I shift with tx right. So this tells me to figure out for each thread which locations of the input memory it is going to access. As we have discussed that all the threads who are working on these output tiles, they are going to bring data collaboratively from 2 of the input tiles. Let us say this one and this one , or if you look at the simplistic picture from this tile and this tile. First of all you would figure which of the locations of the threads of the input tiles they are going to access.

So for that, they simply use the row column indices and using this row column indices they would compute the corresponding location in the original input matrix. So essentially as we can see the row will get multiplied by the entire width of the matrix. So that gives me the position in the row and then with the loop iteration variable, the tile width I can get into the tile with the tx position.

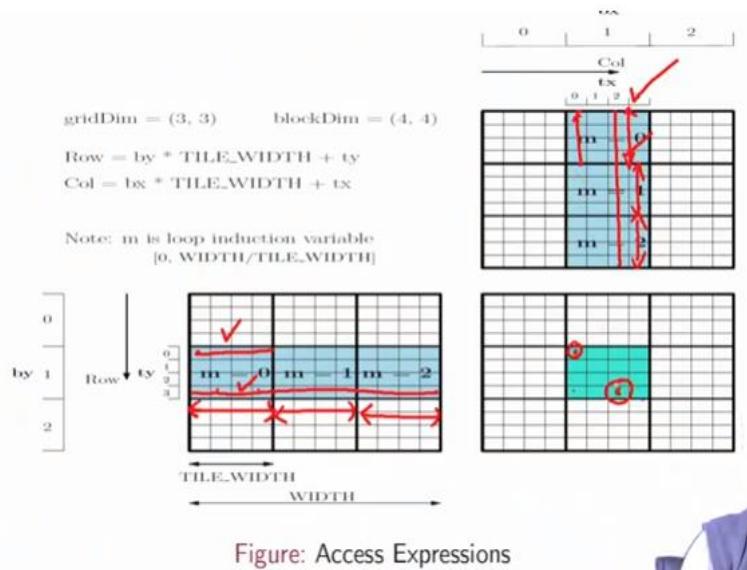
So I hope this gets clear now. So first point is since I am going to do this inside a loop in the first phase, I am going to access from these 2 locations. So for this location essentially, what are the positions that this thread are going to access? So in the first phase of the load compute, it is going to access data from here and here. So when this loop runs for its first iteration $m=0$ so row times width plus tx .

So you have already figured that the row times width gives you that position here that which row and then you shift by tx amount i.e the amount of shift you have here. I hope this is clear. Because this is your tx , you just shift by tx inside the first tile for the corresponding row. So in that way the thread has figured out which data to bring from matrix 1 and similarly using the other expression it will figure out which data to bring from matrix 2.

So as we have already pointed out that we got the exact data points that the thread will bring in and finally when all the threads corresponding to the tile have executed this code what do I really have? I have this situation that for all the threads in this tile, they have brought in these 2 data points from the original global memory of the matrix to the shared memory right.

Now in these tiles, I have got these two data points for the threads working here and for the threads working from the other tile. They will be doing collaborative load from some other locations right. So these threads have done this part of their work. They have first figured out that they belong to which tile and then they have brought in data collaboratively in parallel from here and here. And then they are going to get into this loop through which we will do a partial sum computation.

(Refer Slide Time: 19:29)



So what does this mean? So if you look into this picture. Finally what is going to be the output for this location? Let us take any random location - the one which we are talking about. So that would require me to do an addition, a multiply for things here and here. What am I really doing is that I am breaking this multiply into 3 parts. In the first part, I load things from here and things from here and do the multiplication in the second part of the computation. I will load things from here and things from here and then do the multiply. At the last part of the computation we will do this and this and multiply and get everything done.

So the threads are working collaboratively here to bring in this tile. So essentially after bringing these 2 tiles in shared memory what they will be doing here is that they will be doing their part of the multiply at operations for only these parts i.e. for this thread, this was responsible for bringing data from this location and this location. But parallelly, the other threads for the other locations have bought in the data points and store them in the shared memory which this threads requires right.

So this thread will now actually use data points being brought in by other threads and do the multiply ad operations this thread is suppose to do but upto this point. Why? Because I have not brought in the data for the other tiles into the shared memory. This is the main reason. So similarly what is happening is that while this thread is doing this part of computing for this location, all the threads in the tile are doing their part of the computation for various other locations.

For example if I consider the thread corresponding to this point. It had brought in this data and this data and after bringing in those data points it is traversing this much and this much and doing a multiply here right.

(Refer Slide Time: 22:00)

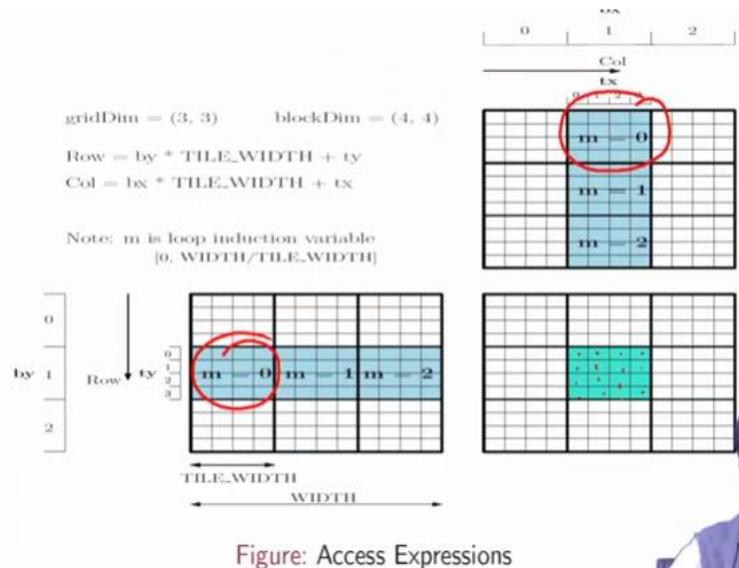


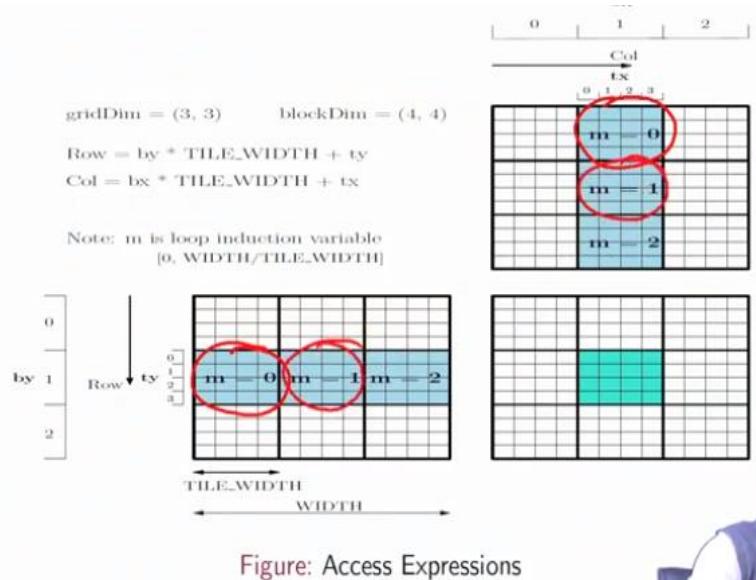
Figure: Access Expressions

So after all the threads for this tile have brought in this tile and this tile and then done the computation and what do I have? I have some data points computed for all the location but they are not the final results. They are all partially computed multiply adds for corresponding rows and columns. But what was the good thing here? The threads were actually using data brought in by other threads in parallel right.

So all the threads in this tile have bought in data, have stored them in shared memory and now while doing the multiply add operations, they have accessing data from the shared memory and they are not only accessing data that they have brought in they are also accessing data that other

threads have collaboratively bought in. So with this if I progress with the code, first of all after ensuring that all the threads have brought in the respective tiles in the load phase, then only I should go into the compute phase and do the partial sum computation. That is why after bringing the data there is `_syncthreads()` and then as long as I am not done with the partial sum computation, I should not go into the outer loop and bring in more tiles.

So that is why until and unless the partial sum computations are already done and whatever data as was brought in the previous load phase, all the threads need to synchronize only after this point. What is the guarantee I have? The guarantee that I have is having partial sums computed considering 2 of the input tiles. So once all those threads synchronize there, they will again revert back to the load phase. (**Refer Slide Time: 23:59**)



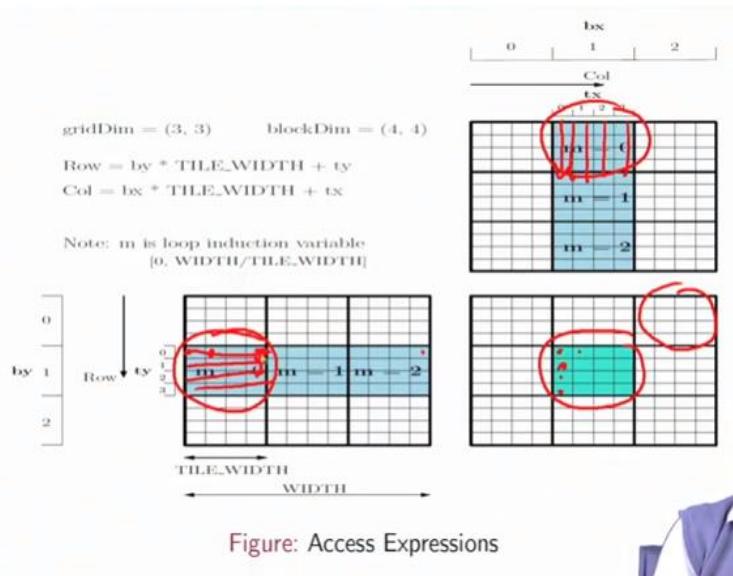
So assuming that this has already been taken care of i.e. these input tiles have already been taken care of, these threads would again bring in more tiles to the load phase and again start doing the computation. So that is what is happening here. So again they will go in here bring in data in the shared memory and again have a `syncthreads`. When all the data points are ready, they will again parallelly collaborate in doing the computation of the multiply add.

So for this simple example it is getting done right here. So since I have only 2×2 number of tiles, essentially the threads which are responsible for doing computation for these (1, 1) position tile, it will have 2 load compute phases. It will load tiles from here and here into the shared memory then collaboratively compute the sum again. Again in the next phase. It will load

tiles collaboratively and then compute the sum. In general, it will be having much more number of load and compute phases like that.

So overall you can see that only when the entire sequence of loads and computes are done the partial sums will finally get an update and they will represent the final results. So once each thread has completed its sequence of loads and computes the private variable corresponding to its own i,j th location. And then it would actually load this final value in the corresponding location in the target output matrix in the device memory. So this d_P is the target output matrix in the device memory. It is getting updated only at the last.

(Refer Slide Time: 26:02)

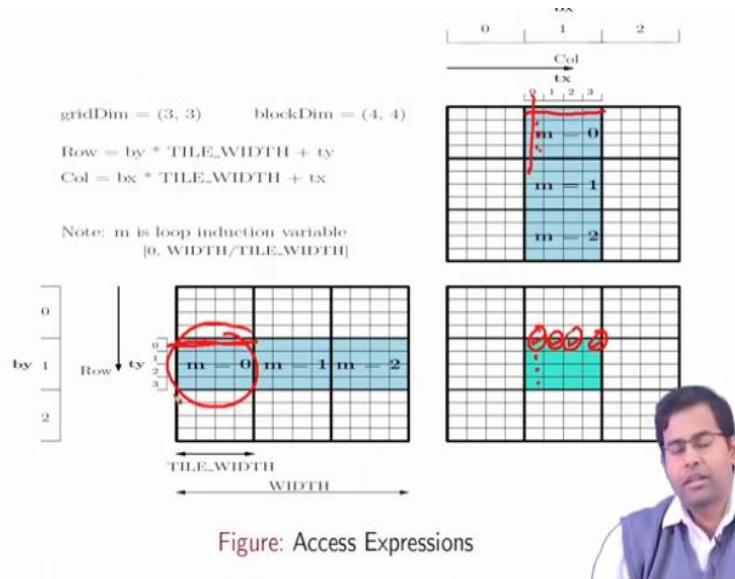


So to make things much more clear what is happening is for doing the computation of this tile, threads for these locations are making progress in parallel. First thing they did is that all the threads together brought in stuff from these 2 tiles and then they made progress. For example this thread made progress like this and this. In parallel some other thread was making progress like this and again in the same column etc..

But after this phase is done, all the threads have covered all these positions. See everything is still going on in parallel like our original optimized CUDA kernel for parallel matrix multiplication. Like while this thread has been computing for this tile, the other threads are of course computing for their respective tiles right. But the good thing that is happening is that each thread is not performing all the loads it is required to perform from the global memory. That means each thread is not bringing the same row of data in multiple times.

This thread has brought in this data, whereas next thread has brought in next position of the data but for doing the computation, this thread is using all those data, loading them when it is requiring them from the shared memory not from the global memory. And that means many number of global memory loads are getting changed to shared memory loads.

(Refer Slide Time: 27:46)



As an example, I would just again revisit this. So consider that the threads are working here. So essentially if I consider all the data that is brought in by these 4 threads, it would be this whereas the other threads are bringing in data for the other locations. But when these threads are into the compute phase, Eg: the first thread is accessing everything from here and here where all these other entries has been brought in by thread of this locations.

But when this thread is doing the compute in loading data from these other locations, it is doing all these loads from shared memory. It is not doing these loads from the global memory. That is the most important point. Essentially once all the global loads have been performed while bringing the things for this entire tile collaboratively by all threads in parallel, after that the next sequence of multiply and add operations occur inside the inner loop.

(Refer Slide Time: 29:09)

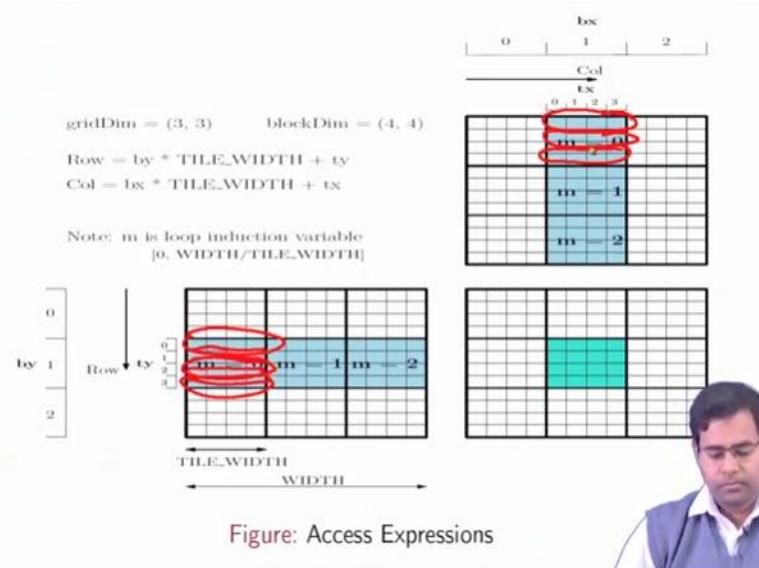
```

int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;
float Pvalue = 0;
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
    Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
        Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
}
d_P[Row*Width + Col] = Pvalue;
}

```

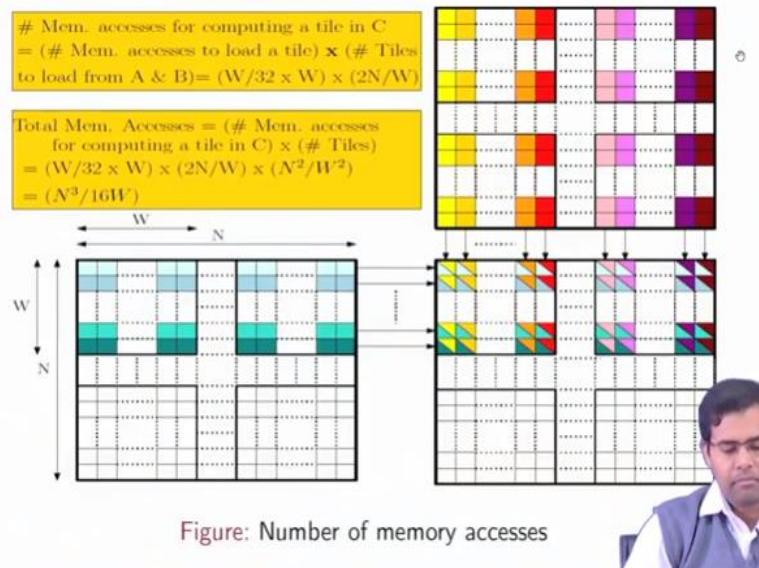
All the operations are now shared memory operations. So I can now say that now I have a clear difference here. All these loads are my global memory loads. But after that all the threads are accessing data from the shared memory. So all these accesses are going to be shared memory operations and those global memory loads. What is the good thing about them? So now I come to the next part which is a fascinating part.

(Refer Slide Time: 29:56)



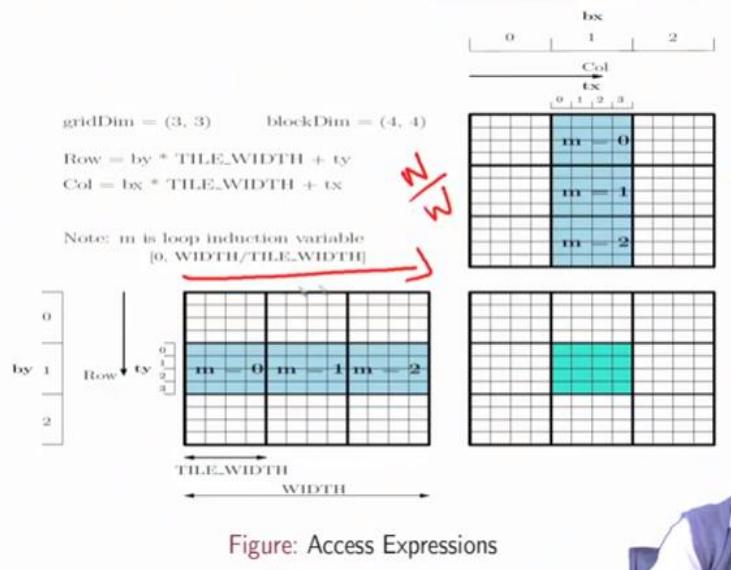
So observed the load pattern. When all the threads are loading data here (loading data from these points), all these loads will be coalesced again. All the other loads will be coalesced and like that similarly here. Since I have broken down the computation to these phases of loads followed by compute again loads followed by compute, so these loads are now nicely coalesced.

(Refer Slide Time: 30:39)



So over all, if I am trying to do a computation of how many memory accesses are done for computing a tile, it is equal to the number of memory accesses is required to load a tile times the number of tiles to load from the 2 input matrices. Let us call them A and B. So what is the number of tiles to load? So you think this is for computing a single tile.

(Refer Slide Time: 31:24)



So the number of tiles to load would be so this N and divide it by the width so that gives your N / W number of tiles here and similarly here. The total you have got is $2N / W$ number of tiles to load for computing a single tile. So you have got in total these many tiles to load and what are

the number of memory accesses required to load a tile? As we have discussed that when I am looking into the tile, all these accesses are going to be coalesced row wise.

So the number of accesses would be $W / 32$ times W . The times W would come for each of the columns but for each of the rows I have got coalescing $W / 32$. So this is the total number of accesses for computing a single tile. The total number of memory accesses in general would be the total number of memory accesses for computing a single tile times the number of tiles. So that would give me $W / 32 * W * 2N / W * \text{the number of tiles}$ which is of course $(N / W)^2$ or N^2 / W^2 . So that gives me $N^3 / 16 W$.

So that would mean if I can choose a W which is significantly big, I can have an order of magnitude deduction here by N . So W is comparable to N , it is a big fraction of N , I can have lots of acceleration here right. So that is the primary take away of the message here. With significant amount of tiling depending on of course the amount of shared memory you have available, I can reduce the total number of global memory accesses for computing this matrix multiplication for a significant factor by getting the threads to collaborate work from the shared memory and also tiling is helping me to increase the amount of coalescing that is possible here.

So in that way it leads to significant reduction in the total computation time. And as we know that matrix multiplication being a fundamental operation being part of many significant optimization techniques, ML workloads and many other places, if you can accelerate matrix multiplication that really helps in so many application domains. So this is one of the most fundamental computations where GPUs really help and we will take several such examples of other computations in the next lecture. Thank you for now.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 05
Lecture No # 23
Memory Access Coalescing (Contd.)

(Refer Slide Time: 00:28)

Transpose Operation

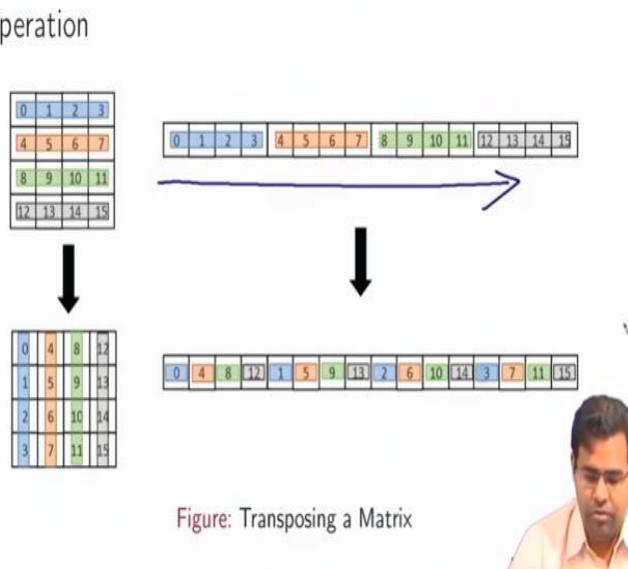


Figure: Transposing a Matrix



Hi. Welcome to the class of GPU architectures and programming. In the last lecture, we have been discussing about how an operation like matrix multiplication can be accelerated with proper optimizations involving the shared memory. And in this lecture we will continue with some more examples may be bit more involved and we will also try to show how you are able to perform some monitoring based on the optimizations and figure out how these optimizations are working for you using some profiling primitives that Nvidia provides for you.

So as an example, in the last case we considered the operation of matrix multiplication. In this case we will consider this example of matrix transpose. So if you remember from your high school algebra, a matrix transpose was a simple operation such that suppose you consider a matrix A is given to you and you want to create a A^T . So the way you would be defining the entries of transpose would be that whatever was an entry of the i th row j th column $A[i][j]$ in the original matrix in the transpose matrix they should be the the same entry in the j th row and i th column.

So the way it is going to work is that in the transposed matrix, if you consider any entity in the j^{th} column it should be same as the entry in the i^{th} column of the original matrix. So this is how it would hold at the element level. So essentially if we take an example that has been provided here so as you can see that we are considering an original matrix which is like the one that is given on the top left and when I do a transpose essentially all the i,j elements in the original matrix become the j,i elements in the transpose matrix.

So essentially, I can just say that the diagonal is remaining the same and the rest of the elements are just flipping over to the other side. So this comes here and this goes here. And similarly for all the other entities here, but how is it going to really happen in the underlying architecture? That is the important question here right. So if I look at the arrangement of this matrix with respect to the memory, then for all practical purposes this is a derivative of C so we have a row major implementation right.

So your matrix is finally resident in the GPU memory in simple consecutive locations for the different elements in the row followed by the next row and so on and henceforth. So when you are really doing a transpose operation, things as you see that which looks very nice in the original 2D representation you may lose that simple property here. Of course in terms of the matrix it is fine, but here things will be happening in this single 1 dimensional array which you will be accessing using the row and column indices of the original matrix. That actually you have to again compute using the block and thread indexes of the original matrix.

(Refer Slide Time: 04:23)

Matrix Transpose CPU only

```
void transposeHost(float *out, float *in, const int nx, const int ny)
{
    for (int iy = 0; iy < ny; ++iy)
    {
        for (int ix = 0; ix < nx; ++ix)
        {
            out[ix*ny+iy] = in[iy*nx+ix];
        }
    }
}
```

Professional CUDA C Programming by Cheng et al.



So if we consider a simple vanilla matrix transpose function in C i.e. a CPU only function. So how really you are going to go about it? You will just write a cascade of loops - 1 loop is iterating over the rows using iy. So the outer loop is iterating over the rows. So this iy index is iterating over the rows and the inner loop is iterating over the columns per row and inside what you are doing is that you have been given an a pointer pointing to the input matrix and you are going to write the data into the output matrix using this out pointer.

So essentially you are just moving i.e. you are just copying elements from the yx^{th} location to the xy^{th} location. So I can just say this is in terms of the representation. Of course whatever is in the yx^{th} location the index in y axis and index in x axis gets flipped to index in x , y location.

So this is some of the examples taken from this book Professional CUDA C programming here.

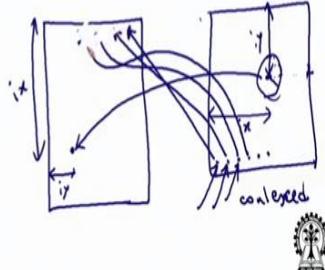
(Refer Slide Time: 06:19)

Matrix Transpose GPU Kernel- Naive Row

1 2 3 . .

```
__global__ void transposeNaiveRow(float *out, float *in, const int nx, int ny)
{
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    if (ix < nx && iy < ny) {
        out[iy * ny + ix] = in[iy * nx + ix];
    }
}
```

Loads by rows and stores by columns



Now if we consider a GPU version of a code what would happen? Of course since in the CPU, I would have cascaded loops and so it is a sequential loop and it is iterating over the 2D arrangement of data points and using the loop iteration variables and doing the transformation from the in matrix to the out matrix by accessing these entries. But when it will be done by the GPU? Again the work will be per thread basis. So you will launch a kernel as always done. You will launch a kernel and you will be able to find the partial activity.

So every thread would need to compute its ix and iy values i.e. on which location of the matrix it is going to work and then it is going to copy the corresponding location from the in matrix to the out matrix. So then every thread which discovers which location it is supposed to work on and then it is going to copy here in the corresponding location. So whatever was the row (it was iy), it changes to the column offset here and whatever was the column offset is going to change to the row number here right.

But again, we are calling it as a naïve implementation because I am not making use of any shared memory. Another thing of course we think that it can be optimized further using such implementations. We will get into that. But here observe what is happening. So essentially I call it as naïve row implementation because we are loading rows and storing by columns. Why do I say that values are loading by rows?

Because think how the warps are getting formed. So you have thread ids for consecutive locations in the rows. You have a warp which is containing thread id's 1, 2, 3 like that and they are accessing the input matrix. They are accessing consecutive locations and forming a warp. And they are loading the data from these consecutive locations which would mean the loads are coalesced.

So whenever this thread inside a single warp is loading this data point, the next thread is loading the immediate next data point and like that. So this is going to get coalesced. But what happens when the threads in the same warp are going to write? So here for this example that I am trying to draw, essentially we are picking up items at the lower level with a large row index.

So that large row index would change to a large column index here. So maybe somewhere from here, I would be starting to drop the elements and we are having offsets like this. So this column offset values would be changing to row offset values. So if I say I have a large row index and a small column offset, that would transfer to a small row index and a large column offset.

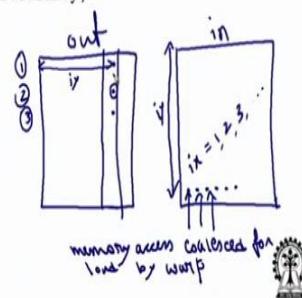
So, essentially this thread would be storing the data somewhere here. The next thread will be storing somewhere here and the next thread would be storing somewhere here like that. But wait a minute. Let me draw this with a bit more accuracy. Sorry.

(Refer Slide Time: 12:14)

Matrix Transpose GPU Kernel- Naive Row

```
__global__ void transposeNaiveRow(float *out, float *in, const int nx, int ny)
{
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    if (ix < nx && iy < ny) {
        out[iy * ny + ix] = in[iy * nx + ix];
    }
}
```

Loads by rows and stores by columns



So consider a fresh picture here. We have the original in and outs. This is my in and this is my out. So let us assume that I am doing again the same example. I am doing loads. The warp is definitely accessing consecutive locations in the row. Why? Because, since the warps are going to access consecutive locations and the consecutive locations would mean consecutive locations in the row in general.

So as long as the size dimension of the matrix is divisible by the warp size observe that you are loading data from this point in a single warp. So these are your loads and they are all coalesced. Now what happens when the threads are going to write? So this index is going to be written like this. So you have a large offset here with respect to y and then for the same row offset, let us give this some name. Let us say this phenomenon is happening for some value of iy and let us say ix values are 1, 2, 3... right.

So now when you think where these coalesced load values, the values which have been loaded by the warp in a coalesced memory access, where are they going to be get stored in the out matrix? That would be interesting. Why? Because now this ix value flips to become this iy value i.e. the row index flips and becomes the column index. So let us consider the first thread here. So it was loading data from a location with (iy, 1). That would mean it is now going to download that data at in the out matrix at a location (1, iy).

So that is it say this is the first row. So the data would be coming here. What about the next one? The data was (iy, 2). So it should get in (2, iy) next (3, iy) like that. So essentially whatever you loaded consecutively are going to get stored in the iyth column from the iyth row. So essentially you are doing a load by row and you are storing the data by column. Now of course if you think from the access patterns, these writes are non-coalesced right.

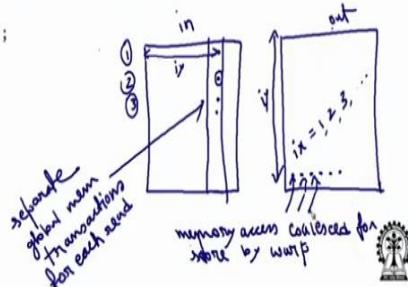
Technically considering a big matrix, all these writes will require separate global memory transactions right. So that is why you will be coalescing. The good thing about the goodness of coalescing is that it will help you with respect to the loads. But since you are storing by columns you will be having a penalty here.

(Refer Slide Time: 16:45)

Matrix Transpose GPU Kernel- Naive Col

```
__global__ void transposeNaiveRow(float *out, float *in, const int nx,int ny)
{
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    if (ix < nx && iy < ny) {
        out[iy*nx + ix] = in[ix*ny + iy];
    }
}
```

Loads by columns and stores by rows



Now consider the other scenario where as you can understand that this was a simple GPU kernel where all we did with respect to the original CPU kernel was we removed those 2 loops and just defined a per thread activity. Per thread activity was that you just load values and put it in the transposed location in the output matrix. But I have 2 options now. I have been discussing here the first option is you just load by row and store by column and since this is one option , of course I have the option of why do not I load by columns and store by rows?

So all that will happen is that your access expression is just going to change. So this is the example where you are going to load the data by column and store the data by row. So I can just reuse the same picture here. Since I am loading by column, I can say that this is my in matrix, this is my out matrix and I am loading these data points from the column and what would coalesced be? - the writes.

So I will have memory access coalesced for write by the same warp. So this is now write. This is now read. So I have now got separate global memory transactions for each read operation or store operation or load operation right. So now I will have the advantage of coalescing while writing since I am storing by rows and I lose the advantage of coalescing in terms of reads. So as we can see these are the 2 options I can have. But in both cases either my reads are coalesced or my writes are coalesced.

So that would give me some parallelization in terms of reducing the global memory loads or stores but I do not get both. So we have explained the idea of this check macro. Let us look at how to write the driver code including this macro.

(Refer Slide Time: 20:13)

Driver Code

```
int main(int argc, char **argv)
{
    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("%s starting transpose at ", argv[0]);
    printf("device %d: %s ", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // set up array size 8192*8192
    int nx = 1 << 13;
    int ny = 1 << 13;

    // select a kernel and block size
    int iKernel = 0;
    int blockx = 32;
    int blocky = 32;

    if (argc > 1) iKernel = atoi(argv[1]);
}
```

So essentially we are trying to write a simple driver code here which is the host program where we will be using this check macro to nicely capture the device properties of the underlying GPU device and print them up. Next we set up the input buffers and we are trying to write a single program through which the user can input what he wants, whether he wants to do a transpose by naïve row or by naïve column and that would be the user preference coming through which will be captured by the main program using the argv parameters.

So essentially I am assuming you are all familiar with handling command line arguments. So the iKernel variable is getting its value by using this atoi() function to actually process the argv string. This argv string is converted from the string type to integer type by atoi() and is stored in the iKernel parameter.

(Refer Slide Time: 21:16)

Driver Code

```
size_t nBytes = nx * ny * sizeof(float);
// execution configuration
dim3 block (blockx, blocky);
dim3 grid ((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y
            );
// allocate host memory
float *h_A = (float *)malloc(nBytes);
float *hostRef = (float *)malloc(nBytes);
float *gpuRef = *(float *)malloc(nBytes);
// initialize host array
initialData(h_A, nx * ny);
// allocate device memory
float *d_A, *d_C;
CHECK(cudaMalloc((float**)&d_A, nBytes));
CHECK(cudaMalloc((float**)&d_C, nBytes));
// copy data from host to device
CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
```



And then so that is actually storing the users preference of what kind of operation he wants and the next thing we do is we just set up the normal grid and block launch parameters for the kernel. I mean what is the block dimension, block x block y definitions and grid dimensions? So the way you want is the number of blocks to be parameterized. This is how we are writing. What should be number of blocks?

I mean essentially you divide nx by the block.x and ny by the block.y. So that gives you this 2D system which will give you the total number of blocks and in here you have the definition of the thread block i.e. how the blocks are arranged in terms of block.x and block.y. The next thing you do is you make suitable allocations from the host memory side. And then you make this call assuming that there is a function which is going to allocate the host array.

This host array h_A with the input with all the input values stored and the dimensions are of course nx x ny for this array. And then you allocate device memory in d_A and d_C. So you want the host array to be transferred to the device memory using this cudaMemcpy() command. So in device, you are defining 2 memory locations d_A and d_C. In d_A, you are copying the host array using the cudaMemcpy() command and everywhere you are using the CHECK macro to just to check if there is any system error. The check macro will nicely print the corresponding statement for debugging purpose.

(Refer Slide Time: 23:00)

Driver Code

```
// kernel pointer and descriptor
void (*kernel)(float *, float *, int, int);
char *kernelName;
// set up kernel
switch (iKernel)
{
    case 0:
        kernel = &transposeNaiveRow; kernelName = "NaiveRow"; break;
    case 1:
        kernel = &transposeNaiveCol; kernelName = "NaiveCol"; break;
}
// run kernel

kernel<<<grid, block>>>(d_C, d_A, nx, ny);
CHECK(cudaGetLastError());
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));
}
```



And here we use the function pointer to actually pass the user's input choice because I have already captured the user's input choice using this iKernel integer right. So argv[1] is containing the first option provided. It is containing what is the functionality that the user wants to execute right. So then here through a switch case we are trying to match this value of iKernel whether it is 0 or 1 and accordingly depending on what is the user's parameter, we are trying to set this function pointer kernel and provide it with the pointer for the device function transposeNaiveRow or transposeNaiveCol.

So essentially we are using this. We are setting this function pointer up here so through this switch structure, the user has already passed what is the preference in terms of the in terms of this argv parameter which have been passed to kernel right. So coming here, we are finally running the kernel. So as you see that now this function pointer kernel has been suitably initialized. It has been provided with the address of either transposeNaiveRow or transposeNaiveCol function whichever you want and then this kernel will execute. And of course these are all asynchronous statements here like the kernel will be launched and the CPU will be waiting for the last error message if there is any.

And then once then this cudaMemcpy() command will again be executed to get back the transposed kernel value from the GPU side.

(Refer Slide Time: 24:51)

Profile using NVPROF

- ▶ nvprof is a command-line profiler available for Linux, Windows, and OS X.
- ▶ nvprof is able to collect statistics pertaining to multiple events/metrics at the same time.
- ▶ nvprof is a standalonetool and does not require the programmer to use the CUDA events API.

Now in this example we will demonstrate usage of a profiler from NVIDIA like how to do a command line profiling using this nvprof tool and how to figure out what is the performance of your program. So with this initial idea that we can monitor performance of our programming using this kind of nvprof profiler, we will end this lecture. In the next lecture we will go into further details of using nvprof for doing CUDA kernel profiling. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 05
Lecture No # 24
Memory Access Coalescing (Contd.)

(Refer Slide Time 00:25)

Profile using NVPROF

- ▶ nvprof is a command-line profiler available for Linux, Windows, and OS X.
- ▶ nvprof is able to collect statistics pertaining to multiple events/metrics at the same time.
- ▶ nvprof is a stand-alone tool and does not require the programmer to use the CUDA events API.



Hi. Welcome to the lectures on GPU architectures and programming. In the previous lecture, we have started our initial discussion on profilers like nvprof. So this is a profiler which essentially you can execute in command line. It is available in Linux, Windows, and OS X Mac OS version. It can collect statistics pertaining to different program events and metrics at the same time.

It is a standard tool and does not require the programmer to use the CUDA events API. This is important because earlier we discussed about how to use the CUDA events API which actually samples the timings from the program and using the CUDA events API you can actually catch the timings. But that actually is a burden on the programmer because then you actually have to insert suitable timing codes or CUDA event calls inside your program.

So in case you do not want to use that path this is an alternate way to use it. That means you run your program under this profiler setup. We will just see some examples on how does it differ. I hope you all remember how to use the CUDA events API. You can consult our earlier lectures

where we showed how I can insert these kind of suitable data types from the CUDA event API and sample out time from the CPU or the GPUs performance counters.

(Refer Slide Time 01:43)

Execute Code: NaiveRow

```
nvprof -devices 0 -metrics gst_throughput, gld_throughput ./transpose 0
==108029== NVPROF is profiling process 108029, command: ./transpose 0
./transpose starting transpose at device 0: Tesla K40m with matrix nx 8192 ny
8192 with kernel 0
==108029== Some kernel(s) will be replayed on device 0 in order to collect all
events/metrics.
==108029== Replying kernel "transposeNaiveRow(float*, float*, int, int)" (
done)
==108029== Metric result:
Invocations Metric Name      Metric Description      Min      Max
Device "Tesla K40m (0)"
Kernel: transposeNaiveRow(float*, float*, int, int)
1          gst_throughput    Global Store Throughput 249.370GB/s 249.370GB/s
1          gld_throughput    Global Load Throughput 31.171GB/s 31.171GB/s
```



So here is an example where what we are doing is we are executing the code of transpose host. This is the original executable. I am passing it parameter 0. So that would execute the naive row kernel and if I pass it transfer parameter 1 it will execute the naive column kernel. If you can just see here case 0 was for naive row case and 1 was for naive column. So this is our original transpose executable. But when I executed I do not execute standalone.

I execute this with under the nvprof profiler. I pass it these different parameters which essentially say what are the performance statistics I am interested in. So essentially it will run this kernel possibly multiple times and try to compute some statistics out of it and give me the required architectural parameters which I am interested in. So here essentially when I specify global store and global load throughput that means I am asking the profiler to provide me with these statistics.

What is the number of loads and stores happening? And what is the bandwidth that is getting utilized i.e. how many loads and stores are happening per second in terms of how many bytes of data is getting loaded and stored per second? these are the minimum and maximum ranges and all that. So you can take a minute and just look into the statistic that has been provided here by the profiler. So it is separately giving me the global load and global store throughput values.

The number of bytes, the maximum that has been written or being read per second and also the minimum. So essentially this profiler is going to run the code multiple times and obtain a statistical observation out of it.

(Refer Slide Time 03:50)

Execute Code: NaiveCol

```
nvprof -devices 0 -metrics gst_throughput, gld_throughput ./transpose 1
==108037== NVPROF is profiling process 108037, command: ./transpose 1
./transpose starting transpose at device 0: Tesla K40m with matrix nx 8192 ny
8192 with kernel 1
==108037== Some kernel(s) will be replayed on device 0 in order to collect all
events/metrics.
==108037== Replying kernel "transposeNaiveCol(float*, float*, int, int)" (
done)
==108037== Metric result:
Invocations Metric Name Metric Description Min Max
Device "Tesla K40m (0)"
Kernel: transposeNaiveCol(float*, float*, int, int)
1 gst_throughput Global Store Throughput 17.421GB/s 17.421GB/s
1 gld_throughput Global Load Throughput 139.37GB/s 139.37GB/s
```



We will just look at the performance statistic once again and try to match with our original ideas. So when you are doing naive row essentially your number of loads will be much smaller than the number of stores you are doing, because you are loading by row which would be coalesced and you are storing by column. And as you can see that the idea quite really matches here. Because the load throughput is quite low whereas the stored throughput is quite high for the naive row implementation.

Now if you look into the other implementation of the naive column. So if you just remember our original program, if the user passes that command i.e. the command line option of 1 with the executable transpose, then that one will get into the argv[1] part and that would transfer to iKernel which will indirectly get into the switch statement and it will actually give the function pointer to the kernel. It will make it point to the kernel for doing the naive column execution right.

So again when I execute that code with this option 1 under the nvprof profiler, I again get the load and store throughput. So now since I am doing a naive column implementation, so I am

loading by columns. That means that there would be too many loads and the number of stores would be much smaller because the store is now coalesced and it gets reflected right. As you can see that the throughput for the store is much smaller whereas the throughput for load is much higher.

(Refer Slide Time 05:34)

Using Nvidia Visual Profiler

- ▶ The nvvp software provides a GUI based tool for analyzing CUDA applications and supports a guided analysis mode for optimizing kernels.
- ▶ nvprof provides a *--analysis-metrics* option to capture all GPU metrics for use by NVIDIA Visual Profiler software during its guided analysis mode.
- ▶ The *-o* flag can be used with nvprof to dump a logs file that can be imported into nvvp.

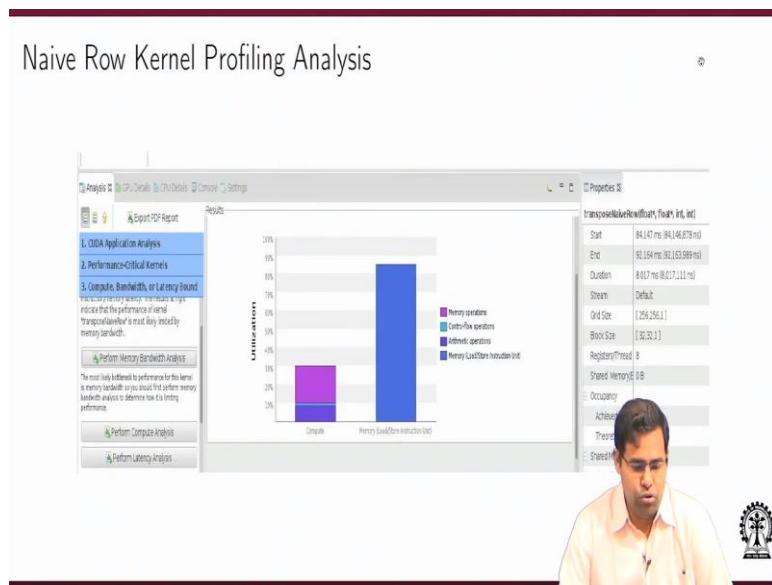


Now the statistics that gets collected by the nvprof can actually be rendered into a nice GUI based tool called nvvp - NVIDIA visual profiler. So this profiler or this software provides a GUI based tool for analyzing CUDA applications and it supports a guided analysis for optimizing across kernels. So I can actually get the statistics from the nvprof. The nvprof tool provides the *--analyse-metrics* option when I execute nvprof which can capture all the GPU metrics for use and this statistic will be stored in a format which is usable by the visual profiler.

So next when I run the visual profiler, it can render the statistics in a nice manner and it can help me to analyze the performance from the GUI right. Now of course I have to use this *--analyse-metrics* flag with nvprof, dump a log file and that has to be imported to nvpp. So you can just learn to use this thing.

(Refer Slide Time 06:44)

Naive Row Kernel Profiling Analysis



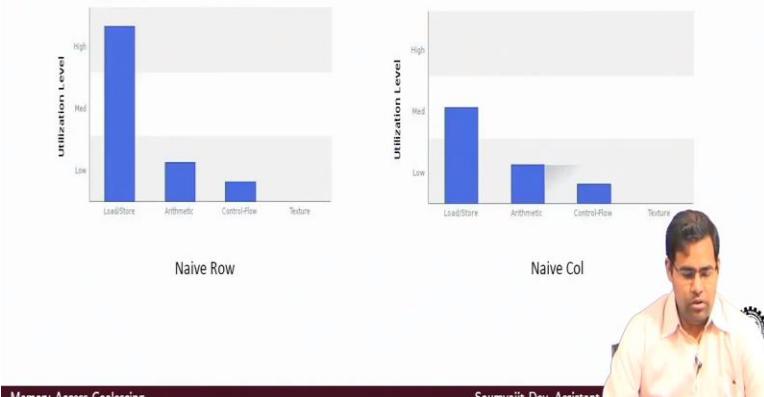
We will just show that when we run this original nvprof commands, the outputs can actually be rendered through nvvp through this kind of a nice UI. So these are the profiling statistics for the naive row kernel. As we can see, the exact timings for the start and end are provided. And we also have the execution duration of 8.017 milliseconds. And also the statistic we have asked here is for different utilizations.

So compute utilization, the memory load store utilization - they are actually plotted nicely. Since this is a less compute intensive but more communication intensive i.e. load store intensive kernel, that can quickly be gauged from this statistic chart that has been provided. So we have lots of memory operations but much less number of compute operations here. And significant amount of very large number of memory loads and stores here. Similarly I can also do that for the load by column case.

So this is a naive column implementation as you can see naive row I have some timing which is 8 milli second. For naive column I have even more timing requirement which is around 14 milliseconds. Now there is an interesting question here right because of course in one case the naive row case the loads are coalesced, but stores are not coalesced. Whereas in the naive column case loads are not coalesced but stored are coalesced.

(Refer Slide Time 08:24)

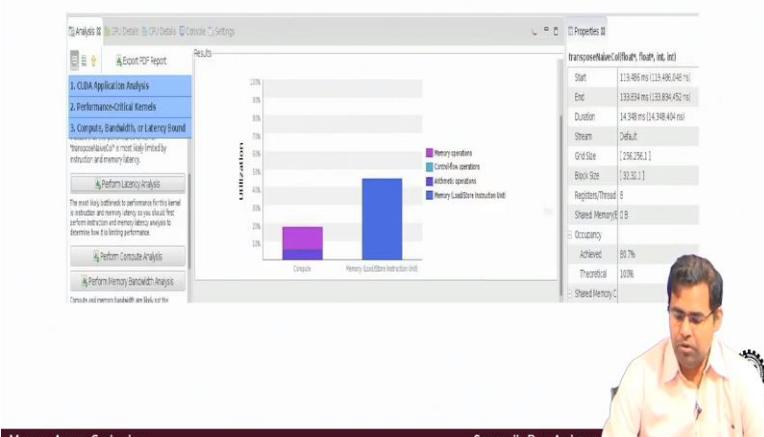
Compute Analysis



So if I do the compute analysis, I also can get the charts here in terms of utilization. So this is the utilization chart here. If I plot them side by side, we can see the loads stores. Apparently if I just provide them the number of the utilizations for the load store is further smaller in the naive column case with respective to the naive row case.

(Refer Slide Time 08:54)

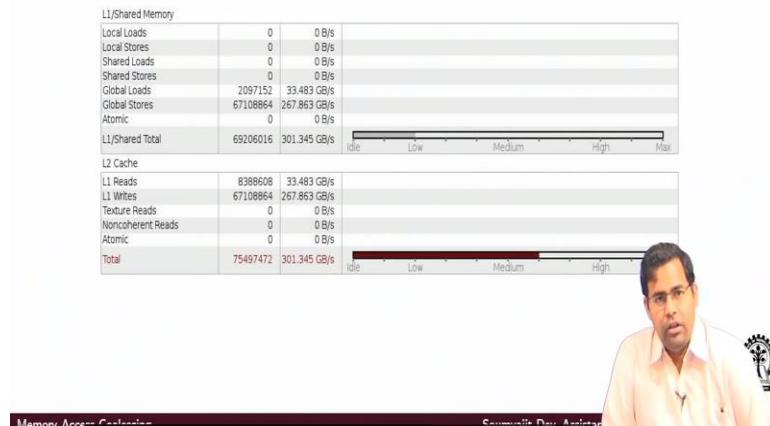
Naive Col Kernel Profiling Analysis



And that actually matches with the execution time statistics because from our observation for the naive row case the execution time was smaller - it was 8 milliseconds. Whereas for the naive column case the execution time is much larger - it is 14 milliseconds. So it is much larger and that gets reflected here in terms of the utilization of loads stores. In case of naive column the utilization is less.

(Refer Slide Time 09:19)

Memory Bandwidth Analysis: Naive Row



But again if we come to doing the comparison with respect to the memory bandwidth that has been consumed here for the naive row as well as the naive column implementations. Let us go one by one. So for the naive row implementation this is what we have. So as we can see that we have loaded this by row, the number of global loads is less.

The number of global stores are much more and so this is basically the number of global loads that we are having. And then this also provides you with the statistics of the L2 cache i.e. how many reads are performed by the L1 from L2 cache and similarly how many writes are performed by the L1 to the L2 cache right. Now something interesting as you can see that the number of L1 reads is much more with respect to the global loads. It is of course 4 times.

Now what is the reason for that? The fundamental reason is that in our GPUs the L1 is 128 byte wide. The cache lines of L1 are 128 byte wide, essentially that is 32×4 bytes. So that is equal to the global memory transaction width. So the L1 which is also is configurable as L1 or shared memory, the width of the cache line is same as the global memory transactions.

But in between I have the L2 cache - the unified L2 cache across the SMs for which the lines are 32 byte width right. So essential when the L1 has to read, the number of reads is 4 per unit and in that way whatever is the number of global loads, since the number of global loads have to go

through L2 to L1. L1 is having a width which is matching the load width . The load width which matches with the warps read width.

But in between you have a L2 cache which is 32 bytes. It is having cache lines which are 32 byte width. So you have a factor of 4 increase in the number of L1 reads required from the L2 right. So overall if you can look at here , since this is naive row, the number of global loads is small. The stores are larger and similarly we can also say for the L1 reads and writes. The number of reads is small. The number of writes are much more which is completely coherent with the number of stores here.

But then if we go to the naive column implementation, we similarly see that this is the other way round. The number of global loads is much more because they are not coalesced. They are loaded from columns. The number of stores is much smaller and similarly so when the L1 is reading. They are same as the number of global loads but when the L1 writes to the L2 cache , of course it is much smaller in number than the number of reads.

But again we have this factor of 4 increase, when I look at the number of times L1 writes to L2 and the number of stores, because again we have this multiplication factor of 4 here because of the width issue that I just discussed. So it would be advisable at this point that you keep these things in mind that whenever doing reads and write what is the width of the cache lines and what is the width of the global memory transactions. And these things may or may not change with architecture families.

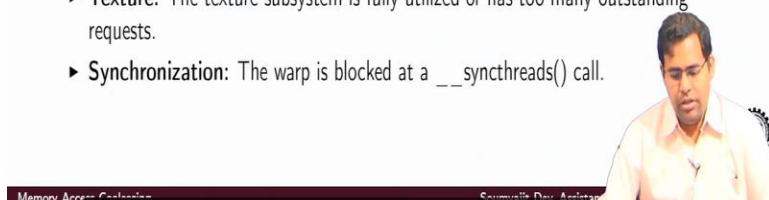
This is something we need to be aware of if you have to correctly interpret profiling data from any execution run.

(Refer Slide Time 13:39)

Latency Analysis in NVVP

Instruction stalls prevent warps from executing on any given cycle and are of the following types.

- ▶ **Pipeline busy:** The compute resources required by the instruction is not available.
- ▶ **Constant:** A constant load is blocked due to a miss in the constants cache.
- ▶ **Memory Throttle:** Large number of pending memory operations prevent further forward progress.
- ▶ **Texture:** The texture subsystem is fully utilized or has too many outstanding requests.
- ▶ **Synchronization:** The warp is blocked at a `__syncthreads()` call.



Now this still does not really explain to us that why the number of number of seconds or milliseconds and the total amount of time taken by the naive row implementation was much smaller with respect to the naive column implementation. Because in both cases, I have some penalty. In one case the penalties with respect to the loads and in the other case the penalties with respect to the stores.

So if you do a latency analysis in using the NVIDIA visual profiler you see that it provides the reasons why instruction stall prevents the warps from executing. So we always know that we have lot of warps to execute but warps are stalled due to certain reasons right and the reasons can be very for example the pipeline maybe busy. If the pipeline is busy then the compute resources that are required by the instruction for the warp is not available right. So the warp is stalled.

Then the issue can be some constant value is going to be required and that constant load is blocked due to a miss in the constant cache. So as we know that the constants can be stored in the separate constant cache rather than using the L1 or shared value L1 or shared memory. The other issue can be of memory throttling like there can be large number of pending memory operations which prevent further impede forward progress. Lots of many memory operations are pending, so a warp has to stall. Then the texture subsystem. If it is already fully utilized and this is more for the graphics part if it is fully utilized. And there are too many outstanding requests from the texture systems. Then also warps can get stalled. And of course the other thing is a warp

can be blocked at a `__syncthreads()` call, because the other threads which are part of the block and hence part of the other warps have not progressed up to the synchronization barrier.

(Refer Slide Time: 15:37)

Latency Analysis in NVVP

Instruction stalls prevent warps from executing on any given cycle and are of the following types.

- ▶ **Instruction Fetch:** The next assembly instruction has not yet been fetched.
- ▶ **Execution Dependency:** An input required by the instruction is not yet available.
- ▶ **Memory Dependency:** A load/store cannot be made because the required resources are not available, or are fully utilized, or too many requests of a given type are outstanding.
- ▶ **Not Selected:** Warp was ready to issue, but some other warp was issued instead.

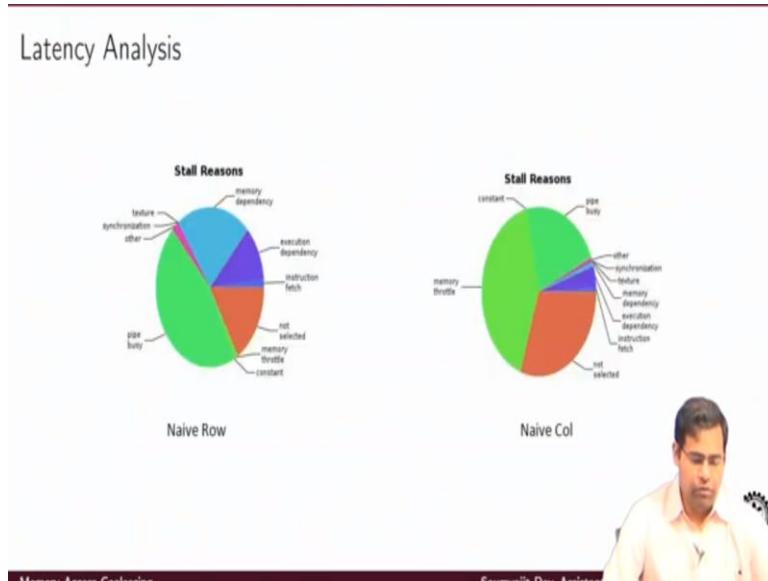


So if there is a stall in any of given cycles then it prevents the warps from executing and the different kind of instruction stalls that can happen can be classified into the following types - like this instruction fetch. In this case if there is a stall then if there is then the next assembly instruction will not be fetched right. Now of course there can be execution dependencies like an input is required by the instruction and that is not available. Of course there can be memory dependency that I require the data points from the memory. I require to store data point into the memory.

It cannot be done because the resource says that a memory system is quite busy and then the warp is ready but the warp scheduler is not ready to issue it. The warp scheduler may be busy with some other threads which it has already issued. So there can be these several reasons due to which warps may stall from execution.

(Refer Slide Time 16:56)

Latency Analysis



But if we do a analysis of these different types, as you can see we have all these listed types of different reasons for which the warp can stall from execution and if we do a latency analysis of both the kernels that we executed (the naive row and naive column kernels) and see what was the primary reason for the warps stalling. Then we can see the naive column kernel has significant amount of stall due to the warps not getting selected and the memory throttling issue.

Whereas in the naive row whenever there was some stall it was primarily to do with the pipeline being busy executing some operation right. Now observe something. If the pipeline is busy in executing some operation, it really is not a bad thing because still you have progress in terms of operations. But when you have the memory throttling issue, if you want you can just have a look into the memory throttling issue. Essentially you have large number of memory operations that are pending or memory dependency issues that say that load stores cannot be made because the required resources are not available. For example, I am reading from multiple banks or 2 warps are going to read from the same bank, but I cannot do that. So due to all these separate issues, I can have stalls happening. But suppose take this example in case of naive column memory throttling is a very big issue.

That means there are lot of pending memory operations and the warp not getting selected is also a significant issue. Now why would that happen? If you remember the naive row implementation you are able to load very fast. Since you are able to load very fast, the warp would make progress. But they will get stalled in the right part. But here your warps cannot even make

progress because you have got the issue that you are loads are getting stalled because the loads are not coalesced in case of naive column.

Since your loads are not coalesced the warps really cannot start executing or at least copying the data. So that would actually create more significant proportion of the scenario that memory throttling is happening and the warp is not selected. Now of course every kind of stall has a corresponding penalty but things related to the memory will always have larger penalty and since you are having lot more load operations which are stalled, that also contributes to the naive column implementation being slower in this case.

(Refer Slide Time 19:27)

Using Shared Memory: Simple Copy

```
__global__ void copySharedMem(float *odata, float *idata, const int nx, const
                           int ny)
{
    __shared__ float tile[TILE_DIM * TILE_DIM];
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = blockDim.x * TILE_DIM;
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] = idata[(y+j)*width + x];
    __syncthreads();
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x];
}
```



Now of course there can be further optimizations that can be made using the shared memory or using a shared memory based implementation of the code. Now this is something which we will like to take up in the next lecture. So we will like to see how the shared memory can actually help in doing transpose operations, because still now all we have done is looked into the transpose operation from the point of view of global memory loads and stores. So with this we will end this lecture here. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 05
Lecture No # 25
Memory Access Coalescing (Contd.)

Hi Welcome back to the lectures on GPU architectures and programming. So in the last lecture if I recall correctly, we have been discussing certain possible ways in which we can actually implement the matrix transpose operation in a GPU. So we actually provided 2 possible options. One was you load by row store by column and another was load by columns store by row - two possible different ways to write that matrix transpose operation.

And we saw what are the possible ways in which the computation gets affected and all that. We will continue in the same frame and we will today actually explore further into how transpose can be accelerated using several optimizations which are possible. So the first optimization would be, if you remember correctly in the previous approaches we did not make use of the shared memory.

So the first optimization would be that you do not directly load from the global memory and then store back into the global memory. But rather you load into the shared memory and then you write back into the global memory.

(Refer Slide Time 01:40)

Transpose using Shared Memory

```
#define TILE_DIM 32
#define BLOCK_ROWS 32
__global__ void transposeCoalesced(float *odata, float *idata, const int nx,
                                    const int ny)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = blockDim.x * TILE_DIM;
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];
    __syncthreads();
}
```

Source: <https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/>

So let us get on with the first example on computing transpose operation on matrix while exploiting shared memory optimization. In the program example that we have provided in our case, we have written the TILE_DIM and the number of BLOCK_ROWS to be equal. So to be more specific here, let me just give an example what we mean by this 2 parameters. So essentially just like our earlier example where we exploited share memory, if you remember it was in matrix multiplication.

So again we will be using a similar concept of tiles and for that we will define the tile dimensions and this space we are considering 32 x 32 size tiles. So we are considering 32 x 32 tiles. And now when we start talking about the number of blocks, what is the size of the blocks? In this specific example that we will be giving we are again considering the number of rows in a block to be also 32. But if you look into some other similar examples for this same problem like for example in the source that as been provided, you will see that essentially they will consider the thread blocks to be a size 32 x 8. Now there is a reason for that. We will come back to that. And there is also a reason why here we are considering both as 32. We will come to that. Here I just want to make clear that there is a difference in the way you will see the example done in the references with respect to the way we have done in the example. So coming back to our idea of doing a matrix transpose while exploiting shared memory.

So the whole reason for using the share memory would be what? Like if you have seen that other naive column and naive row approaches, there was either a problem with non-coalesced loads in

terms of global memory coalesced or there was a problem with non-coalesced stores. We want to elevate the problem and use the shared memory to avoid this problem of global memory coalesces either in terms of loads or in terms of stores.

So what we will do is instead of taking any of those approaches that you load a full row of data from global memory and storing back to global memory into a full column or the reverse. Rather we store at a time a tile of data. So if we just try to give a pictorial representation here. So we will divide the entire memory space in terms of tiles and then we are saying that the first thing I am going to do is I will load a tile of data from the global memory location of the matrix to the shared memory.

So for doing that first if we just keep on following of our example, we define the tile dimension 32 cross 32 and we are considering blocks to be also of the same size as tile dimensions. That is the number of rows in the block is 32. So that would mean in one load operation I have this many numbers of threads. These 32 x 32 number of threads which are operating and loading a full tile into a sharing memory right.

So I have got this many number threads here. Now each thread will work with which point in the global memory? - The corresponding x and y coordinates. It can just compute from this expression right. So consider for example here so this is a block with id (0 0), block with id (0 1), block with id (1 0), block with id (1, 1).

So, I have the blockIdx.x representing this way and blockIdx.y representing this way and again inside the block I have the threadIdx.x representing this way and threadIdx.y representing this way. That is the 2D representation here and so the overall width is of course given by the grid dimension i.e. if I take a simple example it is going to be : grid dimension the x direction is number of blocks in the x direction times the tile dimension which is obviously this right here.

So essentially that would be $32 \times 2 \times 32 \times 2$. In each case, the width essentially is 64 for this simple example I have taken here. So now what will happen is here I am trying to run a loop using which 1 thread would be loading the values corresponding to its location. For our example first of all let us forget this loop. We will come back to this and we will also see why this loop is hidden for our purpose.

So if I forget this loop, that would mean I can also forget this j here right. Now look at the code. So essentially all that is happening is this $idata$ is your input matrix which has been passed. So essentially you have computed for each thread x y coordinates. So let us pick up some place here. So this is my y and this is my x . So I have computed these x y coordinates and accordingly this expression is simply giving me $y * \text{width} + x$ of course the width is the grid dimension i.e. 2 blocks, so that number of blocks in the x direction times the tile dimension is my width. So multiply y by width and plus x . That gives you exactly this location. So essentially you load this thread. You load this position from the $idata$, and it would write into the corresponding tile right. So let us say this is my tile and in here where is it going to write first of all. So of course, now if I look into the corresponding thread id for this x it is this corresponding location.

So the way I have computed the x and y values is essentially the offset of the block in the x direction multiplied by tile dimension plus threadIdx.x . So this essentially would be my threadIdx.y and this essentially would be my threadIdx.x . That is the offset inside block right. That is the exact location which is the corresponding matching location in the tile in the shared memory where this thread would be loading the value.

So essentially I have this tile which has been defined and what is the thread doing ? It is doing a load of this tile. Now let us come back to what essentially is going on here in terms of the loads by the threads. So every thread is going to execute the same code here. It will compute exactly in which location in the tile is going to work with right. So overall you have this shared memory that has been defined and then when you have this code here, this thread is essentially computing where exactly it is going to load from the input data.

And then the next part which is interesting here is that the thread now has to figure out that where in the output matrix is going to write the data that has loaded in to the tile. I hope this part is clear to everybody. Now observe the interest thing here. You have defined the shared memory. As we know the shared memory is defined in a way that it is transparent to the entire block of threads right. In this simplistic setting that we have taken, the block size is technically here equal to the thread dimension. That is what we are considering.

So essentially you have a block of threads which are each copying something - one element into the tile and in that way each thread inside the thread block is actually loading some element into the tile. And up to this point we do not have the problem. Essentially all the threads are doing sequentially row accesses in the global memory. And there actually writing into the shared memory again using a sequential row access as you can see from this expression.

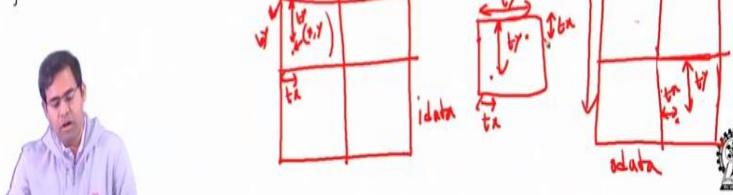
(Refer Slide Time: 11:52)

Transpose using Shared Memory

```

x = blockIdx.y * TILE_DIM + threadIdx.x; // transpose block offset
y = blockIdx.x * TILE_DIM + threadIdx.y;
for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}

```



So up to this is fine. Now let us look at the next step that we do. Now the thread has to figure out where it is going to write in the output matrix. So again let us reproduce the previous figure. Let us say a thread block has actually copied the entire tile here and then it has now ideally copied it into this shared memory which is offered a consistent view to the entire thread block. So this is my input and this is my output matrix.

Now the way we are going to do this first thing is that the thread has to figure out whether it has taken the data from one specific block in the input matrix. The output is going to be retained into each specific block. For doing this, so essentially we understand that everything that is inside this block, (Let us in general write this as a block (i, j)), that would be flipped right. So essentially now I am ideally writing it to j , i th block right.

So this has to be flipped. That means technically whatever was my blockIdx.y should provide me the new block id in the x direction. So to do this flip of the block, I recompute my x and y using this formula. Let us look at it very carefully. So I am computing the new x and y values where

this thread is going to write. The way it is computed is so x is now the original block ids yth position. So if we just try and understand this, this is the i, jth position let us say.

So here I have a blockIdx.x and blockIdx.y or if I just keep it as same with original symbols that we are using in the program, let us just call it x, y. So this is the original x, y now the way which we have divided this is that we have a block id x and a block id y. And then again inside it I have an offset thread id x and thread id y right. So the first thing I have to do is I have to flip the block right.

So that would mean whatever was my original block id x should now start to become my new block id y right. So that is why when I am calculating a new y value, I do a block id x times the tile dimension right. So essentially if this was my original block id x this will now be considered to calculate the offset of the block in the y direction. So that is why I do a block id x times tile dimension. And similarly whatever was my original block id y, now we use to calculate the offset of the block in the x direction.

That is why the block id y is being multiplied by the tile dimension to calculate x right. But the thing is that when I am calculating the xyth position, I do not change the x y coordinates original position relative to the block. Let us understand what it means. So fundamentally speaking if this was my x y position, then here the actual transpose position should be what? In order to find that out, I should have actually considered the original x and original y and just use the representation in the opposite right.

So the x would become the y and the y would become the x right. So if I just consider the previous values of x and y which have been calculated following this expressions here, so that x is used by block id x times the dimension plus the thread id x. So this entirely should be combined to new y. So that would give me exactly the position where I am going to place the transpose. But we are not doing that. We will understand why we will not be doing that. We just see that they can be done in a opposite and hierarchical way.

What we are doing is the first thing we will achieve is we will first flip the position of the block. If this is the original block position, we will find out what is the alternate block position. So when we figured out the alternate block position, we will compute the x and y coordinates of the

corresponding x y values in the original by keeping the relative position here of x and y same. To be more specific consider this was your original tx and ty.

Now you are in the flipped block here. But in the flipped block, when you are calculating the x and y position you do not change your tx and ty. You consider the same tx and ty position. So the relative position of this point with respect to this block is same as the relative position of the original point which is this one block. Just the block position has been transposed.

There is a reason why we are doing that. Now again look at the code that is following. Again for our purpose we are just ignoring the loop. If I ignore the loop I can ignore the j in the expression that has been given. So now what do you have in the loop? First thing is even before looking into the expression, consider the scenario I would have got had I just switched the original x and the original y. Then what would I have done?

Inside the tile, I would have just gone through the corresponding position of the threads and loaded them into the corresponding output data by using the completely flipped x and y right. Just flipping the original x and y, but we are not doing it. I will just reiterate. What we are doing is that we are just computing the x and y in such a way that the relative position of the x and y inside the block is same, but the blocks original location has been flipped right. Now what we are really going to do?

So this is the tile which is containing the data for this block. Now this tile is going to be returned here . Now for the thread which is working in a per thread basis it has computed using this x and y relation that where is it going to write? So it is going to write here. But the question is what is here? What is it going to write here? If it is writing the location of the original point then it is the same data right.

Because it has got the the tx and ty values, then this data is here resident in the tile and I would be writing it using the x and y values. But that is not what I will now be doing.Observe this expression very carefully. So when I am reading from the tile, I am just flipping the position of thread id x and thread id y. If you see our original code, this is our usual access expression from the input data right.

Of course you have y which is the row number. Multiply by width and plus x . And of course you are writing to the y of the 2 dimension tile. The thread id $x \cdot dot \cdot y$ is there. And then you have the x for the column. But now here you use the x to check the tiles except instead of using y to go the tiles y th row. Essentially it means that you are not using data from the identical location to write to the flipped block. Because then what would happen is you just change the position of the data with respect to changing the whole block's position. But you do not do the complete transpose.

That means you do not just flip the x y coordinate into the y x coordinate. But you actually want to do that you do it in a two phase way. So what do you do? The first part is you flip the position of the block and then you write into this position using this threads but from a different position in the column. From where do you write exactly? So when you write, you load the data from the tiles tx^{th} row and ty^{th} column.

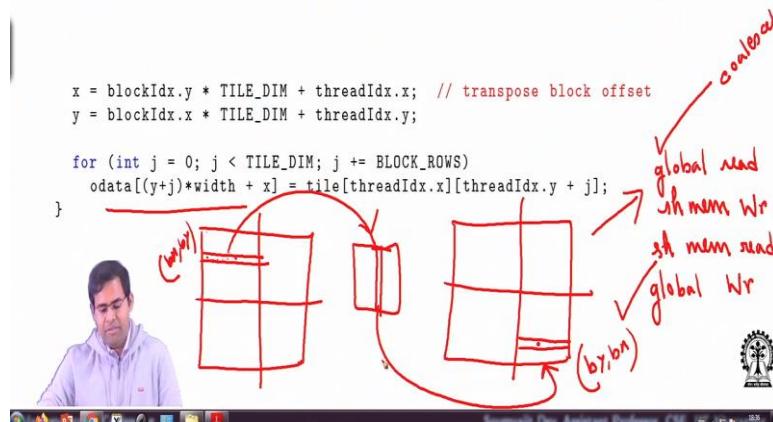
So that would mean you read from somewhere here. So then you use the earlier x offset to calculate the row offset and you use the earlier y offset to calculate the x offset right. Just see the positions are flipped. So here the first thing I do is that I use a different value of x and y here. So essentially the thread will write the same location in the block but it has a flipped location of the block.

All the blocks location is being flipped but inside the block, I am writing in the same offset location. But then I do not write from the identical location in the tile but now flip the input location in the tile. By doing this double flip, I achieve my overall objective of doing an overall flip from x y to y x . Question is why really I am doing that? So now observe the access pattern that you have.

So from the input data you are writing to the tile. So all the global loads that you have i.e. the global loads are now all coalesced because you are reading from the global memory, let us let us observe what are the reads and writes going on in this instruction.

(Refer Slide Time: 23:54)

Transpose using Shared Memory



So here you are doing in terms of memory transactions a global read and then a shared memory write. Now the global read is coalesced because you are accessing the global memory row wise i.e. consecutive locations. So this is coalesced. And then this is some simple computation of x and y but now when I am writing to flip the block, I do not just interchange x and y and make it y and x. Instead of that, I use a modified x and y which gives me the flip block. Inside the flip block, I write from the flipped location in the tile right.

Now why is this a good thing? See what are the operations that I am doing here. So now I am doing a shared memory load i.e. shared memory read or load. Now one may argue that you are loading by column from the shared memory because you are using x here and then you are using the y here right. Why is that not so bad? Well this is shared memory. So essentially you are not having coalescing but that is from the shared memory. So the load times are very small with respect to the global. So I am ok with this.

So these are not coalesced. But as a byproduct of this double flipping now when I write to the global memory that is also coalesced. So I do a global write which is again coalesced. So now the good thing of this two level transpose is that in between I have used shared memory but I have used the shared memory in a very intelligent way. So if I just rewrite this was my input space, then I am writing to the output space.

So I load from a block into the shared memory tile and then this shared memory tile which is visible to every thread, I will just repeat inside the block. It will write into the flip location into the block. But again my write location in the output data matrix is again accessed in a nice row major fashion. When I read from the global memory the warps are accessing this consecutive location in a row wise nice coalesced fashion.

These are all coalesced. The way I am achieving is I do not write the data in such a way that I load here and then when I write I get a conflict. In order to avoid the conflict I calculate the transpose position of the block. I write in the transpose position of the block. And I write in a coalesced manner and still for achieving the transpose I write from the shared memories column wise access.

So I flip in terms of the position from where I have read. So there is a 2 level flip here which is helping me to get the overall flip in the position. The first level of flip is with respect to the block id. The bx and by of the block gets flipped so from block bx and by, I am writing into the block by, bx and when I access the positions, I load data from the shared memory in a column wise fashion.

This 2 level flip helps me to achieve the overall nice coalesced behavior with respect to global memory transactions. But of course, you can see that the shared memory will have lot of conflicts here.

(Refer Slide Time 28:16)

Execute Code: TransposeCoalesced

```
nvprof -devices 0 -metrics shared_store_throughput,shared_load_throughput  
./transpose 2  
==108373== NVPROF is profiling process 108373, command: ./transpose 2  
.transpose starting transpose at device 0: Tesla K40m with matrix nx 8192 ny  
8192 with kernel 2  
==108373== Metric result:  
Invocations Metric Name Metric Description Min Max  
Device "Tesla K40m (0)"  
Kernel: transposeCoalesced(float*, float*, int, int)  
1 shared_store_throughput Shared Memory Store Throughput 81.40GB/s 81.40GB/s  
1 shared_load_throughput Shared Memory Load Throughput 1e+03GB/s 1e+03GB/s
```

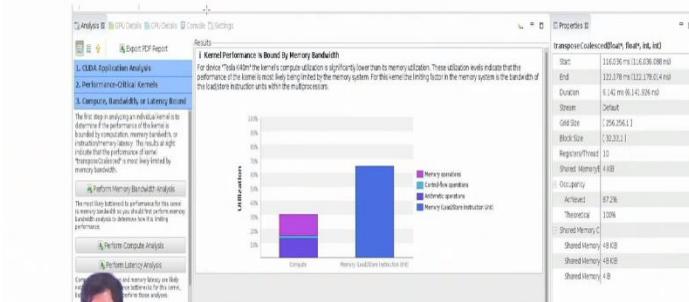


Memory Access Coalescing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

(Refer Slide Time 28:22)

Kernel Analysis



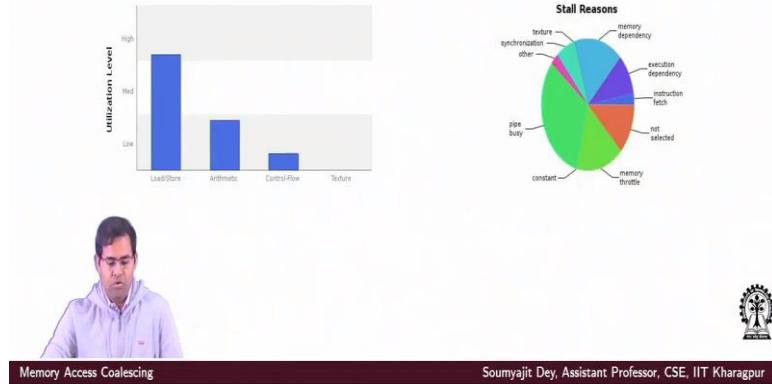
Memory Access Coalescing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So if you just use our earlier methods for doing device level profiling then we can get these statistics. We will see that this is much less if you compare with the implementation where we completely avoid using the share memory. This is much less.

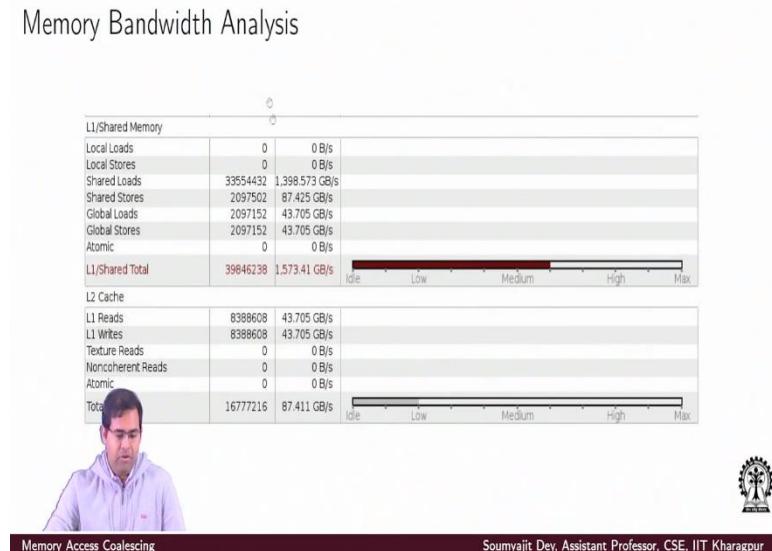
(Refer Slide Time 28:36)

Compute and Latency Analysis



And also if you go to the latency analysis, you see the amount of memory throttling is less right. Because all your global reads and writes are coalesced, and this is one of the basic reasons for that.

(Refer Slide Time 28:48)



And if you look into the memory bandwidth analysis, you see that the number of global reads and stores are small. However if I look into the shared memory access the number of share memory stores are same as the global memory loads. Why? Look back to the code you will see that just like a global memory loads they are coalesced. The share memory writes are also coalesced because they are writing by the row.

But when I load from the share memory as we discussed earlier, we loaded column wise. Now due to this column wise loading there has been conflict right.

(Refer Slide Time: 29:22)

Transpose using Shared Memory

```
#define TILE_DIM 32
#define BLOCK_ROWS 32
__global__ void transposeCoalesced(float *odata, float *idata, const int nx,
const int ny)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = gridDim.x * TILE_DIM;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

    __syncthreads();
}
```

Source: <https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/>



32x32
32x8
32x8

Now just we will just take a look back into the origin code that why effectively we did not have a loop. The reason is in general when you are writing the code, here one problem the way we discuss is we bypass the loop. If I bypass the loop then I have very small per thread activity. Now the per thread activity should not be too small since I am considering very large matrices.

So increase the per thread activity. One standard way that people follow in the examples is that you define data that share memory locations in terms of 32 x32. But you define blocks of size 32 cross 8. So essentially you have each thread working on 4 consecutive locations or 4 specific locations in a tile of data. So then your data space size remains bigger 32 x 32 but you use 4 less number of threads for doing the operation right.

Now if you are doing that then the loops will actually be useful. As you can see what will happen is inside this for loop you iterated j starting from 0. In every iteration you are increasing by a block row number. Assume that the number is 8. Then your loop really iterates. So you have each thread loading 4 data points into the tile. And then in the other loop it is writing 4 data points into the global memory.

Of course, for larger matrices, for different implementations you can have different possible factors here. Just for our simple example first, we wanted to bypass the loop. So we consider tile size and block size similar. It as I am saying that you can have a smaller size block to increase the per thread activity. So if you have a smaller sized block, then you have this tile dimension.

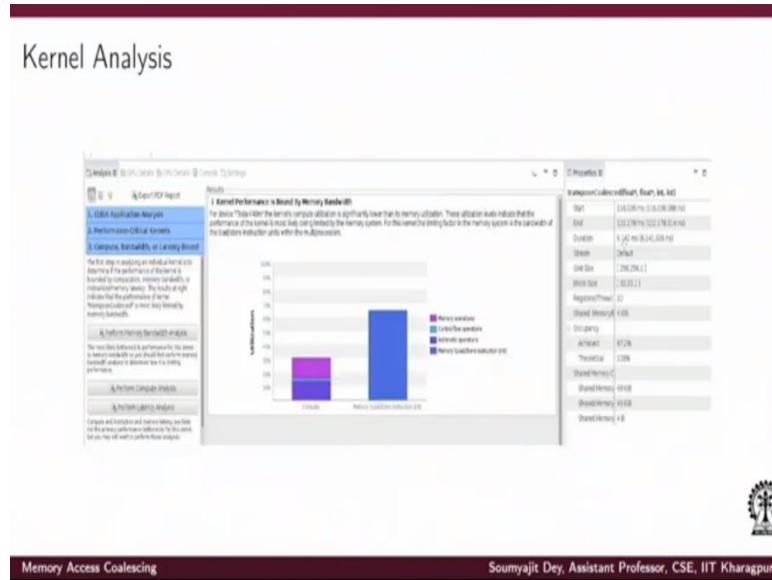
You have j iterating, increasing by 8. The loop iterates for 4 times and each thread loads 4 elements into the tile. And similarly the threads here writes 4 elements into the tile. So with this we will end this lecture where we actually showed a usefulness of shared memory and how it helps transpose kernel to coalesce both the loads and stores while doing a matrix transpose computation. In the next lecture we will take a deeper look into this. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 06
Lecture No # 26
Memory Access Coalescing (Contd.)

Hi. Welcome to the lecture series on GPU architectures and programming. In the last lecture we gave a nice example of optimization using the share memory with respect to matrix transpose computation and we saw that how using shared memory, you can have very nice global, load and store access patterns and that leads to an order of magnitude reduction in the total execution time of the transpose which we saw from our kernel analysis statistics.

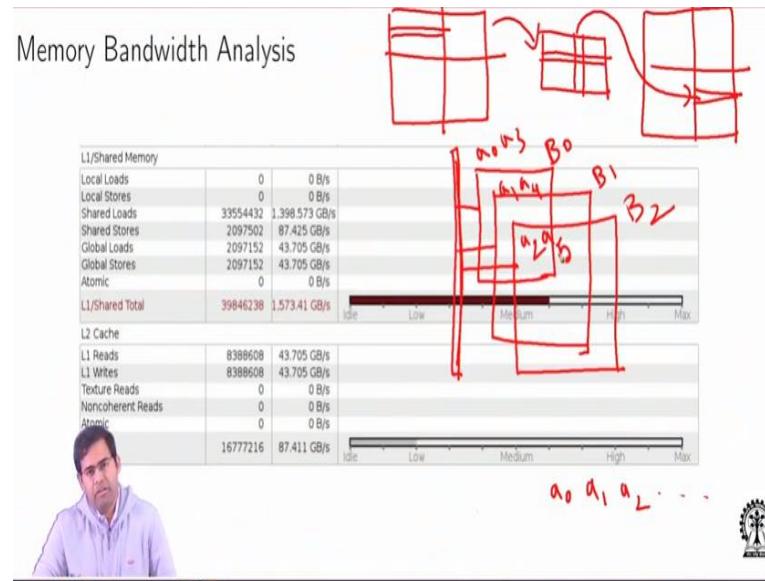
(Refer Slide Time 00:54)



By the way, we will just like to repeat these statistics that has been revealed using our system where we have our Tesla K40 GPU card. Now of course, if you are trying to have results of profiling using nvprof or some other favorite profiler for this kind of computation, the timing you are going to get of course will depend on the system you will using.

Here, we just use these values for comparison among the different optimizations we show, because we are running all the different examples or different variants of the same operation in the same system. So consider the relative performance values here.

(Refer Slide Time 01:38)



So one interesting thing that we started showing here was that since in our way of computation the shared memory load was not coalesced. So just recall our original diagram of how the different locations in the input matrix, output matrix and the tile was getting accessed. So let's say this is the tile. This is the output matrix. So from one input block you are writing here row wise and then you are loading the data from the tile column wise and writing in the flipped block location row wise.

So these loads from shared memory are not coalesced and that would create bank conflicts in the shared memory. So what is a bank conflict? The shared memory is not a contiguous piece of memory element, rather it is arranged in set of banks. So in modern GPUs as we know that the GPU families the NVIDIA defined is what they call as compute capabilities. So since we are working on a device with compute capability 3 , we have a share memory with 32 banks.

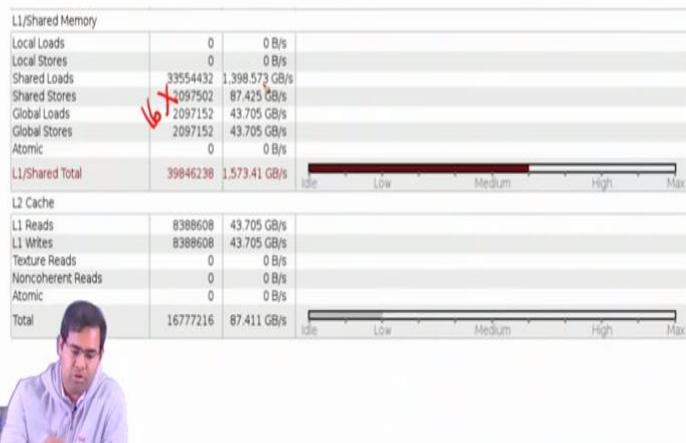
So essentially what is the idea for shared memory bank? Now a memory is arranged into these different sets. Let me redraw this picture in a nice way. So when I say that the memory segment is banked, it is not that I have a one big large chunk of physical memory. Rather than that the physical memory is split into smaller chunks of physical memory and they are all accessible in parallel by the hardware.

So let us say this is the read write block and you can access all these banks in parallel. What is the good thing about this? When you read or write data into the shared memory, let say this is bank 0, this is bank 1, this is bank 2 you can access them in parallel and if your arrangement of data, let us pick out some array it is A[0], A[1], A[2],.... they are at consecutive locations. When you are reading or writing the data, consecutive values will be distributed across banks like this right.

So when you are writing the data, consecutive location gets distributed. Now the amount of data you are writing is the same as the number of banks. Then there is no conflict in terms of the writes. So what do I mean by conflicts?

(Refer Slide Time: 04:50)

Memory Bandwidth Analysis



Think in this fashion that since I have a warp of size 32 and we have devices of compute capability of 3 or higher. I have got 32 memory banks. So the entire warp of threads if they are actually going to store data or load data to or from the shared memory in consecutive locations, all those locations are going to map to different memory banks. Because I have 32 memory banks and I am loading data from this 32 locations using a warp of 32 threads, I will simply access all of the banks in parallel and I will load the data. So that will be 1 load.

But consider the situation that when I am doing the shared memory loads in our case. So I am loading the data from a column in the matrix and column in the tile size is 32 x 32. So essentially the column will get what we call as a 32 way conflict. Because the entire column is located in

one bank. So then the question comes that how much overall number of loads would happen? In this specific case we see that almost 16X time is required right.

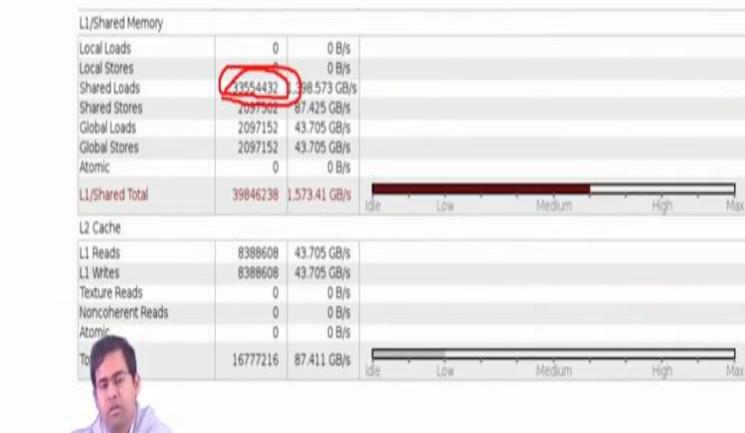
Now as we have discussed with people in NVIDIA forum, this is something that is architecture dependent. Although this is a 32 way conflict that we are getting still the memory banks have got a read write speed with which they can actually read or write 64 bits per cycle. Now since in this case, our data is each of size 4 bytes. So although the conflict is 32 way, the number of loads are increasing by the factor of 16, because the data size is half with respect to whatever I can load from the bank using its bandwidth.

Since in this case for our machine, it is 64 bits per cycle that means 8 bytes in parallel. Now what does that mean? To be more specific, suppose that I am working with 8 byte data. If I am working with 8 byte data, let us say double precision floating point. And then when I am trying to load such a column of data, the data sizes are 8 bytes with all the columns, with all the data positions located in the same bank. I have a 32 way conflict and then since the data width is equal to the bank's per clock memory bandwidth, that is the number of bits I can access for clock cycles.

So in each load operation, I will load one instance of the data. So in that case I will have 32X increase in the number of loads. In this case it is 16X right. Some other interesting activities as you can see that since we have got nice access patterns with respect to global loads and stores, both of them are small. There is no increase based on the problem that we have been discussing.

(Refer Slide Time: 08:14)

Memory Bandwidth Analysis



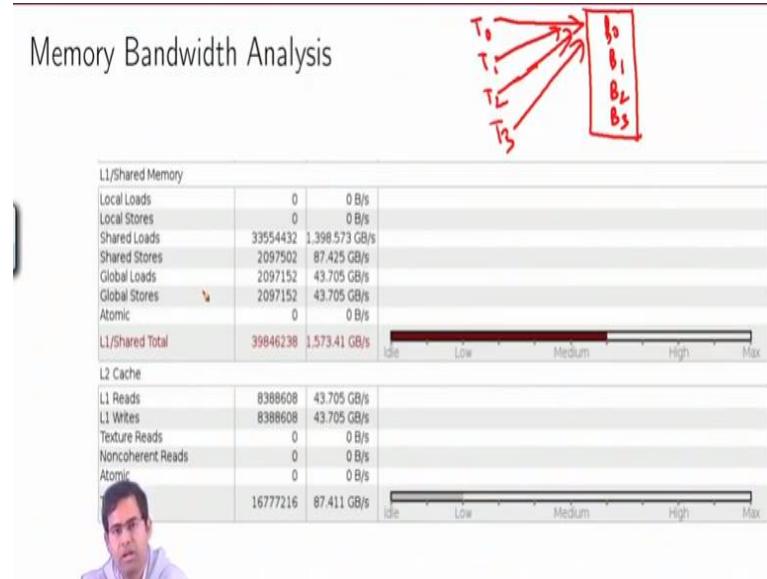
So this is the good thing about the shared memory and also we have pointed out about the bad thing due to bank conflicts. I have got a huge increase in the number of shared memory loads right. Now how do I decrease this because this again is leading to some loss of performance. The question is whether it is possible to some how rewrite the code in a nice way so that this issue of shared memory bank conflicts can be resolved.

Again I will just summarize this idea of bank conflict which is similar to memory access coalescing. So in a global memory I can access consecutive locations. Similarly since a warp can access consecutive 32 locations in parallel and so if that is the amount of data that is required, it can be fetched from global memory using one global memory transaction. It is similar for writes.

Coming to shared memory it is arranged in banks and consecutive locations i.e. consecutive data points are written by the threads in a warp in consecutive banks. So if I have 32 banks and suppose I am going to write 32 consecutive data points it will be written consecutively across the banks. But unless it is such a regular access pattern. For example, suppose I am trying to read or write in a column wise way. That means suppose all the threads in the warp are trying to write for the tile's column or read from the tile's column.

That would mean both read or write operations I am accessing in the same bank. And that gives rise to 32 way conflict. Now this 32 way conflict is also something I will like to explore a bit. So what do I really mean by this? So for that let us take an example for some other cases of conflict.

(Refer Slide Time: 10:17)



So consider some other threads, let us say thread id 0, thread id 1, thread id 2, thread id 3 like that and let's say this is my set of memory banks B0, B1, B2, B3 are trying to write data in a column wise way or in general read or write in a column wise way. That means that suppose all the threads in the warp are trying to write for the tile's column. That would mean for both read or write operations I am accessing the same bank. And that gives rise to 32 way conflicts. Now this 32 way conflict is also something I really like to explore a bit. So what do I mean by this? So for that let us take an example for some other cases of conflict. So consider some other thread so let say thread id 0, thread id 1, thread id 2, thread id 3 like that and let say this is my set of memory bank B0, B1, B2, B3. Now if the access pattern is like this regular then there is no bank conflict.

Now consider some other access pattern for example let us say the threads access in strides of 2. So T1 and T2 access separated by 1 hop. Sorry T0 and T1 accesses are separated by 1 hop. And then lets say T2 and continue this pattern. T1 and T2's access is again separated by 1 hop. And then T2 and T3's access its separated by 1 hop. So let us consider this pattern. Now let us understand what is the number of bank conflicts that I get.

Since I have got 4 threads accessing 2 banks and for each bank, I am looking for 2 transactions, so this makes a 2 way conflict. Now consider this pathological scenario that every bank is considering to access data from bank 0 which happened in our example. We are accessing the tile's column and that would mean that the tile width is equal to the number banks. So if I do the modular mapping, then all the data in the column will fall into the same bank and that would give rise to this scenario. So considering the whole number of banks here, I have a 4 way conflict.

Similarly for this problem we have a 32 way conflict. So this is how the idea of the share memory conflict builds on.

(Refer Slide Time 12:55)

Using Shared Memory: Simple Copy

```
__global__ void copySharedMem(float *odata, float *idata, const int nx, const
                             int ny)
{
    __shared__ float tile[TILE_DIM * TILE_DIM];
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = blockDim.x * TILE_DIM;
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] = idata[(y+j)*width + x];
    __syncthreads();
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x];
}
```




Scouvalit Dev. Assistant Professor, CSE, IIT Kharagpur

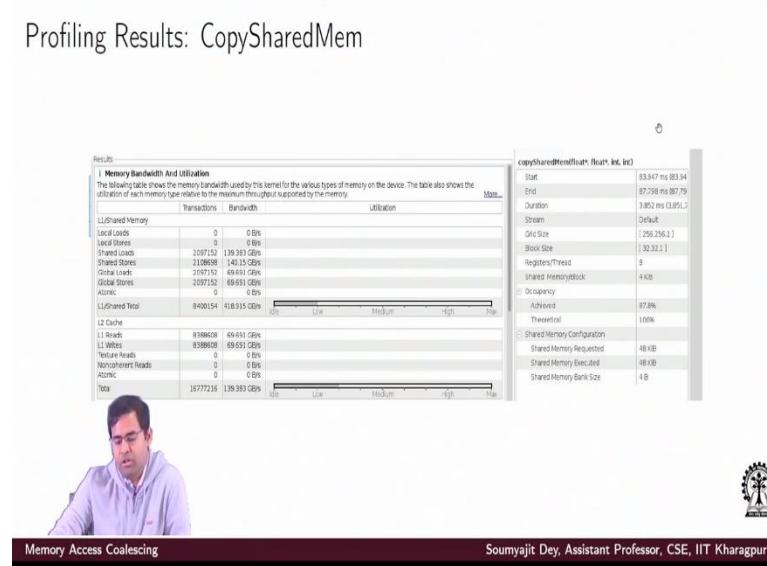
Now just to get some idea that suppose I want to find out some way to resolve the bank conflict. And I want to compare that with the normal share memory copy operation. Before we do a share memory bank conflict resolution, let us look at some performance measures with respect to a simple copy operation. So you have a input matrix. You access that input matrix using the idea of tiles.

And then I mean so as you can see the similar codes that we have again here, where we continue with our definition of tiles and blocks. I can have blocks which are addressing multiple tiles or 1 thread block contains the same number of threads as in a tile. Both are fine. In one case the loops will be active and in other case the loop will only iterate once. But what is going on here?

This is a simple copy kernel i.e. the data is copied tile wise. Input data gets in to a tile and then that tile full of data goes into the corresponding output matrix.

So these are the simple accesses we are doing here. Now we want to compare this simple copy kernel with the situation of optimizations where we are doing a bank conflict resolution.

(Refer Slide Time 14:23)



So, first of all we make an observation that if I am just doing memory copy from one input matrix to output matrix, what is the time taken if I am using shared memory? As you can see it is some 3.8 milliseconds which is much smaller if I compare it with my results on shared memory based transpose which was 6.142. In both cases we are using shared memory but I can attribute this difference due to this bank conflict Why?

Because in both cases my global memory loads and global memory stores are all coalesced. Here also as you can see that the global memory loads and stores are all coalesced along with shared memory loads and stores. So my problem with the code where I used the shared memory but I computed the transpose was that the shared memory has bank conflicts while doing the loads for writing in to the output array. So how do I resolve this bank conflict is the primary problem we want to solve it here.

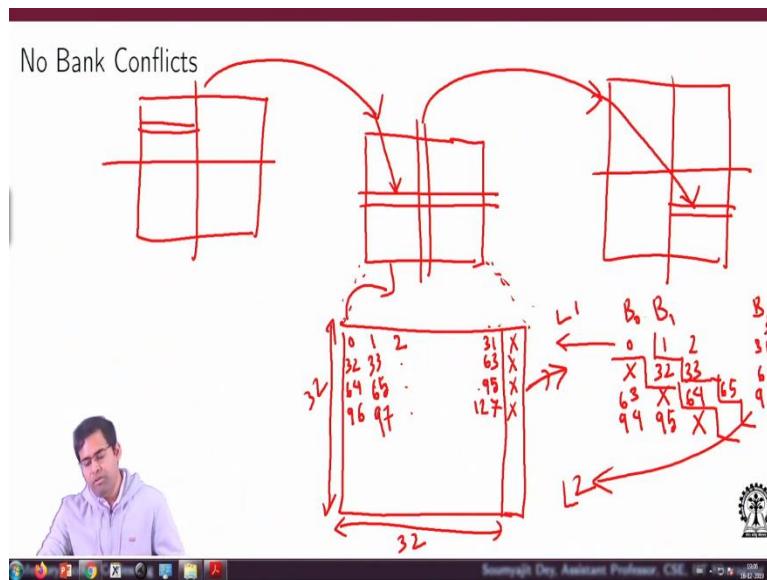
(Refer Slide Time 15:27)

No Bank Conflicts

```
__global__ void transposeNoBankConflicts(float *odata, float *idata, const int nx, const int ny)
{
    __shared__ float tile[TILE_DIM][TILE_DIM+1];
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = blockDim.x * TILE_DIM;
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];
    __syncthreads();
    x = blockIdx.x * TILE_DIM + threadIdx.x; // transpose block offset
    y = blockIdx.y * TILE_DIM + threadIdx.y;
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

So the idea is very simple keep the code exactly same as it was. Increase the tiles dimension by 1 in 1 direction. But the question is how does this really help?.

(Refer Slide Time 15:44)



So let us try to figure out that what is the effect now if I increase the tiles dimension by 1. So these are access patterns that we have already discussed. But now we are trying to figure out that for this tile, when I draw a magnified picture of tile, this dimension in the x direction has been increased by 1. So let us try and figure out. First of all I just like to repeat we have just increased the dimension and nothing else has changed in the code.

Which means I do not really use this extra location for copying data by any of the threads or I do not really load anything from this location and write back somewhere. So this is the last column which I do not really access. I do not write here. I do not read here. But I use the rest of the 32 x 32 locations for whatever earlier operations I was doing. Basically all the same operations

So I have written the data here. Let us understand for one block. I have done a row wise access and returned the data row wise right. That would mean the threads and the corresponding locations if I just write the ids of the locations, are getting mapped nicely like this. I do not write anything here and that is how it continues right. This is what is going on right. Point now is what happens to the arrangement of the data in the banks?

Now in the x dimension of the tile I am mapping all these locations to 32 banks and I have 33 locations to map per row. So in terms of the banks how really is this mapping happening?. So the data for the location 0 goes to bank 0 like this up to bank 31. And then this location is mapped to 0 right and that has changed. So now I try to map this layout in terms of the shared memories actual layout.

So let us rewrite the bank locations B0, B1,... I am just writing the shared memory banks. So I have up to B31. So I load the 0 as first data, second data then up to thirty first data. And then I have a don't care value in B0, then the thirty second data in B1, thirty third data in B2 like this up to sixty third location right. And then now sorry. So since the thirty second location is in B1 what we have here, it has to be the sixty second location.

Please understand this. This is very crucial. Since I have been putting a don't care here , now your B0 starts getting mapped from B1 location right. So sorry here it should be shifted to 62. But then where does 63 go? 63 would now get mapped in B0 and then after that there was a don't care. I am removing this. It was initially drawn just to understand the banks. But since I have a physical picture just to avoid the confusion, we removed this.

This is the enlarged view of the tile and this is a corresponding mapping in terms of the actual share memory banks. So sixty third data point is in bank 0 after that we have this do not care. So then 64 data point is then in bank 2. Now if I continue like this this is in B2. So what should we

have here? Here we should now have the ninety third data point. So that would bring the ninety fourth and ninety fifth data points in banks B0 and B1 and here you get a don't care.

Now consider what happens to the load instructions. So I am doing a load by column. So when I am trying to access the data, look at the loads that will happen. So in the first load, I get all the data. So essentially first of all let us point out some data points that I want to load. Assume that I want to load from this column. Because just to keep consistency, let us go back to the original code and see what we were really loading.

This is the code without bank conflict and it is just that I am loading from a column right. So as a representative column let us just take the first column which was in our original implementation where I did not increase the width of the tile. All these data was going to bank 0. Data at 0, 32, 64, 96 all were going to bank 0. But now what is the location as you can see 0, 34, 64 like this it is all now distributed right across the banks.

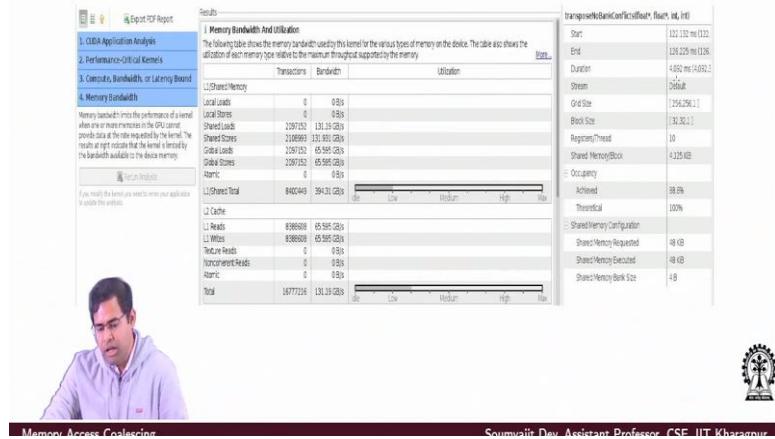
So you have 0, 32, 64 and like this up to the n everything is distributed across the bank. They all get out with 1 load right. Similarly next what do I get? In the next column what do I have? 1, 33, 65, 97. And similarly like that this is the output that you will get in 1 load . And in the next access you are going to get all the data in the second load and so on henceforth.

So now due to this insertion of an extra dimension -an extra column in the tile I have got memory padding in the shared memory and this memory padding is actually removing the conflict that was happening in each load instruction of the column in the tile when I look at that access in terms of banks. I am always accessing all the banks parallelly. So I do not have any conflict by loading the values for the column from the bank shared memory.

So this work like magic but it is only that simple. All you would do is you modify the tiles definition by doing a padding of an extra column. It changes the entire arrangement in the bank. So every column of the data actually is now diagonally distributed in the bank and due to the distribution you have that column uniformly distributed across different banks and they can all be accessed using a single load instruction.

(Refer Slide Time 25:23)

Profiling Results: No bank conflicts



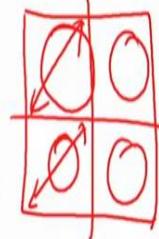
So that is the wonderful optimization I would say. With that we have very reduced execution time. As you can see it is almost 4.1 milliseconds and the all the shared loads, shared stores, global loads, global stores all are coalesced. And as you can see this is almost similar in execution time. It is only 3.85 with the normal copy operation using share memory. So here you have successfully resolved all the bank conflicts and your code executes with very minimal execution time and it is almost comparable with a normal simple copy operation without doing the transpose.

This is how if we understand the memory hierarchy correctly, we can remove bank conflicts and do better optimization with respective data.

(Refer Slide Time 26:14)

Transpose Fine Grained

```
__global__ void transposeFineGrained(float *odata, float *idata, int width,
                                     int height)
{
    __shared__ float block[TILE_DIM][TILE_DIM+1];
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index = xIndex + (yIndex)*width;
    for (int i=0; i < TILE_DIM; i += BLOCK_ROWS)
        block[threadIdx.y+i][threadIdx.x] = idata[index+i*width];
    __syncthreads();
    for (int i=0; i < TILE_DIM; i += BLOCK_ROWS)
        odata[index+i*height] = block[threadIdx.x][threadIdx.y+i];
}
```



With this we will like to end. Before that we would like to revisit some other example. If you remember the way we spoke of transpose we did a 2 level optimization in the transpose. The first computation we did was just to change the x and y in the axis. In a such a way you do not completely flip the x and y and y and x. But you compute the x and y for the original transpose kernel using shared memory such that they represent the flipped block's location.

And then in the flipped block location, you flip while reading from the tile and the reason for doing that was that you achieved very nice patterns with respect to read and write. But you did a overall matrix transpose. Now if we require that you do not need the entire matrix to be transposes, you only need to do a tile level transpose. What initially I mean is so suppose you have this entire matrix. You do not want to change the relative position of blocks. You just need to do a local transpose here inside the tiles. You want to flip inside the tiles.

So that would mean you do not do any re computation of the block ids. But rather you access the data from the input, load to a tile and then you read from the tile column wise and write to the output data, without doing a intermediate flipping of blocks. It will just repeat. If you remember optimization which is here you had a 2 level thing. First I flip the blocks and then I flip the location of reads from the tile. Consider this is not done.

So you do not flip the block. So you are reading and writing from the same block but while reading from shared memory tile you have flipped that means you read column wise. So that

would effectively do a fine grain transpose as we call it here. So what essentially I mean is I do not do a local transpose. That means I do not change the block position but inside the same block I am now reading it in a different way and writing it back.

So I am effectively just inside the same block and I am doing the transpose operation. So you compute the index to read and then you read the data. In our definition here we are defining block as the tile using the name block here and then from this you write using the flip pattern that we discussed earlier. Suppose if we do not want to do this. But you want to do a block level transpose that means you just flip the location of the block but you do not change the internal arrangement of data.

(Refer Slide Time 29:35)

Transpose using Shared Memory

```
x = blockIdx.y * TILE_DIM + threadIdx.x; // transpose block offset  
y = blockIdx.x * TILE_DIM + threadIdx.y;  
  
for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)  
    odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];  
}
```



Memory Access Coalescing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Again if I go back here so if I do not do this operation, then I do not flip the position of the block. Suppose I do these operations but then I do not do this flipping here. I do a normal read write What would that mean? I flip the location of the block by this. But then I do the normal read write. So that means inside the block relative offsets of the location do not change. So that is what we can call as a coarse grain transpose.

(Refer Slide Time 30:04)

Transpose Coarse Grained

```
__global__ void transposeCoarseGrained(float *odata, float *idata, int width,
                                         int height)
{
    __shared__ float block[TILE_DIM][TILE_DIM+1];
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;
    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;
    for (int i=0; i<TILE_DIM; i += BLOCK_ROWS)
        block[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
    __syncthreads();
    for (int i=0; i<TILE_DIM; i += BLOCK_ROWS)
        odata[index_out+i*height] = block[threadIdx.y+i][threadIdx.x];
}
```



Memory Access Coalescing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So for fine grain transpose, we have an execution time as given here and then for the coarse grain transpose as we discussed I computed the index_in first i.e. how to read. And then I flip the block locations using the original code segment I showed, and I use this flipped block location to compute the index_out to write and then I do a normal read write.

I read from idata using index into the block or the tile and then from the tile, I use a normal access. As you can see there is no flipping while reading and writing from the tile. You directly write to the tile. You directly read from the tile. You use index_in to read from idata, you use index_out to write to odata. Now the difference is index_in is the original usual consecutive location index.

But when I use the index_out, I have flipped the blocks using the first part of the switching. But I am not using the second part of the switching. Here, I do not read by column. I read by rows from the tile. So this is the coarse grain transpose. That means this block comes here and some relative location of data remains same with respect to the block. So these are also associated operations that you can look into and we also provide the execution time and other things of that. You can actually look into these data points and try and compare. With this we like to end this lecture. Thank you.

GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 06
Lecture No # 27
Memory Access Coalescing (Contd.)

Hi. Welcome to the lecture series on GPU architectures and programming. So in the last lecture we have seen certain examples of how coalesced access on the shared memory was helping us to do nice optimizations with respect to computing transpose of a matrix. So we figured out that with possible optimizations that could be done, we were able to achieve almost similar execution times which shared memory based transpose computation.

(Refer Slide Time: 01:01)

Partition Camping

- ▶ Just as shared memory performance can be degraded via bank conflicts, an analogous performance degradation can occur with global memory access through 'partition camping'.
- ▶ Global memory is divided into either 6 partitions (on 8- and 9-series GPUs) or 8 partitions (on 200-and 10-series GPUs) of 256-byte width.
- ▶ To use global memory effectively, concurrent accesses to global memory by all active warps should be divided evenly amongst partitions.
- ▶ partition camping occurs when: global memory accesses are directed through a subset of partitions, causing requests to queue up at some partitions while other partitions go unused.



As is the case with shared memory based simple copy operation. So this also highlights the architectural issues that one has to keep in mind while writing programs for the GPU architectures. Now, upto this point of time, the memory conflicts that we handle were of 2 types- One was how to coalesce global memory accesses and another was how to remove bank conflicts while accessing the shared memory.

Now there is a related problem with respect to global memory which is also relevant in this context and it is known as the problem of partition camping. So this is again something related to global memory and we thought it would be good to discuss this after we discuss this shared

memory bank conflict because the idea is similar in that sense. So just like we have this notion of bank conflicts in shared memory, the reason being this shared memory is divided across banks in order to facilitate parallel access.

Similar ideas hold for global memory that the global memory is also divided into certain partitions. So depending on the GPU series, there can be the different number of partitions. So it can be either a 6 way partitioning of the global memory or it can be an 8 way partitioning of the global memory. So we will consider the 8 way partitioning where for each partition we consider that the memory is 256 bytes wide i.e. I can read a chunk of 256 bytes from 1 partition in the global memory.

So that would also mean that if I consider an 8 way partitioned global memory, I can read 8 chunks of 256 bytes in parallel from the global memory. So what we like to do is to access the global memory effectively and facilitate as much concurrent access as possible to get data from the different partitions in the global memory in parallel. Now just like we had this issue of bank conflicts, we have the similar issue here. The name is different. It is called partition camping So why will that occur?

It occurs with the global access memory are directed to a subset of partition just like imagine multiple threads in a warp are executing and they are trying to access the same bank of a shared memory. Now just take the idea at a higher level. Consider that you have different SMs and blocks executing in the different SMs and they are trying to access in parallel the same global memory partition that would actually queue the corresponding access requests and the access will be slow as a result of that.

So instead of considering access of threads inside the warp for the single bank of shared memory you go to the higher level. I am repeating here that you consider multiple blocks which are executing in different SMs and as we know, each SM as a memory controller based interface with the global memory and each of these blocks are placing their request to their respective memory controllers for accessing the global memory. And it happens to be the case where those requests are going for the same partition.

Now inside the same partition, I can only read in 1 transaction, 256 bytes. So all those queries we get queued and it will create this issue of partition camping.

(Refer Slide Time: 04:47)

Partition Camping

- ▶ Since partition camping concerns how active thread blocks behave, the issue of how thread blocks are scheduled on multiprocessors is important.
- ▶ When a kernel is launched, the order in which blocks are assigned to multiprocessors is determined by the one-dimensional block ID defined as:
`bid = blockIdx.x + gridDim.x*blockIdx.y;`
 - a row-major ordering of the blocks in the grid.
- ▶ Ref: "Optimizing Matrix Transpose in CUDA" - Greg Ruetsch, Paulius Micikevicius
- ▶ Ref: "High-Performance Computing with CUDA" - Marc Moreno Maza



So we will summarize like this that since the partition camping concerns how active thread blocks behave, the issue of how thread blocks are scheduled on the multi processors is important. As we are discussing that the partition can be accessed in parallel by these SMs. So what really matters is how the blocks has been distributed across the SMs? Now how does it get done? So when a kernel is launched, the order in which blocks are assigned to multiprocessors is determined by a single dimensional block id.

So suppose you have a 2D block. So if you map this arrangement of 2D blocks into a row major ordering of blocks. So then you get a unique block id following this relation. So essentially we are linearizing the 2D arrangement of blocks to get a total order on a blockid's and this single integer total order will actually tell you how the blocks are getting distributed. So the following are round-robin arrangements, the blocks will get distributed following this bid value.

(Refer Slide Time: 06:01)

Partition Camping

- ▶ Once maximum occupancy is reached, additional blocks are assigned to multiprocessors as needed
- ▶ How quickly and the order in which blocks complete cannot be determined¹
- ▶ So active blocks are initially contiguous but become less contiguous as execution of the kernel progresses.

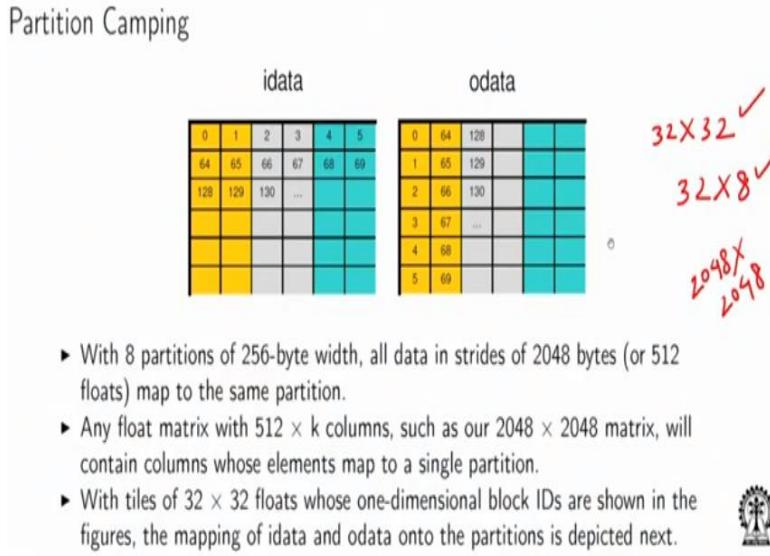
No so in this way once blocks get distributed across the SMs. I am doing a data crunching over significant size data so that all the SMs have got fully full engagements and each SM have been mapped with multiple blocks. So when I have reached this maximum occupancy, additional blocks are assigned to the multiprocessor as it is needed.

Now suppose I have this assignment of blocks done, but then there is no control. So let us assume that I have a set of blocks assigned to multiprocessor 1 ,set of blocks assigned to multiprocessor 2, set of blocks assigned to multiprocessor 3 and so on. But then as we had discussed earlier, there is absolutely no control of execution ordering of the blocks across the SMs. Because the high level scheduler of the GPU system is distributing the blocks across the SMs. Inside the SMs, you have a low level scheduler which is deciding or which is actually forming the warps from the blocks and is dispatching the warps to execute across SP cores.

So you do not have really any control over what blocks progress. As we have discussed earlier inside an SMs there will be a low level warp scheduler who will follow some method. It will dispatch warps by resolving dependences but across SMs there is no such control and it is done intentionally to achieve maximum parallelism as much as can be extracted from the application.

So how quickly and in which order the blocks get executed cannot be determined. But the active blocks are initially contiguous, I mean the active blocks that are initially contiguous but become less continuous as the execution of the kernel will progress. That is for a fact.

(Refer Slide Time: 08:20)



- With 8 partitions of 256-byte width, all data in strides of 2048 bytes (or 512 floats) map to the same partition.
- Any float matrix with $512 \times k$ columns, such as our 2048×2048 matrix, will contain columns whose elements map to a single partition.
- With tiles of 32×32 floats whose one-dimensional block IDs are shown in the figures, the mapping of idata and odata onto the partitions is depicted next.



But now let us try and understand how the blocks across the SM's are going to access the data in the global memory. So for this problem, we consider a situation that so for this problem we consider this kind of a definition of threads - tiles and blocks. So if you remember from our earlier examples, we are considering that the data tile size is 32×32 . And we have blocks of size 32×8 right so it is a 2D arrangement here.

Now with this kind of a setting we consider that with this kind of tile size for the data and with this kind of dimension of blocks we are trying to do a transpose kind of computation on a matrix whose size is 2048×2048 . So if I am doing it like we let us try and identify how data maps onto the global memory. So since in the global memory i have this division of 8 partitions. Let me just number the partitions 0, 1, 2, 3 like that.

Each of the partitions are 256 bytes wide. So sorry this is not the numbering of the partitions. The partitions have been shown here using the colors and we will define the numbering here. So the first yellow color part is basically partition 0 the second one is partition 1 and like that. Now since each of these partitions are 256 bytes wide, I have data sitting in strides of 2048 bytes or for that matter 512 floats. Because each float means 4 bytes. So multiply and get 512 times 4 which is 2048.

So if I have 8 partitions, each partition is 256 byte wide. So if I just do this calculation and this then multiplies to become 2048. So I have a large sequence of data points. So every data point and whatever is its partition, that data point index + 2048 bytes would give me the data starting from that data point to the data point which is sitting 2048 bytes further from that. They both will belong to the same partition. I mean it is really obvious because each partition is again 256 bytes wide.

So you get this stride size of 2048 just by doing this simple calculation of the total width offered by the 8 partitions. Now what does this really mean in terms of the data types? For example let me consider a float matrix, let us say it is size is $512 \times K$ right. So I have 512 rows. So 512 rows means that the row is 512 times 4 bytes so that is 2048 bytes wide. So essentially each column will get mapped to a unit partition or a single partition.

So I have a 2048×2048 matrix where essentially 2048 is a multiple of 512. Again I will explain what is going on. I have 8 partitions, each partition is 256 byte width. So overall width is 2048 bytes. So I can accommodate 512 floats into 2048 bytes. So if I consider a float matrix with 512 cross K number of columns then essentially what is happening every column of data maps to a single partition.

Now the logic will hold for a matrix of higher dimension where the dimension is a multiple of 512. Right now since 2048 is also a multiple of 512, so for our target matrix of this size the columns will also get mapped to the single partition here. Now since I am considering data tiles of 32×32 floats they are accessed by 1 dimensional blocks. So we are considering our earlier examples where if you go to the block dimension definition it was 32×8 . So we are considering that I have these many threads to process the tile of this size.

Now if I do a mapping of the tile with respect to the block id, then I try and figure out how exactly these tiles are getting distributed across the partitions. So I have tiles of 32×32 size whose 1 dimensional block ids are shown in this figure. I will just repeat the blocks are smaller because inside the block, we have a set of threads and each thread accesses 4 elements in the tile but since 1 block takes care of accessing 32×32 floats, I represent that tile size with the corresponding block id. Why?

Because the number of threads in the block does not matter in this context. I am trying to see what is the memory consumption by that block. So that block of small size - the block of id 0 or 1 each of them are working on a tile size of area 32 x 32. Now since these are 2D maps let us understand how they are distributed across partitions. So in the x direction I have 32 data points.

And so each of them are of size 4 bytes so I have 128 bytes in the x direction for each tile and since the partition is 256 bytes wide, I will just repeat this part again. Partition is 256 bytes wide. Each tile in the x direction holds 32 floats. That would mean it is 32 x 4 bytes so it is 128 byte wide. That means I can accommodate 2 consecutive tiles in 1 partition so there is a 2 level argument here.

First one, I would say is that why am I mapping this tile with a block id. I am just writing 0 for tile 0. Because I am saying that this 32 x 32 tile will be processed by a corresponding block of size 32 x 8. But what is the way in which the tile is going to be represented here? So the tile is in 2 dimensions so we map to the memory in the x direction. I have 32 floats being arranged. So they would consume 128 bytes.

So I can have 2 consecutive tiles mapped in this memory. In this way my input data that is the high data matrix considering it the very big matrix is getting mapped like this. So how many tiles do I really have in the x direction? Of course that would mean you divide 2048 by 32 and you have got 64 tiles. So the tiles with id 0, 1, 2 like that. I am writing the corresponding block id's here which are going to work in these tiles. So you map these tiles across the partitions 0 to 63 .

Now so overall, I have got 8 partitions now. So in each partition, I have got 2 of them sitting 0,1 and then 64, 65 and like that. So this is how the original arrangement is here. So this is how it is going to continue but the operation that I want to do on this data i.e. in the global memory is to perform a transpose operation. So the expected pattern in which the output data should be organized has to be like this. As we can see just take a look into the odata part.

So again just to summarize, the overall problems faced here I have the input and the output matrices here and they are arranged in idata and odata. I am just trying to show how they map across partitions. So overall I am just figuring out that inside each partition I can have 2

consecutive tiles here and in that way if I continue I will get this kind of a distribution . So I have overall these 8 partitions and all data which is in strides of 2048 will just keep on repeating and for each tiles I have this 0 and 1.

And here I am trying to show 1 possible arrangement. Of course this is not this. The indexes are just trying to represent how the 1 dimensional blockids get mapped here. Now let us try and figure out that for doing a transpose computation, what is the corresponding overhead in terms of partition camping from the global memory side?

(Refer Slide Time: 19:34)

Partition Camping

idata						odata					
0	1	2	3	4	5	0	64	128			
64	65	66	67	68	69	1	65	129			
128	129	130	...			2	66	130			
						3	67	...			
						4	68				
						5	69				

- ▶ Concurrent blocks will be accessing tiles row-wise in idata which will be roughly equally distributed amongst partitions
- ▶ However these blocks will access tiles column-wise in odata which will try to access global memory through just a few partitions.
- ▶ Just as with shared memory, padding would be an option (potentially expensive) but there is a better one ...



So earlier we have been looking into the problem that 1 block is working in the shared memory and how much optimizations are getting possible. But now we are taking the global view of the problem. So I have got concurrent blocks executing across SMs and they are accessing the tiles from the global memory. Now let us understand what is the access pattern for the input data matrix and what is the access pattern for the output data matrix right.

So concurrent blocks will be accessing the tiles row wise in the idata matrix and in this as we can see this roughly is equally distributed among particles right. As we can understand the block id's the concurrent blocks that are executing zeroth block, first block, second block, third block like that each of them have got a working set which is the 32×32 tiles size and 2 consecutive tiles map into the same partition right.

So since it is the row wise access, I can expect that in a average scenario they are consecutive block ids and their accesses are almost equally distributed among the partitions. So I mean of course after some time they lose the synchronization in terms of which blocks are executing in which SM. But since the blocks are distributed across SM's and there is no fine grain control over their execution order roughly from each SM or whatever partition is being accessed for a block, since the access have to be concurrent.

On the average I would expect that they are roughly equally distributed among the partitions. Again we will just try to understand that in a perfect scenario it will be equally distributed. Otherwise also it will be roughly equally distributed. The reason being because I do not have concurrency control across the SMs. I do not have control over the block execution ordering across SMs.

We are expecting that concurrent blocks that are executing across these SMs, have accesses to the partitions that are equally distributed. Concurrent blocks will access that tile data row wise. Now if I consider how these concurrent blocks across SMs are going to access the output data, again I do not have any control over concurrent blocks across the SM's. But they will access the data column wise in odata.

Now let us understand what is happening here. So I have got blocks from different multiple processors which are going to access the odata and as we understand the blocks are roughly distributed equally across the SMs. So they fetched row wise and they are trying to write column wise in the overall arrangement of the matrix and that will also get reflected here in terms of the global memory accesses. These blocks will access the tiles column wise in odata. By columns here we mean this column of tiles sitting in the same partition.

So I will just summarize. I have different blocks executing across SMs. They are accessing these tiles in parallel right. But when these blocks are going to access the data in odata matrix, for writing it out, the writes are going to happen in this order. Why? Because as we can see that these are the consecutive tile id's and they are sitting in the same column. Here column does not mean column of data but the column of tiles indexed with the column of tiles which are indexed with the corresponding block ids. Be very careful here right.

So since these tiles will be accessed column wise in odata, this will typically access only a few partitions. Again we are repeating here we cannot say strongly that the access will be on exactly the same partition. In an ideal scenario yes, because I have exactly the consecutive block ids which are executing across SMs and they are accessing the same column of tiles in odata which means all of them are accessing exactly the same partition.

But since I do not have any control over the execution order of the blocks, but still I can expect that their execution order will not be exactly distributed, it will be a bit different from the perfect column access. But still it is not going to be distributed across all partitions. But it would be restricted to much smaller number of partitions. So, overall the philosophy here is that when you are going to read from idata, you are actually reading in a perfect scenario from all partitions in parallel. In an actual scenario, you are reading from a large number of partitions in parallel.

I mean you may be reading from very small number of partitions in parallel. Now just like shared memory bank conflicts, you have this issue of partitions camping here because from each partition you can read only this width of 256 bytes. So essentially you will read the first row here and the first row here in the consecutive tiles right. Now what happens if I consider the optimization with this for resolving the conflict of shared memory? Well what was the optimization?

The optimization was that okay you just increase the tile dimension by 1 that we did not really use. It was a don't care dimension. That is what we call as padding the memory with data so that the access actually resolves the bank conflicts. But if we are going to use that same concepts here it is going to be potentially expensive. Why? Because instead of adding a single column you are going to add full tiles here.

So that is 1 problem. But also you do not have any control over the blocks that are executing the scheduling. So you are not also sure how good that approach is going to work in this case. We can do it in a different way possibly. We will see how. So first of all we understand that padding can be an option here but with respect to shared memory. In this case the option is much more expensive option.

So when the writes are happening in this order. As we can see writes are accessing the same partition more frequently I mean writes across blocks y. Because as we can see that tiles are kind of arranged here in a column wise way. So that would mean multiple blocks across SMs are kind of not accessing all the tiles in all the partitions available in parallel, but they are accessing only a few tiles for few partitions.

Now why would that happen? Because as we have discussed in ideal scenario it will be the worst case. Because if blocks across SMs are executing in a perfect lockstep round robin order, then they would be accessing this kind of a single column of tiles. So it will be only accessing a single partition. But since I am not controlling the execution speed of the blocks across the SMs and their scheduling, I would still expect that their behavior would not be totally uniform and they would access only a few partitions while doing the writes on the odata.

So just to summarize when multiple blocks are going to access the idata, it is expected that lot of partitions would get accessed in parallel. But when these multiple blocks are going to access the odata, it is expected that lot of blocks are going only access a few partitions right.

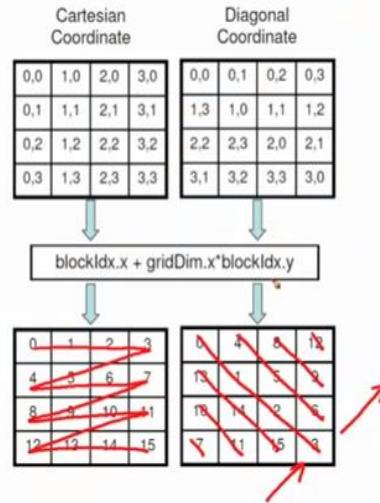
So in the first case while reading since I am kind of reading row wise data , it is expected that I will read from a lot of partitions. But when I write, I write only to a few partitions. Now to reduce this issue of partition camping we can follow the same strategy as was done in case of shared memory. So in that case the strategy was padding. By padding what we essentially did was we introduced a new column in the shared memory.

Now the good thing of introducing that extra column in the tile was that the way in which the data was written. Earlier the data was getting loaded from a tile and was causing conflicts But after we did the padding, there was full coalescing achieved in terms of both loads and stores from the shared memory. Similar option was there of course.But here we can understand it is a potentially very expensive option.

Because you are not just only adding one single column of 0's here right. It is a much more expensive option and it is possible to have a better option here. So let us see what is the better option.

(Refer Slide Time: 30:02)

Diagonal block reordering



So in this case what we do is that we diagonal block reorder now. What does that mean? So, if you look at the normal way in which the blocks are arranged in a memory i.e. the blocks are arranged as per the programming model here ,we follow our normal Cartesian coordinate. So following the normal Cartesian coordinate system the blockids in terms of x and y dimensions gets distributed in the left hand side fox of the figure

And this distribution of the Cartesian coordinates are easily linearized using this formula that has been given. And they give me a total ordering of blocks which is the row major ordering. As we can see here it is a normal row major array. And primarily due to this ordering we get the issue of partition camping with respect to the global memory. Because this order is only creating the corresponding arrangement of write access of the blocks by the respective blockids and you do not want the accesses to happen in such a way. I can actually defer the accesses by modifying the order of accesses by changing the allocation of blockids to the different tiles right.

So essentially that is what this picture is about. I am modifying the id of the block who is going to access a specific tile. So just to understand again, if you look into this figure I have a mapping in the memory of the tiles and what we are trying to ask is that which block of threads is going to access which tile. So the work is same. The work is that you to do a transpose operation on the memory. What we are now going to change is what is the block id of the thread block that is going to do to the respective work..

Now why is that going to work? Because if I change the ordering of blocks, then accordingly the schedule will also pick up blocks following the computation of the blockids. This is an important point. Just to summarize here. We understand one thing that the scheduler is going to assign blocks to SMs following this relation. So if I am going to change the way blockids are going to be fed for computing this, things will be different. So that is what we are going to do.

So we shift from this Cartesian coordinate system to the diagonal coordinate system so that we can have a different blockid computation scheme while doing the access of the memory. Essentially what I am doing is I am giving a same job to a different block. That is what we can say. Now of course, I cannot pass a different block id to the scheduler. But instead of that what I can do is that I can make the scheduler's corresponding block work on a different id memory address.

Now this is the important thing we need to understand here. So instead of having the blocks work on these tiles i.e. essentially the tile numbers, we are trying to say here that if I am following a one on one mapping of blockids with tiles and to respective positions. This is the way the tiles are going to be arranged. Now we are saying that no let it not be the case. Let the tile indexes there be different. Because essentially the tile index represents the way different blocks are going to work on them.

And we modify the way the tiles are being accessed by suitable access expressions. So as you can see from this Cartesian coordinate which shifts to something called a diagonal encoding. So what is that? So we start enumerating blocks in a different way. So I am just trying to show how we are going to enumerate the blocks. First we will do it here and then we will see how it translates to the original scheme here.

So this is my access pattern right here. I am saying that okay let me enumerate them like this - primary diagonal followed by the first secondary diagonal in the right hand side, then this, then the other diagonal here, then this, then here, then this. So essentially you start with the primary diagonal then you start moving this way and then you switch here and start moving this way and you keep on switching back and forth.

So the primary diagonal switches to the right side, moves on diagonal, switches to left side, moves on diagonal. So in both cases you are moving in this direction but you are switching among the sides once in a while. So I want this enumeration scheme for the ids. I want this to be the way in which the block should be accessing the data. So for doing that I use this diagonal coordinate scheme, so that if this formula is applied on this scheme I get this enumeration order.

Now the question is why do I have to change the scheme? Can I alter the formula? Ideally no. Why? Because we are provided with the system's own definition of variables in terms of block id's and thread id's. So what we can really do? We do not have direct control and we cannot in our programs, given a thread redefine the block id component or the thread id component values. Because it is thread id x and block id x. They are all system variables.

But what we can do is we can define a different mapping here such that the respective block follows this mapping while accessing the different locations in the memory. So just to understand this was my Cartesian coordinate system which gave me this nice linear representation of block ids. I shift to a different diagonal coordinates system on whom if I apply this same relation of $\text{blockIdx.x} + \text{gridDim.x} \times \text{blockIdx.y}$, then I get a different linearization which is more of an access to the diagonal right.

The good thing is if I have access to the diagonal, that ensures that I am writing the output data across more number of partitions and removing the problem of partition camping. We will see how. But just now see how is this diagonal coordinate defined. So observe any entry here with unequal components. So let us say this entry and this entry. You see the second component and the first component are the same. And this is true for all the other entries. So here also the second component here is 1 and the first component here is 1. So they are same.

(Refer Slide Time: 37:46)

Diagonal block reordering

- ▶ The key idea is to view the grid under a diagonal coordinate system. If `blockIdx.x` and `blockIdx.y` represent the diagonal coordinates, then (for block-square matrixes) the corresponding cartesian coordinates are given by: `blockIdx_y = blockIdx.x; blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;`
- ▶ One would simply include the previous two lines of code at the beginning of the kernel, and write the kernel assuming the cartesian interpretation of `blockIdx` fields, except using `blockIdx_x` and `blockIdx_y` in place of `blockIdx.x` and `blockIdx.y`, respectively, throughout the kernel.



So what we do is we use this kind of ideas to re-compute the new block id x. So original `blockIdx` variables are `blockIdx.x` and `blockIdx.y`. Define a new diagonal coordinate system and call it `blockIdx_y` which is `blockIdx.x`. As you can see the second component here becomes the first component here so this relation holds. And the `blockIdx.x` which is the other variable i.e. the second component here is becoming the first component here.

But what happens to the other component? So the other component here which is `blockIdx_x` is computed using this relation. Now this is the relation which is easy to check. So all you need to do is you apply these 2 relations on the original `blockIdx.x` and `blockIdx.y` variables. Here you will get this modified diagonal coding system that means you get this `blockIdx_y` and `blockIdx_x` variables recomputed to give you new values of block ids here in the x and y direction.

Now why do you really want to use it? Now you just introduce these 2 new lines of code and use this new interpretation of block id's in x and y direction throughout the kernel. So essentially at the start of your original code you replaced this `blockIdx_y`. You replace the `blockIdx.y` with `blockIdx_y` and you replace your `blockIdx.x` with this `blockIdx_x` values that you just computed here. And then throughout your program uniformly, you replace the dot x and dot y's with underscore x and underscore y's for the block id's.

(Refer Slide Time: 39:50)

Diagonal block reordering

```
__global__ void transposeDiagonal(float *odata,
float *idata, int width, int height)
{
    __shared__ float tile[TILE_DIM][TILE_DIM+1];
    int blockIdx_x, blockIdx_y;
    // diagonal reordering
    if (width == height) {
        blockIdx_y = blockIdx.x;
        blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
    } else {
        int bid = blockIdx.x + gridDim.x*blockIdx.y;
        blockIdx_y = bid%gridDim.y;
        blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
    }
}
```



Now so essentially what you do is this is your re-computation. So of course you have to decide about the re-computation in 2 different ways. Because in case the matrix is symmetric, the width is equal to height. Then this relation will hold uniformly for all block id's and blockIdx.x and blockIdx.y values. Otherwise is not symmetric. Then you have to some extra calculation which is provided here.

We are not getting into the details. We should be able to understand when we are looking into the code later on. So essentially this is the segment of code through which you do the diagonal reordering and essentially you transit from Cartesian code in this system to a diagonal coordinates system and then if you apply this linearization formula, what you get is the different access pattern of the said blocks right.

So essentially what we will do is with this you have a different value of blockIdx_x and blockIdx_y. And instead of using the original blockIdx variable, if we use these blockIdx variables, then the order that we just discussed i.e. the diagonal based ordering is the ordering in which the thread blocks will be accessing the data. So try and really think what is happening? You are not really changing the system variables. You are not changing thread schedulers or anything which is it is out of your hand.

But what you are changing is the access expression of each of the blocks. That is the most important thing we need to understand.

(Refer Slide Time: 41:36)

Diagonal block reordering

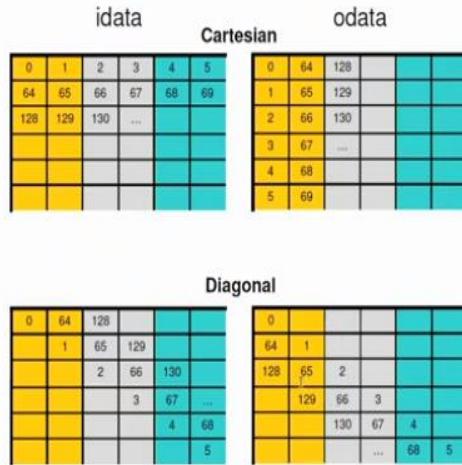
```
int xIndex = blockIdx_x*TILE_DIM + threadIdx.x;
int yIndex = blockIdx_y*TILE_DIM + threadIdx.y;
int index_in = xIndex + (yIndex)*width;
xIndex = blockIdx_y*TILE_DIM + threadIdx.x;
yIndex = blockIdx_x*TILE_DIM + threadIdx.y;
int index_out = xIndex + (yIndex)*height;
for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    tile[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
}
__syncthreads();
for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i*height] =
        tile[threadIdx.x][threadIdx.y+i];
}
}
```

Without changing anything in the original program you transform the program from one coordinate system to another coordinate system. And now you are using this new underscore x and underscore y variables to compute the x and y indexes to do the tile based computation i.e. you load from idata and you again use the tile to write to the odata. This is just the original program. Only you are using this blockIdx_y and blockIdx_x variables to do the x index and y index computation and then using the x index and y index to compute the index_out.

So this is the only modification that you have. You just start using underscore x instead of the dot x for the block id's. With this essentially what you are ensuring is that earlier your block id's mapping was exactly the order in which you were accessing the tiles based on the access expression. But now due to this alternate mapping, you have consecutive blockids. But they are making access of the tiles like this. They are going to access the tile values like this.

(Refer Slide Time: 43:02)

Diagonal block reordering



So with the same program, let us see how the blockids will get distributed when I consider its access of the partitions. So in the Cartesian coordinate system this is what was happening. The blocksids were distributed in a row wise manner and the output data came in a column wise manner. But now if we use the diagonal coordinate system this is the first input data stream. So you have the first diagonal. They came on the right hand side of the next diagonal. Then on the left hand side, you will have another diagonal like that.

And then when you transform here ,what is really happening? So essentially the diagonal switch and you get the same effect. So all that is happening is the diagonals are switching spaces. So when you are reading or writing that data, you have a nice balanced way in which the majority of the partitions are going to be accessed. Because of this diagonal scheme, this is the good thing you get right. So when you read the data, you read from majority of the partitions. Because you have the blocks with id's 0, 1, 2, 3 like that. They are going to read from this tiles right.

The blocks are going to read from these tiles. Essentially the blockids represent what tiles they are going to map and when they are going to write. And again to write following this diagonal based ordering, they are again going to write to multiple partitions in parallel which is a significant number of partitions. So that is the primary idea here that by making this shift in the coordinate system you are able to increase the number of partitions which are being used for writing the data instead of doing pure column wise write of data.

And with this we will come to a conclusion of our discussion on memory access policies. I hope it was really useful for increasing your understanding of how GPU memory system is organized and how it can be exploited for optimizing your code. Just to summarize, remember one thing when we have talked about these block id's. As I have told time and again but I think it is important also.

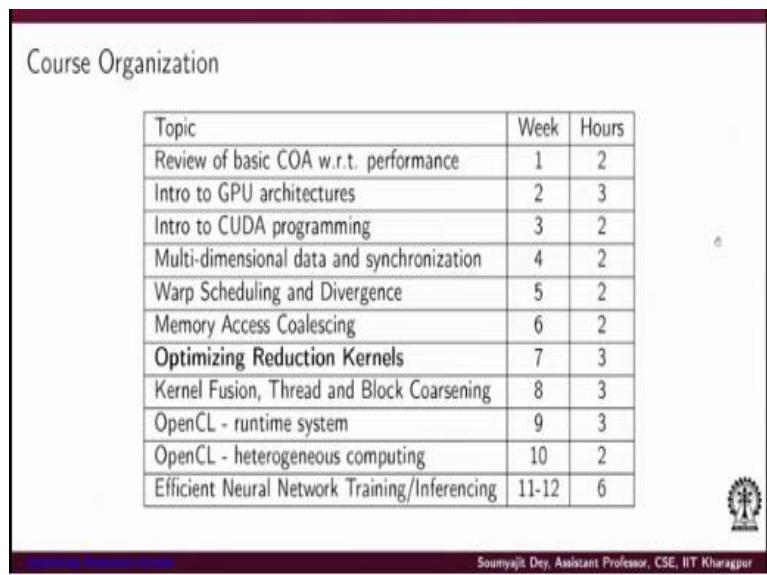
The block id's are determining which tile you are accessing and which tile you are accessing depends on what is its location in the partitions of the memory. Since we have changed the block id's ordering here i.e. the block id's linearization scheme have changed in a the way in which the partitions are getting accessed with respect to reads as well as writes. And in both cases I am accessing a lot of partitions together and that is fundamentally giving me the speed up with respect to the previous version of the code. With this we will end this lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology - Kharagpur

Lecture – 28
Optimising Reduction Kernels

Hi, welcome back to the lectures on GPU architectures and programming. In the last weeks we have been discussing more on memory access and coalesce related issues. And throughout the last few lectures while studying about warps and their scheduling, the divergence and how memory access is coalesced, we have fundamentally understood the GPU architectures memory hierarchy and how it can be used to write optimised programs.

(Refer Slide Time: 00:56)



The slide shows a table titled "Course Organization" with the following data:

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

(Refer Slide Time: 01:00)

Recap

- ▶ The Host-Kernel Model for CPU-GPU Systems
- ▶ The CUDA programming language
- ▶ Mapping multi-dimensional kernels to multi-dimensional data



So, with this background, we are going to this topic of optimising reduction kernels that means, we will try to use this concept for some regular programming jobs, the primary one being reduction kernel.

(Refer Slide Time: 01:07)

Recap

- ▶ Querying device properties
- ▶ The concept of scheduling warps
- ▶ Performance bottlenecks
 - ▶ Branch Divergence
 - ▶ Global memory accesses



So, these are the concepts which we have already covered and we will be trying to use those concepts here.

(Refer Slide Time: 01:14)

Parallel Patterns

- ▶ Matrix Multiplication (Gather Operation)
- ▶ Convolution (Stencil Operation)
- ▶ Reduction



Now, before getting into reduction kernels, we have to understand, in general, while doing parallel programming what is important is to understand the concept of parallel patterns. Now, a parallel pattern would mean specific pattern of tasks that is a specific execution order and data access order of tasks which can be found occurring very frequently in some algorithm. So, if I have a specific sequence of computation and a specific sequence of data access that is occurring very very frequently, that is what we call as a pattern, if that has lot of parallel jobs embedded in the pattern in terms of computation or communication of data. It is the parallel pattern and we can see such parallel patterns occurring in very large number of computation intensive things we do. For example, matrix multiplication, for example convolution and also the reduction operations.

(Refer Slide Time: 02:25)

Reduction Algorithm

- ▶ Reduce vector to a single value via an associative operator
- ▶ Example: sum, min, max, average, AND, OR etc.
- ▶ Visits every element in the array
- ▶ Large arrays motivate parallel execution of the reduction
- ▶ Not compute bound but memory bound



So, we will try and focus first on reduction operations. Now typically, what is the reduction operation? So, a reduction algorithm or reduction operation is primarily the generic name for any operation that takes as input of vector and reduces it to a single value via some associative operations. For example, sum, min, max, as you can see that these are all associative operations.

Suppose, I am computing the min. It is same as if I write (follow slide). These are generic name for operations which reduce a vector to a single value and for fundamental, the operation is the associative operations. These are operations which may visit large number of elements in the array, I mean, you have lot of elements and you carry on these associative operations to get to the final value. Fundamentally, we will need to visit every element in the array and perform the operation.

Now, this is interesting from the point of view when I have a very large array to work with, a very large vector, lot of values and my final goal is to reduce it to a single value. Now, why is this interesting? Because this is kind of a pathological work load, in this case as you can see that there is not much of compute operations to do but significant amount of memory operations to do.

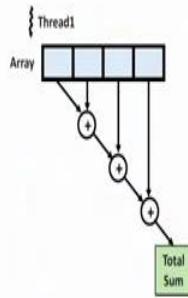
Since, there is lot of memory operations involved, we feel that most of the optimisations we have discussed earlier should have a role to play in case of reduction algorithms; and that is what makes them interesting in this context.

(Refer Slide Time: 04:31)

Serial and Parallel Implementation

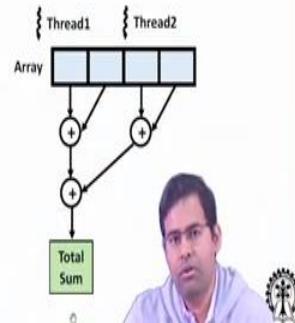
A sequential version

- ▶ $O(n)$
- ▶ `for(int i = 0, i < n, ++i) ...`



A parallel version

- ▶ $O(\log_2 n)$
- ▶ "tree"-based implementation



So, we just start with the parallel sum implementation. You have a lot of numbers and you want to sum them. If you have a sequential version, it is going to an order of n operation, if you are going to do a parallel version, which we have seen earlier in some other examples, is going to be a $\log n$ operations. Because you have threads which are working on different parts of a large array, the array is in sized and you are going to do the sum. And then, you are going to compute the sum of sums so on and so forth, so it is going to be a $\log n$ operation.

(Refer Slide Time: 05:07)

Parallel Reduction Algorithm

To process very large arrays:

- ▶ Multiple thread blocks required
- ▶ Each block reduces a portion of the array
- ▶ Need to communicate partial results between blocks
- ▶ Need global synchronization

Problem:

- ▶ CUDA does not support global synchronization

Solution:

- ▶ Kernel decomposition



Soumyajit Dey, Ass.

Now, if I am going to do this kind of parallel reduction in large array, then first I will require multiple thread blocks. Because one block can only accommodate 1024 threads, I need more than that, so definitely multiple thread blocks. Now, each block can reduce a portion of the array. Now once that has been done, I need the results to be communicated.

Because each block may reduce a portion of the array and these portions will need to be reduce further; but that would definitely need global synchronisation across blocks. Now, we know that using “`__syncthread`” kind of mechanism, we can only synchronise inside blocks and CUDA does not support global synchronisation. So essentially you have to design a reduction kernel for a block and you have to call it multiple times and use the results of each block to do the computation over the results in a significant number of iterations.

(Refer Slide Time: 06:15)

Kernel Decomposition

- ▶ Decompose computation into multiple kernel invocations
- ▶ Kernel launch serves as a global synchronization point
- ▶ Negligible HW overhead, low SW overhead

Figure from 'Optimizing Parallel Reduction in CUDA' by Mark Harris

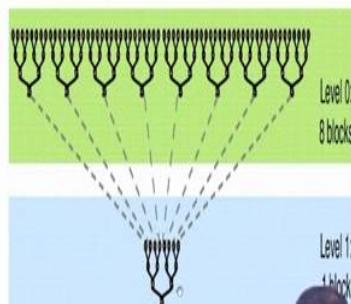


Figure: Multiple Kernel Invocation

Soumyajit Dey, Asst. Prof.

So, here our purpose is to understand, this is just a small picture we are trying to show, if you have a very large size data, it needs to be decomposed into subparts. You decompose the computation of the overall sum across multiple kernel invocations and the kernel launch serves as a global synchronisation point. So that, after every kernel launch, when you get the data back, those are the synchronisation points from which you can gather the data.

And again, launch further kernels. So this kind of sequence tree of kernel launches which we are trying to show that you have large data, with each launch you get some part reduced, and finally it comes down to 1 block.

(Refer Slide Time: 07:01)

Optimization In Reduction

- ▶ Metrics for GPU performance:
 - ▶ GFLOP/s for compute-bound kernels
 - ▶ One billion floating-point operations per second
 - ▶ Bandwidth for memory-bound kernels
 - ▶ Rate at which data can be read from or stored into memory by a processor
- ▶ Reduction has very low arithmetic intensity
 - ▶ Take 1 flop per element loaded
- ▶ Strive for peak bandwidth

Soumyajit Dey, Asst. Prof.

So, here we have examples of codes of how to do these reduction, but we will actually use them more for the assignment purpose. And for our discussions, we will focus more on a single block reduction and how that can be accelerated. So, before getting into that let us understand what are the matrixes for GPU performance which we will use, of course, this is something that we have seen earlier also, and this is just a recap.

So, ideally we will like to achieve more number of gigaflops per second; flops is floating point operations per second. This is basically about the compute the total I can do, for compute bound kernels, but, in this case, it is more of a memory bound kernel. So I will like to see that I am being able to use the full bandwidth of the memory, that is the full rate at which data can be read or written into the memory sub system, whether I have able to use that.

Now, in this case, if I am just doing a small reduction operation, it has low arithmetic intensity; as we have seen earlier. So it is more like, we are going to optimise for achieving the highest possible memory band width that I can get here.

(Refer Slide Time: 08:22)

Reduction 1: Interleaved Addressing

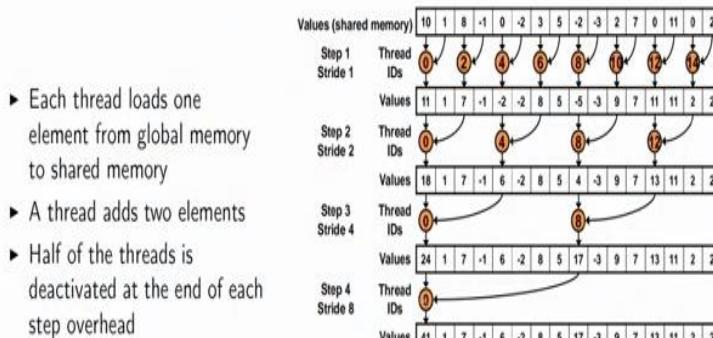


Figure: Reduction with Interleaved Addressing and Divergent Branch



So, the first possible technique, here, for reducing a block would be to use interleaved addressing. Now, this is some algorithm, which you have studied earlier also, it is just a brief recap. So, what we do is; we have this many data points to add and we have launched half the

number of threads. I mean, essentially there are threads and but we are not utilising those threads here.

There must have been threads which copied the data to the shared memory. By the way, I will just repeat these are optimisation when you are talking about it, we are doing it in the shared memory, assuming that threads have done a par thread data load into the shared memory. So, this is the first job, each thread loads one element from global memory to shared memory.

And then, I will actually engage half of the threads to perform the first level of addition, a thread adds 2 elements and half of the threads are actually deactivated after the initial load. Because they do not have anything to do, since one thread can add 2 elements. So, all the threads together collaboratively load data, then half of the threads perform the step 1 of addition. This addition is performed in a stride of 1, because you just use the threads; thread IDs corresponding memory index value and the next value.

And then, in the next iteration, you just increase the stride, the thread ID; the thread will add the location; its own locations value plus the data sitting at a stride of 2 and then stride of 4 and then stride of 8, in that way finally, the 0th thread will compute the final sum.

(Refer Slide Time: 10:13)

Reduction 1: Kernel

```
__global__ void reduce1(int *g_idata, int *g_odata, unsigned int n){  
    int *sdata = SharedMemory<int>();  
    // load shared mem  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;  
    sdata[tid] = (i < n) ? g_idata[i] : 0;  
    __syncthreads();  
    // do reduction in shared mem  
    for (unsigned int s=1; s < blockDim.x; s *= 2)  
    { // modulo arithmetic is slow!  
        if ((tid % (2*s)) == 0)  
            sdata[tid] += sdata[tid + s];  
        __syncthreads();  
    }  
    // write result for this block to global mem  
    if (tid == 0)  
        g_odata[blockIdx.x] = sdata[0];  
}
```



Soumali Das Arik

So, this is the reduction kernel, again I will repeat, we have seen it earlier but here we have to repeat it because we need to understand the starting point and what are its flaws. So, first thing you do is; you load the data into the shared memory, so this is the load operation, and then once that is done after a `__syncthread`, then all the loads are done; you start doing a reduction. So, this is a reduction loop, you start with the stride of 1 and in each iteration of the loop, you reduced by one level.

So, stride of 1 to the first level reduction; second level, you operate with the stride of 2. So essentially you operate on consecutive data points that is stride of 1, you operate on 2 stride valued data points, so that stride of 2, that is what we have here. So, s equal to 1, you do 1 level, then you multiply by 2, you go to stride of 2 and then you again do the addition for these values that are loaded in s data shared memory array and their index by the tid.

So, each step does the addition at a stride of 2, then in the next iteration, at a stride of 4 because you again multiply by 2. Now what are the good things and the bad things here; where you have parallelism, you have the threads loading the data in parallel and then you have the threads doing the addition in parallel although, half of the threads are not used. The bad thing is the modulo arithmetic done here is very slow.

So, for every thread, you are doing a thread ID percentile to s, to identify that whether this thread is the one which is supposed to do the job addition. Because as you can see with every level, half of the threads further go idle, whatever were the active threads in the earlier iteration, half of them would go in active.

So, in every iteration, you take the tid and do a percentile 2 to find out whether the tid will be active or not. If it is active, then you do an addition with the stride of value s and you keep on doing this until and unless, your this s is up to the half of the block dimension, that means, you have actually summed up all the values and you are at the last level. And then at the last level, you have the value available in s data 0 which is a final sum.

(Refer Slide Time: 12:53)

Reduction 1: Host

- ▶ The GPU kernel calculates data per block
- ▶ Partial sums computed by individual blocks
- ▶ Results will be stored in the first block elements of the global memory
- ▶ Final addition need to be done on this reduced data set
- ▶ By launching the same kernel again



And that's what is written back into the global memory. So this was our reduction kernel, for this kernel, you can write a host program, off course, it will be quite complicated in case, you are going to launch it again and again for reducing in different parts of the memory.

(Refer Slide Time: 13:06)

Reduction 1: Host Code for Multiple Kernel Launch

```
...//cudaMemcpyHostToDevice...
int threadsPerBlock = 64;
int old_blocks, blocks = (N / threadsPerBlock) / 2;
blocks = (blocks == 0) ? 1 : blocks;
old_blocks = blocks;
while (blocks > 0) // call compute kernel
{
    sum<<<blocks, threadsPerBlock>>>(devPtrA);
    old_blocks = blocks;
    blocks = (blocks / threadsPerBlock) / 2;
};
if (blocks == 0 && old_blocks != 1) // final kernel call, if still needed
    sum<<<1, old_blocks / 2>>>(devPtrA);
...//cudaMemcpyDeviceToHost...
```



(Refer Slide Time: 13:11)

Reduction 1: Analysis

Interleaved addressing with divergent branching

Problems:

- ▶ highly divergent
- ▶ warps are very inefficient
- ▶ half of the threads does nothing!
- ▶ % operator is very slow
- ▶ loop is expensive

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, these are host code, which were not discussing right now, we will actually provide some nice assignments using them and those will be useful and it will be described at that level. But try and understand the kernels execution here. So, this is the reduction kernel and as you can see, first problem is it is highly divergent, why because inside the loop, you have this percentile operation which the thread may or may not satisfy and the modulo arithmetic which is computing this is very slow.

Warps are very inefficient due to the reason that in every warp, you have divergence and anyway half of the threads do nothing after performing the global load and also the loop is expensive. So, some statistics here, considering an area of this size, if you perform this; if you actually execute this reduction kernel, multiple times using multiple kernel launches using a GPU tesla K40, this is the bandwidth and execution time that we can see here.

So, just to keep that this is not an execution of 1 kernel launch, we are actually following this picture of orchestrating multiple kernel launches and finally, getting the value, well inside each kernel launch, the code will execute is this one. In each launch, you actually reduce some part of the array. So, as we understand that there are significant numbers of problems in this reduction code, first of all it is highly divergent and warps are inefficient.

(Refer Slide Time: 14:52)

Reduction 2: Interleaved Addressing

- ▶ Replace divergent branch in inner loop
- ▶ With strided index and non-divergent branch
- ▶ New Problem: Shared Memory Bank Conflicts

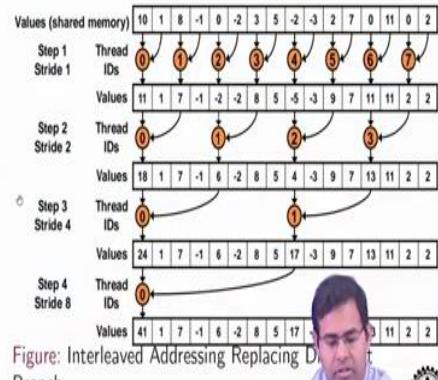


Figure: Interleaved Addressing Replacing Divergent Branch



The percent operation also is very slow and we will like to replace it. So, what can be the next possibility? The next possibility is that you do interleaved addressing, that means, first thing is I will like to remove the divergent branch from the inner loop and also I will like to remove this divergence but still, off course, I will use strider index that will help me to remove this percentile operation.

And also, the situation that in every execution of the warp, consecutive; half of the thread IDs, I mean, actually diverge. Now, how can we do that? If we look back into the pictures here, let us understand the divergence here. So, in the first iteration itself, we can see the divergence because these all started inside a single warp, inside a single warp at this point due to that percentile 2s, tid percent and 2s operation, the divergence came in.

Because only this thread progressed. So, essentially inside the warp, you are getting half of the work done. Now, this does not loop very bad here but consider large number of warps working for a big block of code for a big thread block, that would mean, inside each warp, you have half of the thread is not doing the addition at this level and again, half of the threads not doing anything here.

Try and understand there is a difference between the situation that you have half of the threads going inactive and half of the threads inside a warp going inactive. So, again I will just repeat;

there is a difference between half of the threads going inactive, while there is a property of the operation here but if half of the threads inside a warp going active, then your warps are not efficient, whatever operation the other threads are supposed to do, they are not synchronised inside the warp.

Alternative would have been that whichever were the active threads, whichever were the threads that are supposed to do the real addition, if I can pack them inside the same warp that would be more efficient because as you can see here, inside the warp, half of them are progressing, the reason is the thread IDs are what; 0, 2, 4, 6 like that, have the thread IDs of this thread itself being 0, 1, 2, 3 like that.

Then, I looking into a warp of size 32, I could have said that okay, all the threads are working, so originally there would have been a higher number of threads which have done the load but then, inside the warp, every thread is working, so try and understand the difference, half of the threads getting inactive at each level is one thing but still if the threads which are working on having consecutive ID that would mean that they are packed inside does not work.

And when the warps execute? I have fast progress, now that is what we achieve with this reduction 2 here, we will do some modification in the code to ensure that the although, half of the threads going inactive but the threads which are really working are of consecutive ID that would ensure they are executing inside a warp and they are all progressing in parallel because if you extend this picture, you have 32 threads which are inside a warp.

That would mean in one lock step, this step of the reduction would get done, which will not be the case if you extend this picture for the earlier reduction 1 kernel, you can understand this. Now, with the differences you can see between these 2 pictures is with respect to the thread IDs that, which thread is doing the job. So, here I still have half number of total threads doing the job at each level but the threads are all sequential in index, they form a common war.

I have removed the divergence, so whoever has the job to do, they will be doing the job maximally, there is no idle slot; that is not the case that a warp is divergent and some of the

threads are not working, at least in the initial parts of the reduction. Now, that would significantly speed up the execution of the code, when I launch the kernel multiple times and reduce in different parts of the program.

So, how to do this? So, essentially what I want is still my stride is going to increase from 1, 2, 4 like that but the thread IDs is going to be like this. So, while doing the reduction in the shared memory, this is just a snapshot of the reduction 1 kernel, these were the problems. The first problem was this tid percentile operation and that was slow and that was also getting the divergence.

(Refer Slide Time: 19:49)

Reduction 2: Kernel

```
__global__ void reduce2(int *g_idata, int *g_odata, unsigned int n){  
    int *sdata = SharedMemory<int>();  
    // load shared mem  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;  
    sdata[tid] = (i < n) ? g_idata[i] : 0;  
    __syncthreads();  
    // do reduction in shared mem  
    for (unsigned int s=1; s < blockDim.x; s *= 2){  
        int index = 2 * s * tid;  
        if (index < blockDim.x)  
            sdata[index] += sdata[index + s];  
        __syncthreads();  
    }  
    // write result for this block to global mem  
    if (tid == 0)  
        g_odata[blockIdx.x] = sdata[0];  
}
```

Soumyajit Dey, A

But now, you do as the different thing. So, I have a more complex access expression, earlier my access expression was identity map, so every tid will map its corresponding index data in s data. But now, what I do is; for every tid, I multiply it with s and 2 to get the index where it is suppose, to warp. So, initially s is 1 that means, if the tid is 0, it is working on the 0th location, if the tid is 1, it is working on the second location, if the tid is 2, it is working on the 4th level so on and so forth.

So, that's what we want here tid1; these are the thread IDs, it is working on location 2, tid 2 working on location 4, tid 3 working on location 6 like that and that is achieved by this modified access expression. How long will this work? As long as the index is less than the block

dimension and off course, the stride remains the same. The striding mechanism remains the same from 1 to multiplied by 2, that is a hop; in every hop you multiply by 2, you start from s equal to 1.

But you change the access expression from my simple identity map to $2s$ multiplied by tid and that gives you this nice access pattern of threads. So, with this modification, you have warps getting utilised, lack of divergence, so you remove the divergence of the threads and you remove the complex percentile 2 operation but this also will suffer from the shared memory bank conflict issue which we will see soon.

(Refer Slide Time: 21:34)

Reduction 2: Analysis

Interleaved addressing with divergent branching

Problems:

- ▶ highly divergent
- ▶ warps are very inefficient
- ▶ % operator is very slow
- ▶ half of the threads does nothing!
- ▶ loop is expensive
- ▶ shared memory bank conflicts

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction Unit	Time Second	Bandwidth GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117



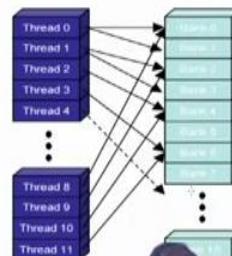
Soumyajit Dey, Asst Prof

First, let us understand why we will have the shared memory bank conflict. So as we can see that we will have a shared memory bank conflict here, now if you look into this picture, every thread is accessing 2 consecutive locations, the next thread is again accessing 2 consequent locations so on and so forth.

(Refer Slide Time: 21:56)

Shared Memory Bank Conflict

- ▶ Shared Memory is divided into banks and each bank has serial read/write access
- ▶ If more than one thread attempts to access same bank at same time, the accesses are serialized (Bank Conflict)
- ▶ The hardware splits a memory request decreasing the effective bandwidth



Soumyajit Dey, Asst. Prof.

So, with this, if you look at the way the threads access the shared memory, thread 0 access bank 0, bank 1, thread 1 access bank 1 and bank 2, like this. If I draw the picture for a collection of threads, I would see that I will have inside a warp, I will have 2 threads accessing the same bank in parallel. Since, every thread access as parallel banks, the number of threads in a warp is equal to the number of banks.

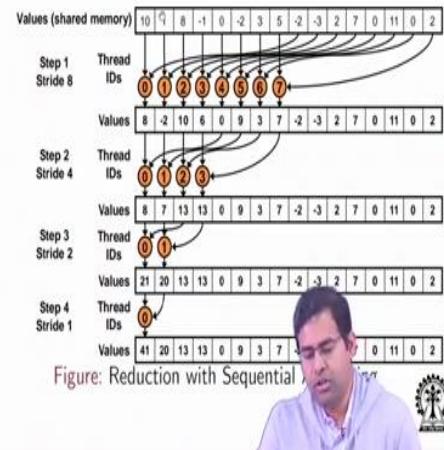
So, I have a 2 way conflict for every thread, the accesses are not uniform across banks, every thread is accessing 2 consecutive banks so, I have a bank conflict here. So, if more than 1 thread attempts to access the same bank, we know, we get the bank conflict and that needs to be resolved now. The way that you can resolve it is, you have to again perform some transformation into the code.

Now, again we will try to see, every thread, this access is doing, they are accessing into consecutive banks.

(Refer Slide Time: 23:11)

Reduction 3: Sequential Addressing

- ▶ Replace strided indexing in inner loop
- ▶ With reversed loop and threadID-based indexing
- ▶ New Problem: Idle Threads on first loop iteration



So, how can this be modified? It can be modified by doing a sequential addressing. You do not go from stride value equal to 1 and keep on increasing the stride but rather than that, you keep on decreasing the stride. Now, how does this help? Because, when the threads are going to perform the loads, the loads from the shared memory you will be able to remove the conflicts.

So, this is again something I think before getting into this, you should go back and read once the literature that we studied about shared memory. At this point we will be stopping today's lecture and we will resume in the next lecture from this reduction 3 and so, just to summarise in the 2 reductions that we have discussed. The first was the original reduction where, let me just through the picture, we had half of the threads working.

But the way the thread IDs are being arranged inside every warp, I have half of the threads working that was a bad thing, in order to remove that, what we did was, we came up with nice and intelligent access expressions, so that inside every warp, I have all the consecutive thread ID is working that gave me some speed up but as we can see if I do a memory access analysis, there is an issue of bank conflict here.

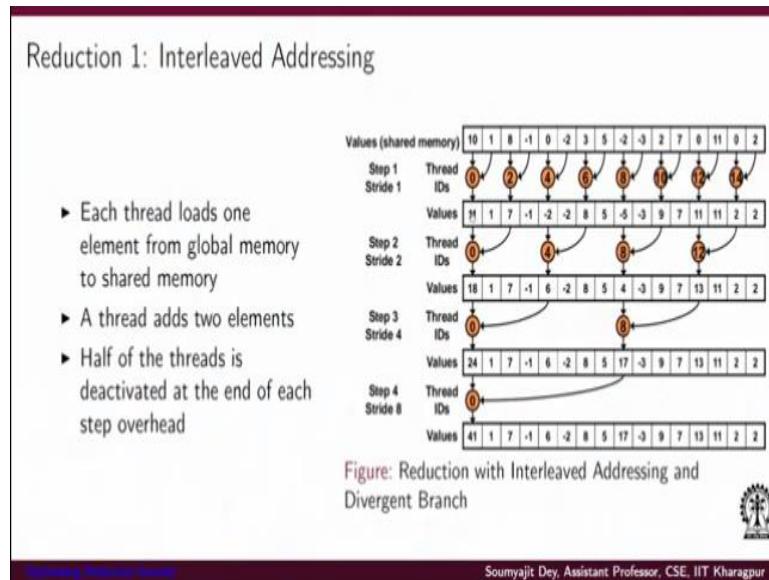
And we need to remove that, we will see how that can be removed by doing further modifications to the expressions in the program and with this, we would like to end our lecture here, thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology - Kharagpur

Lecture – 29
Optimising Reduction Kernels (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming. Just as a small recall, in the last lecture we have been discussing possible optimisation techniques to write a code for parallel reduction in GPU architecture.

(Refer Slide Time: 00:45)



So, we just picked up the problem and discussed a few optimisation techniques. So a small recap on the techniques we already discussed, the first was the very nice parallel algorithm where we just used threads for doing the local computation. The first thing is that you just use the threads to load the data from the global to the shared memory and then each thread is doing a sum of consecutive elements and in that way, what you are essentially doing is you are using thread IDs with alternate values.

I mean, their thread Id is differing by 1 here, to add consecutive values and you go to the next iteration. What is happening in each iteration? I mean half of the threads are going idle and also in parallel, the threads are doing the addition in strides which are increasing by a factor of 2.

(Refer Slide Time: 01:52)

Reduction 1: Analysis

Interleaved addressing with divergent branching

Problems:

- ▶ highly divergent
- ▶ warps are very inefficient
- ▶ half of the threads does nothing!
- ▶ % operator is very slow
- ▶ loop is expensive

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And the problems, that we had with this implementation was as follows like, first of all, it was a highly divergent implementation.

(Refer Slide Time: 01:57)

Reduction 1: Kernel

```
__global__ void reduce1(int *g_idata, int *g_odata, unsigned int n){  
    int *sdata = SharedMemory<int>();  
    // load shared mem  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;  
    sdata[tid] = (i < n) ? g_idata[i] : 0;  
    __syncthreads();  
    // do reduction in shared mem  
    for (unsigned int s=1; s < blockDim.x; s *= 2)  
    { // modulo arithmetic is slow!  
        if ((tid % (2*s)) == 0)  
            sdata[tid] += sdata[tid + s];  
        __syncthreads();  
    }  
    // write result for this block to global mem  
    if (tid == 0)  
        g_odata[blockIdx.x] = sdata[0];  
}
```



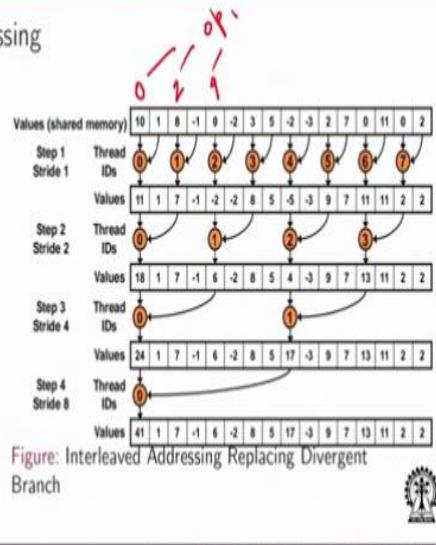
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

As you can see due to this if condition, half of your threads inside the same warp is getting; I mean, getting to diverge here at this point and also you have a loop over it and you are doing percentile computation that also has got sum over it.

(Refer Slide Time: 02:27)

Reduction 2: Interleaved Addressing

- ▶ Replace divergent branch in inner loop
- ▶ With strided index and non-divergent branch
- ▶ New Problem: Shared Memory Bank Conflicts



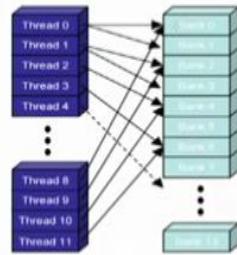
And, these were the most important problems here. The first thing we do as part of optimisation 2; in this case of reduction 2, I would say the first optimisation here is that you make consecutive threads to the addition. So, if you remember the earlier picture, the thread IDs were 0, 2, 4, like that but we do some program transformation and make consecutive threads take care of doing the addition.

So, we have the other features constant that means the strides keep on increasing in multiples of 2 but you are working with consecutive threads. Due to this, you always have warps which are non-divergent because all consecutive threads are always active most of the time but the problem here is as we can see that you have a situation of shared memory bank conflict which we can look into by studying the access pattern of the threads.

(Refer Slide Time: 03:25)

Shared Memory Bank Conflict

- ▶ Shared Memory is divided into banks and each bank has serial read/write access
- ▶ If more than one thread attempts to access same bank at same time, the accesses are serialized (Bank Conflict)
- ▶ The hardware splits a memory request decreasing the effective bandwidth



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

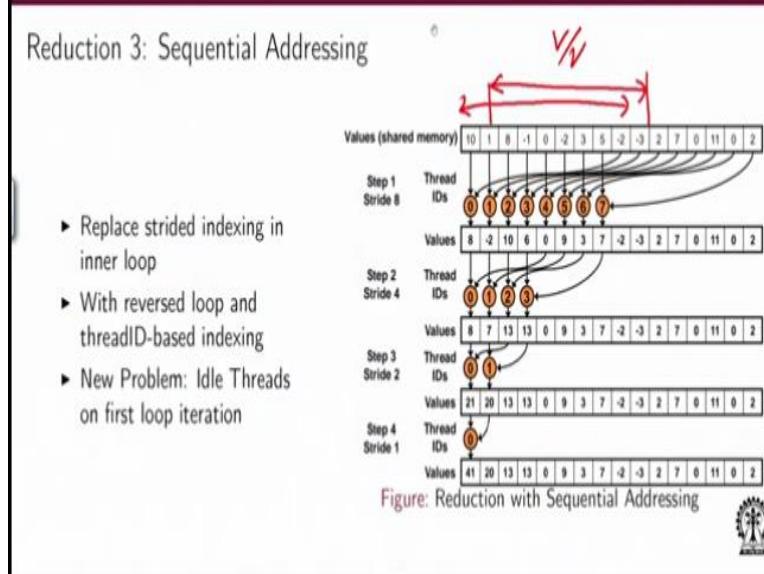
So, every thread is accessing two consecutive values and this will give rise to a 2-way bank conflict for each thread here. In this example the solution here would be that, okay, let us now change the way in which the threads access the corresponding values. So, if you just take a close look here, we made consecutive threads warp but consecutive threads are working on 2 consecutive values from the shared memory.

So, when the operation is going to happen, for each thread first there would be a load, I mean for operand 1, load for operand 2 and then the addition. Now, here, since if I look at the access pattern for the threads, all the threads that we are going to do for operand 1, they are not consecutive, that is the basic problem. So, thread 0 is loading from here, thread 1 is loading from here, thread 2 is loading from here, as you can see.

So, to be more specific, thread 0 is loading from location 0, thread 1 is loading from location 2, thread 2 is loading from the location 4 and all that for operand 1. So that is the instruction that will fire and when this instruction fires, all the threads be in the warp, since they are not accessing exactly consecutive locations. So we have got this issue of the bank conflict here. But how do we remove that?

Now, let us look at the access pattern of a transform thread for a transformed operation here. Let us look at the alternate code that we can have to resolve the bank conflict.

(Refer Slide Time: 05:23)



So, now let us start saying that, okay, let us arrange the threads in such a way that I have consecutive threads working, i.e. removing the issue of divergent inside the warp, also we change the way the consecutive thread IDs access the memory. So, what we do is; earlier thread Id is 0, was accessing memory location 0 and memory location 1 but now, let us make thread Id 0, access memory location 0.

And the next element that is going to access is sitting at an offset which is of size $L/2$, where L is the total size. And the same is going to hold for all the consecutive threads. For all of them, their accessing 2 operands were sitting at a large offset here rather than accessing consecutive operands. So, what is the good thing? Now, see that we are trying to do a load followed by add operation.

So, again we will just repeat, if I am looking at the SIMD instruction that will fire; all the threads will first load operand1, the thread will load operand 2, the threads will add. So, now if I look at the way the threads are going to load operand1, I have that all the threads are accessing consecutive shared memory banks, all the threads in a warp or accessing consecutive shared memory banks constitutional and that resolves the bank conflict issue which was earlier.

So just for clarity, let me just repeat again, what happened here in the earlier scenario. Here, when all the threads were consecutive ID from the same warp, no divergence but they were loading operand1, they were accessing locations with 1 hop and that would effectively lead to the 2-way bank conflict problem which would occur for both of the loads individually but now, when I write a code, which is going to with me this kind of an access pattern.

So, I have consecutive thread IDs inside the warp and they are all loading the operand1 from the shared memory without any bank conflict because all the threads are accessing all the banks in parallel, that is the good thing. So, this load will have no bank conflict, the second load where it is going operand 2, that also is not going to have any bank conflict because again, they are all accessing consecutive banks.

So, I have removed the bank conflicts of both the loads and then I have the unusual addition. But now the question is; how do I write the code, so that I can make it behave like this?

(Refer Slide Time: 08:20)

Reduction 2: Kernel

```
--global__ void reduce2(int *g_idata, int *g_odata, unsigned int n){
    int *sdata = SharedMemory<int>();
    // load shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();
    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2){
        int index = 2 * s * tid;
        if (index < blockDim.x)
            sdata[index] += sdata[index + s];
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];
}
```



Engineering Mathematics

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, observe what was the earlier code; the earlier code started with a stride of 1 that is why the threads were loading consecutive IDs in the first iteration, then it was loading with a stride of 2, in the second iteration then stride of 4, in the third iteration so on and so forth. So, this was the kind of code that we are executing. But now, if I want this kind of an access pattern that means, I

want all the operand one of consecutive threads to be also consecutive or operand 2 are consecutive thread to be also consecutive.

Then, I need each of the threads to start with the highest stride and then in the next iteration, the stride should reduce. So, here this is like; the stride is half of the total array size and then it should reduce by half and then it should further reduce by half like that, which would meant, I do not count from the lower value of stride to the higher value in multiples of 2 rather I start the loop with a highest value of the stride and get down, each time dividing the strides factor by 2.

(Refer Slide Time: 09:45)

Reduction 3: Kernel

```
__global__ void reduce3(int *g_idata, int *g_odata, unsigned int n){  
    int *sdata = SharedMemory<int>();  
    // load shared mem  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;  
    sdata[tid] = (i < n) ? g_idata[i] : 0;  
    __syncthreads();  
    // do reduction in shared mem  
    for (unsigned int s=blockDim.x/2; s>0; s>=1)  
    {  
        if (tid < s)  
            sdata[tid] += sdata[tid + s];  
        __syncthreads();  
    }  
    // write result for this block to global mem  
    if (tid == 0)  
        g_odata[blockIdx.x] = sdata[0];  
}
```



Soumavir Das, Assistant Professor, CSE, IIT Kharagpur

So, if we continue like that, then this is what we get, which was the original reduction code. Now I start with the highest stride which is half of the block dimension and in each iteration, I reduced it by 2, inside all we are doing is, we are checking whether that the ID is less than the stride because, off course, I have started with the highest stride value, so every tid that is going to really get into the loop will be less than the stride value as you can see that, here the stride value is this much.

So, the tid's that are going to get into the loop, should be up to this. In the next iteration, the stride value is this, the tid is that should be active should be 1 less than this, so on so forth. So, in this way we keep on continuing and the loop will keep on executing, decreasing the stride value.

And inside each iteration, I have got, I mean, all the threads getting half in number but I also have in each iteration, the threads who are consecutively idle in a warp, they do not diverge.

And they also do not get into any shared memory bank conflict, I hope this is clear without example, yeah, earlier was reduction 2, this is reduction 3. So, I have, yes, that is the stride value starting from the highest stride keeps on decreasing and inside the loop, every thread is doing the usual sum; and then again in the loop, you decrease the stride and again, getting the loop, you count the sum with the stride decreased and you keep on doing this.

(Refer Slide Time: 11:25)

Reduction 3: Analysis

Interleaved addressing with divergent branching

Problems:

- ▶ loop is expensive
- ▶ shared memory bank conflicts
- ▶ Half of the threads are idle on first loop iteration

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839



Now, let us look an analysis of this code; as you can see that there is gradual reduction in execution time. Here we have given some statistics considering an array size of 2 to the power 26, thread blocks of size 1024 which means, I am calling the kernel multiple times and doing the reduction, using my reduction kernel, 1, 2 or 3 whichever in each iteration, it is reducing the set of blocks and with requisite call of the kernel, finally I get to 1 value.

So, the reduction code when it reduces this entire array for tesla K40 GPU as you can see these are the timings and these are the bandwidths, the timings tell me the total execution time considering these initial array size, so off course, you can understand the timings with respect to multiple kernel invocations, the number of kernel invocations are required for reducing the entire

array and this bandwidth gives me the total number of loads and stores that have happened divided by the amount of time spent.

So, what is the GB per second bandwidth consumed from the memory? So as we can see with reduction of conflicts for the shared memory, I have higher bandwidth and also I have a reduced execution time as an effect of that and primarily, all the acceleration that is coming in is due to the increased memory band width exploited by the code and that actually is effecting the timing and reducing the timing here.

I hope this is clear, just to repeat again, the reduced timing that I am getting is basically due to the increase in memory band width that the program is able to exploit. Now, still I would say that we have certain issues, what is that? First of all, this loop is still very expensive again, why do we call a loop as expensive; just to we clear from our earlier discussion on basic architecture, every time your threads get into the loop, you have to check for the loop conditions.

So, that is one branching check that you have to do before the threads have getting inside the loop, the good thing here is we have removed the module operation and also the most important thing is half of the threads are idle on loop; on first loop iteration, that means initially, after loading all the data from the global memory to the shared memory in every launch of the kernel, only half of the threads get inside the actual loop.

So, this is the code that every loop is executing. This is a line of code that every loop executes, apart from just computing the global ID using the block dimension, block ID and thread IDs and here, as you can see that, we are just doing the load from the global memory to the shared memory, after that immediately, I am starting to use half of the active thread that I have inside the loop.

(Refer Slide Time: 14:34)

Reduction 4: First Add During Load

- ▶ Make busy all threads in the first step
- ▶ Halve the number of blocks
- ▶ Replace single load with two loads
- ▶ Allocation process performs the first reduction

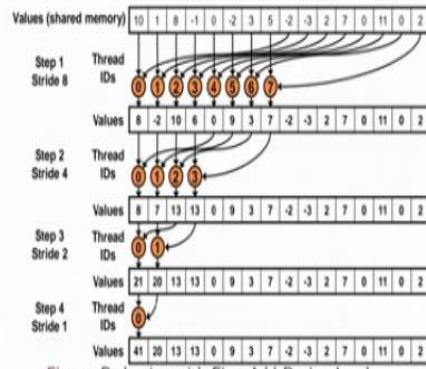


Figure: Reduction with First Add During Load

As an alternative, just to increase the par thread activity, what I can do is; I can increase the amount of computation that every thread has to do in the first iteration, as you can see in the first iteration up to reduction 3 for method; in all the methods; reduction 1, reduction 2, reduction 3, there is not much activity to be done by each of the threads apart from the loads, after one half of the threads.

All they are doing is just the loading the shared memory and after that half of the threads do not have any activity, instead of that what you suggest is; we give more warp for each of the threads by making them do some more additions. So, make busy all threads in the first half and half the number of blocks. What do we mean is; let all threads not only load data from global memory to the shared memory. But all threads also do some reduction by themselves for example, okay, maybe I make each thread load 2 data from the global memory and then add them and then stored into the shared memory that is increasing the par stride activity, as a side effect that is going to half the number of blocks also. So, I increase the par thread activity, I am as the side effect, decreasing the number of blocks.

So, essentially I am replacing one single load with 2 loads and in the addition for all the thread. So this is the first step of the reduction which we ensure that it is going to be done by all the threads together. So, I mean from mathematical point of view, what's happening is you have too

many threads that you have launched, you have launched too many threads here and then you are certainly increasing, I mean you are using only half of them.

So, if I am trying to draw a picture representation here. Let us say launching these many threads, all of them just do the global load and then half of them keep on working. So, if I plot par thread activity, this is how things are really happening. Now with this step what are we essentially doing? So, then okay, let me keep the original picture here, so with this step, we are making it like this, because I have increased the par thread activity here.

I am launching less number of threads and this threads are only taking care of this part of the activity as well as the other activities, so I am launching this many number of threads and they are taking care of the activities which earlier this half of the threads were doing. Now I do not launch them, an increase the overall par thread activity by launching half number of threads and make them do more of initial work here.

So, that is the important point, we are trying to make that if we can find a nice balance between par thread activity and the number of threads that is a good way to look into a issue of parallelism. So, just to explain further, I have got some activity to be executed and I have launched too many threads and I am not executing all the threads throughout the lifetime of the activity, so that does not make sense.

Rather than that I should try to do a load balancing here. I increase the amount of work to be done per thread and decrease the number of thread, then since and I am using less number of threads and utilising them more, I will have less execution time. Because I also put less overhead on the GPU schedule and other resources in terms of context saving and doing the scheduling job of a higher number of threads.

So, the good thing is I do not use too many threads which are idle, for most of the time, rather in that way I save time in saving context and scheduling.

(Refer Slide Time: 19:41)

Reduction 4: Kernel

```
__global__ void reduce4(int *g_idata, int *g_odata, unsigned int n){  
    int *sdata = SharedMemory<int>();  
    // reading from global memory, writing to shared memory  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;  
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];  
    __syncthreads();  
    // do reduction in shared mem  
    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {  
        if (tid < s)  
            sdata[tid] += sdata[tid + s];  
        __syncthreads();  
    }  
    // write result for this block to global mem  
    if (tid == 0)  
        g_odata[blockIdx.x] = sdata[0];  
}
```



Source: NVIDIA GPU

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, then if this was my original reduction 3 kernel, just have a look, this is the part of code, which is now going to change and it instead will become just like an earlier, it was just, you load 1 global data point and store into 1 shared data point. Now, each thread is loading 2 global data points and they are storing into the shared memory, that is all happening.

(Refer Slide Time: 20:07)

Reduction 4: Analysis

Memory bandwidth is still underutilized

Problems:

- ▶ Half of the threads are idle on first loop iteration
- ▶ loop overhead
- ▶ Another likely bottleneck is instruction overhead

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, the analysis would be like that I have just increased the par thread activity due to the, I have a tremendous increase in effective bandwidth usage and other result, further reduction in the execution time. So, I do not have this problem of half of the threads going idle on the first loop iteration. But I still have this issue of the loop over it, because every time I launched threads

eventually, getting to the loop and they have to in every iteration of the loop, they have to check the loop conditions and all that.

And now, the another likely bottle neck can be the instruction overhead because now, I am executing more instructions per thread but I can see that is going to be a nice balance here.

(Refer Slide Time: 20:50)

Reduction 5: Unrolling the Last Warp

- ▶ Number of active threads decreases with the number of iteration
- ▶ When $s \leq 32$, only one warp is left
- ▶ Warp runs the same instruction (SIMD)
- ▶ That means when $s \leq 32$:
 - ▶ "`__syncthreads()`" is not needed
 - ▶ "`"if (tid < s)"` is not needed
- ▶ Unroll last 6 iterations

Without unrolling, all warps execute every iteration of the for loop and if statement



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, the 5th reduction version is about the observation that I do not need this kind of synchronisation barrier to be put in, when I am executing the last warp that means, when the number of active threads reduce to 32, then what happens; as we can see that we already removed the divergence from the reduction 1 that means, the threads that are active they are now with consecutive IDs.

So, they actually form a warp, since these threads are themselves forming a warp and we know that warps execute instructions in lockstep from the programming point of view and that would actually make these `__syncthread` operations redundant here. So this is not needed and moreover, this check will also not be needed that whether the tid is less than s is also not needed.

Because whatever the threads are there, I mean, which are going to warp, the other threads are really not going to do any effective warp. So, because I am final interested with only the thread ID 0, which will give me the sum, so at this point, I do not need any of them. So, the good thing I

can do is; I can remove the same thread and I can unroll the last 6 iterations; why last 6 iterations?

Because from 32, then 16, then 8, then 4 and like that finally, with 1 so, this last 6 iterations, I can also save into the loop condition checking. So the good thing would be that I simply remove the loop condition and all if conditions, when the number of threads on which I required to focus changes to 32 and they all come inside a warp. So, then I will simply unroll this loop and write similar statements consecutively.

And then I do not need this loop at all for those last 6 iterations, I do not need to check this loop condition, I do not need to do this if-else check. So without the unrolling, all warps execute every iteration of the for loop and the if statement and this is what we can just remove by doing this unrolling operation. So, just recalling what we had in loop 4; this reduction 4 was that we were every time in the loop iteration was checking this condition.

(Refer Slide Time: 23:38)

Reduction 5: Kernel

```
__global__ void reduce5(int *g_idata, int *g_odata, unsigned int n){  
    int *sdata = SharedMemory<int>();  
    // reading from global memory, writing to shared memory  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;  
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];  
    __syncthreads();  
    // do reduction in shared mem  
    for (unsigned int s=blockDim.x/2; s>32; s>>=1) {  
        if (tid < s)  
            sdata[tid] += sdata[tid + s];  
        __syncthreads();  
    }  
    if (tid < 32)  
        warpReduce(sdata, tid);  
    // write result for this block to global mem  
    if (tid == 0)  
        g_odata[blockIdx.x] = sdata[0];  
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur



And now, we want to remove it, so the alternate thing that we will do is; earlier this loop was executing up to stride greater than 0, now I make every warp go through this program for loop, as long as the stride is greater than 32 but once the stride variable decreases to less than 32, okay, once this stride variable decreases to less than 32 for the threads, then I do not let them get into

the loop at all and then I just make a call to this warp reduce function for all threads with the tid less than 32 and just forget the other threads here.

So, as you can see this code will be executed as long as s is greater than equal to 32, once s is less than 32, I completely skip this loop. So, now the s is less than 32 and at that point, I just check for the last warp because that is what I'm interested in so, for the last warp, I make a call here to this warp reduce function. I hope this is clear so, earlier what was happening is all threads were executing this loop.

We have identified that once I am only at the last few stages of the reduction where I have only 32 consecutive values to warp with and that would mean 32 consecutive threads to warp with, sorry, 64 consecutive values and 32 consecutive threads to warp with, I do not need to getting to the loop anymore because I will just write those statements in line one after another, avoid the loop condition and more importantly, I can avoid the `__syncthread` here.

So, I execute this loop only up to s greater than 32 after that I am, practically, only interested in the tid's which are less than 32, because with s equal to 32, what will be interesting is; the first 32 threads working on the 64 consecutive data points which will be handled by this warp reduce function.

(Refer Slide Time: 25:48)

Reduction 5: warpReduce

```
_device__ void warpReduce(int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```

And what it will do is; it will just as you can see, I have just copied this body of loop in line one after another with decreasing value of s, so manually I am doing this without executing the loop here, I have completely hard coded 32, 16, 8, 4, 2 and 1, so here after this point, I do not need, I am just repeating this part again, after this point when s is; I have executed the loop with s equal to 64.

After that point, I am only interested in the first warp, I leave all the other warps, I allocate the first warp to execute this sequence of statements, and as you can understand clearly now that this will be the thing that I am only interested in. I do not care about the other warps now and so, when I execute this sequence of statements, the advantage I gain is; I do not have to execute this `_syncthread`, I do not have to execute this for loops condition check and also the if condition check.

So, that would speed up the execution here and then and I just use the final thread to do the final sum, I mean to report the final sum from this `sdata[0]` to `g_data`, the output data point in the global memory.

(Refer Slide Time: 27:09)

Reduction 5: Analysis																							
Array Size:	2^{26}																						
Threads/Block:	1024																						
GPU used:	Tesla K40m																						
Problems:																							
<ul style="list-style-type: none"> ▶ Still have iterations ▶ loop overhead 																							
<table border="1"> <thead> <tr> <th>Reduction</th><th>Time</th><th>Bandwidth</th></tr> <tr> <th>Unit</th><th>Second</th><th>GB/Second</th></tr> </thead> <tbody> <tr> <td>Reduce 1</td><td>0.03276</td><td>8.1951</td></tr> <tr> <td>Reduce 2</td><td>0.02312</td><td>11.6117</td></tr> <tr> <td>Reduce 3</td><td>0.01939</td><td>13.839</td></tr> <tr> <td>Reduce 4</td><td>0.01104</td><td>24.3098</td></tr> <tr> <td>Reduce 5</td><td>0.00836</td><td>32.1053</td></tr> </tbody> </table>			Reduction	Time	Bandwidth	Unit	Second	GB/Second	Reduce 1	0.03276	8.1951	Reduce 2	0.02312	11.6117	Reduce 3	0.01939	13.839	Reduce 4	0.01104	24.3098	Reduce 5	0.00836	32.1053
Reduction	Time	Bandwidth																					
Unit	Second	GB/Second																					
Reduce 1	0.03276	8.1951																					
Reduce 2	0.02312	11.6117																					
Reduce 3	0.01939	13.839																					
Reduce 4	0.01104	24.3098																					
Reduce 5	0.00836	32.1053																					

So, if I do an analysis here, then as you can see that this is further doing an order of reduction in the timing by unrolling the last warp, so this has been a quite a successful optimisation again and this was also the result of increasing the band width further, I mean, effective increase in the

band width that we are getting for the same code, same array and same thread block choice but still I have iterations and loop over it.

Because, although, this has made the last warp go find but still for the previous warps, I have this issue of loop overhead and also I have this multiple iterations to execute.

(Refer Slide Time: 27:52)

Reduction 6: Complete Unrolling

If number of iterations is known at compile time, could completely unroll the reduction.

- ▶ Block size is limited to 512 or 1024 threads
- ▶ Block size should be of power-of-2
- ▶ For a fixed block size, complete unrolling is easy
- ▶ For generic implementation, solution is-
 - ▶ CUDA supports C++ template parameters on device and host functions
 - ▶ Block size can be specified as a function template parameter



So, what can be an alternative here; the alternative here is that if I can completely unroll the execution, I can remove the loop completely but the issue is why do I really need the loop; I need the loop that because at the compile time, the number of iterations required is unknown. Because I do not know the block size, the number of threads on which reduction kernel is going to be applied.

Now, if I take some static decisions that let us say, I limit the block size to 512 or 1024 threads; because anyway, it is going to be power of 2. So, if I know the block size, then doing a complete unrolling of the code is going to be easy. So, in this case what we can do is; we can take an advantage of C++ templates and that is something that CUDA is going to support. So, without getting into too much detail of what is a template and all that, we will just go through the example.

Off course, this is something advance, so we will just give you an example here, so I mean you can consider this addendum to our original syllabus material, I mean this is just for the people who are more interested into the fundamentals of I mean, programming and further optimisations here, so instead of passing the block size as, I mean, making an implementation inside which I am handling a generic block size.

Let us assume that the block size can be specified as the function parameter and or more I would say, is like the template parameter.

(Refer Slide Time: 29:29)

Reduction 6: Kernel

Specify block size as a function template parameter

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n){
    int *sdata = SharedMemory<int>();
    // reading from global memory, writing to shared memory
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
    __syncthreads();

    // do reduction in shared memory
    if (blockSize >= 512) {
        if (tid < 256)
            sdata[tid] += sdata[tid + 256];
        __syncthreads();
    }
}
```



What is the good thing then? So, instead of having just the reduction kernel, I make a template of it with the parameterised block size. So, what is the advantage? You just tell, what is the advantage of function template here? So if I have a template declaration here with parameterised block size followed by the usual reduce kernel, then the system will make multiple implementations of this function but different possible block sizes here.

Actually, I mean, I am adding more dynamism here, so essentially as you can see that if I decide that, okay, the block size is limited by the size 512 or 1024 threads, then here, since I am creating a function template here, so what I will do is; I will now do the reduction entirely through sequence of if-else blocks, completely remove the loop, so this part is usual, you can just consider this as an computation of the global ID.

And then, okay, I think we have some, we can remove this, yeah, so you are computing these here the s data thread ID locations and then you do the reduction in the shared memory using a sequence of if-else blocks, so you check what is the block size and based on that you get into the corresponding part, where you are starting to do the reduction. So, if the block size is greater than 512 billion, then you will execute this block, with threads that are of ID up to 256, I mean less than 256.

Because then you have a block of size 512 and you are going to use half of the threads to do the iteration operations, like that so on so forth.

(Refer Slide Time: 31:55)

Reduction 6: Kernel

```
if (blockSize >= 256) {
    if (tid < 128)
        sdata[tid] += sdata[tid + 128];
    __syncthreads();
}
if (blockSize >= 128) {
    if (tid < 64)
        sdata[tid] += sdata[tid + 64];
    __syncthreads();
}
if (tid < 32)
    varpReduce<blockSize>(sdata, tid);

// write result for this block to global mem
if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```



So, that I show it is going to continue, so essentially we are just unroll the loop and we have just written a sequence of if-else statements, where the statement block will get activated depending on the block size so essentially, the block size is being provided as a template parameter here, so that you have different versions of the code and you have based on the block size, the code will actually execute whichever is applicable.

So, suppose your block size is 512 that would mean, your code will start executing and will execute this entire sequence of if blocks, because that is what the loop will do; the equivalent loop will do. So, essentially with 512, you have how many in each iteration? You have got the;

so essentially, when to compute with strides which is starting with 256 because block size 512, stride would be 256 and with that we start the reduction.

Just that code as you can see in every step, the reduction; reduction stride size will decrease, all we have done here is we have simply replicated their behaviour without the loop, so we just put in the if-else blocks, so each block size is greater than 512, I start executing from this block. If the block size is not greater than 512 but greater than 256, greater than or equal to, then I just start writing it from this block.

And why do I have this reduction by half because of the nature of the code, we know that the block size will be multiples of 2 and in every iteration, it will be parts of 2 and in every iteration, you are going to divide the stride by half. So just have a closed loop into our earlier reduction kernels, look at the for loops and mentally, unroll the for loop with the reducing value of strides and starting from the 5th block size and this is the execution sequence which was suppose to get.

All we are doing is; we are removing the loop, so that you are removing the loop chip and you are just executing the internal behaviour of the loop here, because as you can see everything else is same, at the end you again reduce the last warp but again you are using function template parameter here.

So, as I am repeating here for our course purpose, we do not assume that you will have knowledge of C++ or function templates, considered this as just some extra interesting piece of code, where we show that how using this function templates, you can make good use of; I mean, you can actually make good use of this kind of templates to do further optimisation with respect to reductions.

(Refer Slide Time: 34:52)

Reduction 6: Kernel

Modified warpReduce function:

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid)
{
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, this is the corresponding way when you are going to write a warp reduce function as in a template form. It is just the same but all we are doing is you are essentially creating different versions of the function for different possible block sizes.

(Refer Slide Time: 35:13)

Reduction 6: Analysis

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

- Algorithm Cascading can lead to significant speedups in practice

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6111
Reduce 3	0.01939	13.8391
Reduce 4	0.01104	24.3001
Reduce 5	0.00836	27.1111
Reduce 6	0.00769	28.3333

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if you do analysis here, as you can see that, since this approach is able to remove the loop and just unroll the execution of the loop as the sequence of if-else blocks, the sequence of if blocks, it leads to further reduction in the execution over it. So, I hope this is a bit clear to you, so just remember the point to be taken in this case here is the issue with unrolling is; I do not know what is my start size is.

So, instead of writing a function, I write a function template where I give the start size is the parameter, so this can further be optimised with significant speedups in practice.

(Refer Slide Time: 36:04)

Reduction 7: Multiple Adds / Thread

Algorithm Cascading:

- ▶ Combine sequential and parallel reduction
 - ▶ Each thread loads and sums multiple elements into shared memory
 - ▶ Tree-based reduction in shared memory
- ▶ Replace load and add two elements
- ▶ With a loop to add as many as necessary



And the approach would be known as algorithmic cascading. Let us try and understand what is this issue of algorithmic cascading? Now earlier, what we saw was that to increase the per thread activity, we just included some activity for all the threads initially, so we did some addition on the global memory operands before doing; before storing them in the shared memory and then proceeding with half of the threads but that part is also configurable.

So, I can actually do a combination of sequential and parallel reduction, so each thread instead of loading and summing just 2 elements, it can do more activity, it can load and sum a sequence of elements and then it can follow the idea of tree based reduction that we showed in shared memory. So, all we are trying to say here is instead of just doing 1 add and then storing the data in the shared memory, so this as you can see that this as a per thread activity.

If we do not do this, then what do we really have; we have half of the threads working with the addition and only initially, all the threads were active in doing the load, with these I increase per thread activity by doing one at globally on the data, so this is the sequential addition performed by each of the threads on the data global; on the global data before storing it into the shared memory.

(Refer Slide Time: 37:34)

Reduction 7: Kernel

```
__global__ void reduce7(int *g_idata, int *g_odata, unsigned int n)
{
    int *sdata = SharedMemory<int>();
    // reading from global memory, writing to shared memory
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    sdata[tid] = 0;
    while (i < n) {
        sdata[tid] += g_idata[i] + g_idata[i+gridSize];
        i += gridSize;
    }
    __syncthreads();
    // do reduction in shared mem
    ...
    // write result for this block to global mem
    ...
}
```

Soumavlit Dev, Assistant Professor, CSE, IIT Kharagpur



I am saying that we can increase the activity here, so that would mean, I can do some reduction on the global memory and then do the rest of the reduction in the shared memory. So, here for and off course, since this is a par thread activity, this is going to be sequential, so as you can see there is similar set of code here, this is a; but this is sequential, so each thread is doing computation on the global data that is as working and then it storing here into the shared memory, it is bringing in more data and like that up to the grid size.

So, essentially you are going to do; all you are doing is for each thread, you are doing some part of the reduction sequentially in the global memory, so you read form global memory and write to shared memory, earlier we are doing one addition but here instead you are doing a sequence of additions. So, if we start here with the tid, then you compute for this thread, what is the global data is going to bring that data plus its block size and then you make it warp for some iterations with respect to the grid size.

(Refer Slide Time: 38:56)

Reduction 7: Analysis

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction Unit	Time Second	Bandwidth GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098
Reduce 5	0.00836	32.1053
Reduce 6	0.00769	34.9014
Reduce 7	0.00277	96.8672



Source: NVIDIA CUDA

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And you do instead of doing only on 1 addition, you make it do multiple additions here and with that you increase more amount of par thread activity and that would give you further reduction by this step. So, just if you want to just go through this code here, let us have a look again, so you are using this thread ID from your threadIdx.x, so then inside your block, what is this tid doing?

So, this thread for its component of the shared memory, off course, you have defined the shared memory here, and then for this thread, for its component of the shared memory is going to do multiple adds here from the global memory as you can see, so you are accessing the ith global location with the offset of a block size. So, you are going to add across this off set of a block size.

And then you are going to put it here. So, just have a look into this, we will come back to this reduction again to get into the intricacy of this reduction for right now, the basic idea will I like to say is that earlier we just outsourced one par thread addition to all the threads for doing a global add, we are just trying to bring in some more sequential activity par thread to do a load balancing here.

So, with this we will end this lecture, thank you for your attention. From the next lecture, we will go into this in a bit more detail and we will do performance analysis of each of the reductions and continue with further topics, thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology - Kharagpur

Lecture – 30
Optimising Reduction Kernels (Contd.)

Hi, welcome back to the lectures on GPU architectures and programming. In the last lecture, we have been discussing some more parallel reduction techniques and if you remember, there were 2 specific optimizations we talked about in the last lecture; oh, sorry, 3 specific optimizations; one was the unrolling of the last warp and after that we also discussed unrolling the entire computation.

(Refer Slide Time: 00:57)

Reduction 5: Kernel

```
__global__ void reduce5(int *g_idata, int *g_odata, unsigned int n){  
    extern __shared__ int sdata[];  
    // reading from global memory, writing to shared memory  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;  
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];  
    __syncthreads();  
    // do reduction in shared mem  
    for (unsigned int s=blockDim.x/2; s>32; s>>=1) {  
        if (tid < s)  
            sdata[tid] += sdata[tid + s];  
        __syncthreads();  
    }  
    if (tid < 32)  
        warpReduce(sdata, tid);  
    // write result for this block to global mem  
    if (tid == 0)  
        g_odata[blockIdx.x] = sdata[0];  
}
```



And at the end, there was algorithmic cascading. So before moving on, we will just revisit here for some more intricate points for each of these 3 optimizations that we had. So, first of all when we were trying to do the unrolling for the last warp; this was the code here. So, when the tid is less than 32, for that we are just unrolling the last warp and calling this to warp reduce function.

(Refer Slide Time: 01:29)

Reduction 5: warpReduce

```
_device__ void warpReduce(int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```

Now, for the other tid's, we will be executing the normal loop and just to be sure, I want to check out here that why certain things went missing in this code. We actually discussed what is the advantage of writing it like this. First of all if you remember, that we could remove the loop out of the equation and that actually remove the loop overhead associated and essentially, the loop behaviour of the last warp gets executed in sequence.

So, I do not have the checks of the loops to be done again and again, because as we can say this is completely unnecessary; because I am essentially writing the same behaviour that would repeat for all the statements here and since the loop is not there, I do not have to do this check by executing unnecessary instructions; advantage one. The second advantage was that since, it is the warp of instructions, I do not need the same thread as we have already discussed.

And also something important that we have here that there was this check inside the loop whether tid is less than s or not. Now, off course, when I am unloading the loop, these are the instructions which I should repeat but as you can see that if instruction is also missing. Now, I feel it is very easy here to understand that we do not really need the if instruction here because it says that, okay, if I am only interested in the threads inside the stride here.

But here, since I am only considering the last warp and going to consider the warp with the thread id is from tid 0 to 31, so technically I only want that warp to make progress here. The issue is why do I remove this if condition? Well, even if the other threads are executing, I do

not mind because finally, this is the warp which I am interested in which contains the thread id's from 0 to 31, only those will actually get through this if condition.

So, this if condition is already ensuring the warp that actually gets executed for this warp reduce function is basically this sequence of instructions and here only the tid's that are having the value 0 to 31 will come here as a complete warp. Now, since I have; I am actually interested to execute the warp and let it flow through all these sequence of instructions.

And I do not really care whether I have this check of tid less than s or not, I mean, it is easy to see here, you can just check that, okay, finally I will just pluck out the result from the thread with tid 0 but since, the warp is anyway progressing in lockstep is completely unnecessary to put in that condition again for each of these instructions here. I do not; I hope that is clear now, that since here I only have this warp containing the instructions 0 to 13; the thread id is 0 to 31 executing together.

They are anyway executing in lockstep, if I put in a check here, it really does not matter, only thing that it will give me some more instruction over it. I want to eliminate that so, for that reason it does not; it is not present here and anyway, functionally I get the value computed that I required and it is stored in tid 0. So, off course, all the tid's are doing the computation but by the behaviour of the code, I will get `sdata[0]` containing the final value once all the threads inside this sequence finish executing their respective instructions here.

So, that would be our discussion for the warp produced just wanted to highlight, off course, we have discussed earlier why `__syncthread` should be not there, loop should be not there, we wanted to push in the fact that, okay, this check is also unnecessary. Now, if you want to further motivate it, you can also look at the situation that would happen with, let us proceed, the part you can think over this part also.

(Refer Slide Time: 06:07)

Reduction 6: Complete Unrolling

If number of iterations is known at compile time, could completely unroll the reduction.

- ▶ Block size is limited to 512 or 1024 threads
- ▶ Block size should be of power-of-2
- ▶ For a fixed block size, complete unrolling is easy
- ▶ For generic implementation, solution is:
 - ▶ CUDA supports C++ template parameters on device and host function
 - ▶ Block size can be specified as a function template parameter



(Refer Slide Time: 06:13)

Reduction 6: Kernel

Specify block size as a function template parameter and all code highlighted in yellow will be evaluated at compile time.

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n){
    extern __shared__ int sdata[];
    // reading from global memory, writing to shared memory
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
    __syncthreads();

    // do reduction in shared memory
    if (blockSize >= 512) {
        if (tid < 256)
            sdata[tid] += sdata[tid + 256];
        __syncthreads();
    }
}
```



So, next the thing that we attempted was complete unrolling and there we brought in the concept of templates and with templates, our idea was that. Okay, block sizes are going to be multiples of 2 and anyway in the GPU we are limiting our block sizes to the sizes here in the code that is up to maximum; we are allowing is 512. So, anyway I will not need the for loop since, I know that instead of the loop, if I come to a complete unrolling, I can write a sequence of if-else blocks.

(Refer Slide Time: 06:41)

Reduction 6: Kernel

```
if (blockSize >= 256) {  
    if (tid < 128)  
        sdata[tid] += sdata[tid + 128];  
    __syncthreads();  
}  
if (blockSize >= 128) {  
    if (tid < 64)  
        sdata[tid] += sdata[tid + 64];  
    __syncthreads();  
}  
if (tid < 32)  
    warpReduce<blockSize>(sdata, tid);  
  
// write result for this block to global mem  
if (tid == 0)  
    g_odata[blockIdx.x] = sdata[0];  
}
```



Viewing Reduction 6

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

That if block size greater than 512, this will happen, followed by greater than 256 this would happen so, this would be the sequence here. So, anyway if the block size is 512 for that, I would get in here and execute this entire sequence of code, if it is 256 then, I will not need the previous if condition but I should start from here, so on so forth.

So, the question that is also interesting here is; since the block parameter is templated, it is interesting to figure out that how I can make the system know that what is the block size that is going to be finally used and accordingly, the compiler can resolve these branches, this is something we did not discuss, so let us focus on that aspect here. So, just think if we are going to run the corresponding host code for this reduction, so here we have unrolled the complete loop, it is beneficial here.

Let me again point out because the maximum block size we are considering is limited to these values.

(Refer Slide Time: 07:49)

Reduction 6: Kernel

Modified warpReduce function:

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid)
{
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8)  sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4)  sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2)  sdata[tid] += sdata[tid + 1];
}
```

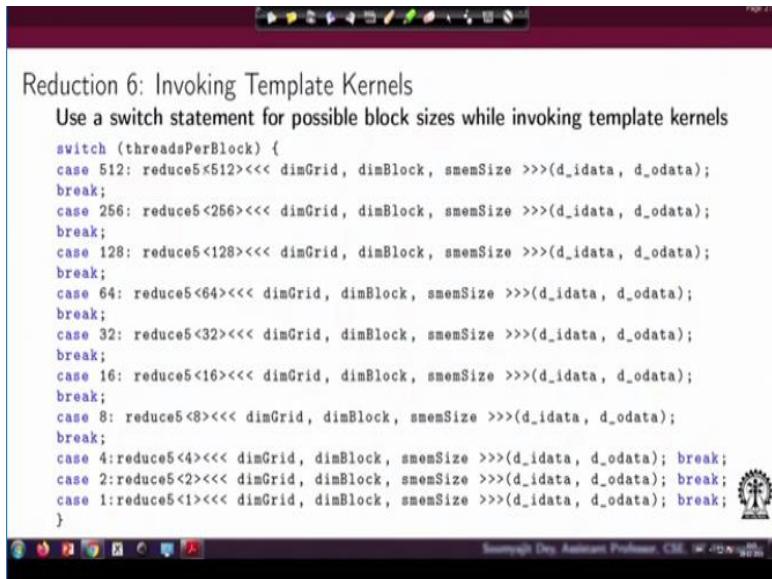


Now, this would be also the corresponding warp reduced function, I mean if you just see that well again, when the tid is less than 32 just like previous case, I am calling a warp reduce function now, the warp reduce function is also being templated with the block size. So, inside the warp reduce function I have this sequence of sdata computations but now, I have an extra check here whether the block size is greater or equal 64, greater equals 32, 16, 8, 4, 2 like that.

I mean it is again coming because, off course, it may so happen that in my block size is smaller than 64 in that case, I do not really need to execute this instruction, I will start from the next. So this is the reason for which I will now push in this block size parameter here provided I am using the warp reduce function in conjunction with a templated version of reduce 6.

I hope this is clear, I am just trying to push in the point here that if we are going to use reduce since in a templated version with block size templated, then correspondingly I also need to make a template function called for warp produced with block size being a template parameter.

(Refer Slide Time: 09:14)



```
Reduction 6: Invoking Template Kernels
Use a switch statement for possible block sizes while invoking template kernels
switch (threadsPerBlock) {
    case 512: reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata);
    break;
    case 256: reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata);
    break;
    case 128: reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata);
    break;
    case 64: reduce5<64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata);
    break;
    case 32: reduce5<32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata);
    break;
    case 16: reduce5<16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata);
    break;
    case 8: reduce5<8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata);
    break;
    case 4:reduce5<4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 2:reduce5<2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 1:reduce5<1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
}
```

The good thing here is look at the corresponding call from the host side for this kernel, so just a small correction here, this is the reduce 6 kernel, so consider this as reduce 6; so this is the host side call. Now, as you can see here when you make a function call to this templated function, this is your normal function call of the kernel, where for the function call of the kernel reduce 5, you are passing the dimension of the grid, the dimension of the block and also the shared memory size. So that is a new function parameter which we are introducing here just to let you know that when you launch a CUDA kernel as a launch parameter, you can pass the block and the grid dimensions and along with that, you can also pass the maximum shared memory value, this kernel would be using. So, that is additional parameter that we have introduced here.

Now, observe that since it is call to a templated function, you can also pass the template parameter; okay, I want to call it with a block size of this, so from a user or somebody from some other file, if I read what is the threads per block, then I can match the suitable case here whether it is 512 or 256 or 128 like that and accordingly, pass this as a template parameter.

Now, what is the good thing about this passing the template parameter; well, this is then resolved here that what will be the call. Now, when the compiler will be compiling this code, it will be ready with all these different versions of the reduce 6 kernel, so it will actually be ready with these versions of the reduce 6 kernel because it knows that the block size values can be so on so forth.

And accordingly, using the block size value, it would have resolved these if-else statements of the block size. Now that would mean, when you have the application binary using this template, you have actually got reduce 6 kernels different versions, where this block size greater than equal to some value, this if conditions has been resolved in the compile time. I hope this is clear; these are standard property of templatizing a function call or a class in C++.

You resolve dependencies, you met analysis and you are able to resolve as many possibilities as possible and accordingly, you create different binaries. So, when you are really executing the kernel, you are not going to really execute any branch instruction here that is the most important point. So, depending on the template parameter that has been passed, you will be executing a version of the kernel where it is already known from which if-else block you start.

And you just do the corresponding addition computations, the parallel reductions computations for this kernel as well as for the warp reduce function. So, that is how templatizing and complete unrolling achieves its full parallelism here. First of all, you have removed the loop and also you have removed the if-else blocks during the compile time analysis.

So, this is also the advantage here just so, in the first case when we unrolled the warp 0 with the tid is 0 to 31, the good thing was; we are able to remove the for loop, we were also being able to remove the if condition because as we figured out, it is completely unnecessary. Because I am finally only interested in the warp containing that tid is 0 to 31 and that is the only warp which is being made to progress through this if condition.

So, when that warp is executing, I do not need to really do any final check and unnecessary waste time, let the warp execute without any kind of divergence because finally, I will only be using the data, sdata[tid] is sdata[0] that was I mean summarizing here for completeness, so that is how reduction 5 is done and this is how reduction 6 helps 0.1, you completely unroll the loop, 0.2 you resolve the if condition statically.

So, you really do not get the code diverging at these points both for the reduce 6 kernel and as well as for the call to the warp reduce function. So, this is very clear I hope now, with this

incorporation with the switch statement based function calls from this the possibilities are already known.

(Refer Slide Time: 14:09)

Reduction 6: Analysis

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

- Algorithm Cascading can lead to significant speedups in practice

Reduction Unit	Time Second	Bandwidth GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098
Reduce 5	0.00836	32.1053
Reduce 6	0.00769	34.9014



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, the last thing that we were discussing was algorithmic cascading, so if you remember the basic idea was that okay, we are going to increase the number of part thread activities.

(Refer Slide Time: 14:22)

Reduction 7: Multiple Adds / Thread

Algorithm Cascading:

- Combine sequential and parallel reduction
 - Each thread loads and sums multiple elements into shared memory
 - Tree-based reduction in shared memory
- Replace load and add two elements
- With a loop to add as many as necessary



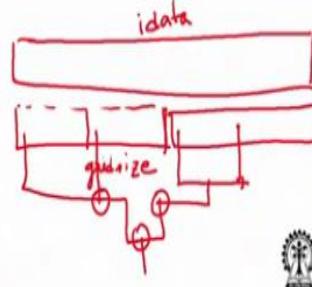
Earlier in one of the optimizations, we increase the part thread activity by doing the fast addition after global load.

(Refer Slide Time: 14:27)

Reduction 7: Kernel

```
__global__ void reduce7(int *g_idata, int *g_odata, unsigned int n)
{
    extern __shared__ int sdata[];
    // reading from global memory, writing to shared memory
    unsigned int tid = threadIdx.x;
    unsigned int i = blockDim.x * (blockDim.x / 2) + threadIdx.x;
    unsigned int gridSize = blockDim.x * 2 * gridDim.x;

    sdata[tid] = 0;
    while (i < n) {
        sdata[tid] += g_idata[i] + g_idata[i + gridSize];
        i += gridSize;
    }
    __syncthreads();
    // do reduction in shared mem
    ...
    // write result for this block to global mem
    ...
}
```



Here we are generalizing it that instead of doing the 1 addition, we will do multiple additions here. Now, if you look into the code here so, let us draw a figure to better understand what is really is happening. So, consider that this is your actual arrangement of the data and if I do a mapping of the actual threads you are going to launch let us say, you are going to launch this grid of threads, okay.

So, since this is a 1d situation, so let us say this is your set of the data that you want to reduce but you are going to launch less number of threads, so your threads; the grid of the threads is up to this and let us say 2 of them are able to cover up for the entire situation. Now, inside the thread so, this is your grid size, this entire thing is your grid size and now inside your grid, let there be 2 blocks like that.

So, what we are doing here in the reduction step is as follows; so first thing you do is you compute the thread Id here and then you compute fellow i which is essentially block Id times block dimension times 2 plus thread Id, the 2 factor comes because of course, we are going to each thread would be loading 2 values at a time and then doing the addition. So, when I compute the i ; the i is basically indexing how many addition operations I am going to do here.

So, there is the; is basically iterating over that space so, every time I am going to load 2 of the elements from the same block and do the addition, so when I compute this index i , I do a block Id times block dimension times 2, so that is why this 2 factor would come in here and

then you multiply by and then you add the threadIdx for the offset. Now, this is your grid size, so grid size is again the grid dimension times block size times 2.

The factor of 2 is again here to take care of the fact that we just discussed that every thread would load 2 and then do the addition. Now, when the addition comes, so what essentially you do is; so you are trying to make each thread, add, perform multiple additions instead of one addition, each addition will take 2 data points, so you have this multiplied by 2 factor here as well as here.

So, when you add, you add across blocks, so the first addition would be of 2 blocks considering this i as zero here, so that is the addition then you do and then you hop to the next part of the data, so this is my grid size and I am saying that the total amount of data that I have is a multiple of the grid size so, you go to the next part of the data of the chunk that is of size grid size.

And then again, you load the data from the value at here as i and then shift by block size and this is how you progress. If there are more than, you can again hop to the next part for another grid size part and then again, there you do 1 addition and like that, so technically speaking, in each side, in each grid size chunk of data, what you are doing is each tid is computing an index i .

And it is shifting by a block and so it is essentially adding from i index plus 1 shifted with a block size. So, in that way if I am technically saying that okay, I consider a data with multiple chunks of grid size present, then for each tid I am effectively doing and let us say the number of such chunks is some k , then this i variable and n is of course, the total n , so I am going to increment i in step size of grid size.

And since, the total data size is k times the grid size so, i would increase in steps of k and I will be able to do k number of additions here, so in that way this is the general addition that if I have a large data block with this step, I am reducing the data block by doing additions on global memory data, then storing into the shared memory. So, with this step I am reducing the data block by a factor of k to a single grid size.

And then, again I am so essentially, what I am doing is for different grips, I am bringing the data and doing the addition across blocks, so it is not only the case that I am bringing data from different grid sizes also, I am doing 1 reduction by taking data from 1 grid and adding with a block size offset. So, I hope this is clear, in the original version you are considering only this as your width and you are doing 2 additions, you are doing 1 addition for 2 data points at a block width.

But now, what you do is; you consider the actual data of much more size by a factor of k, so here you take a; you not only do an addition inside the grid size but you do such k number of additions across different chunks of grid size and then you reduce this entire thing to a smaller block of this size, the grid size and then go for the shared memory based reduction.

So, this is how the thing progresses here also, there is something important which we have not discussed till date that what is the way the host code looks, because we have discussed that okay, I can have multiple blocks of data for each block how the kernel will do the reduction, but in general we have always said that okay if there is a lot of blocks, then the host code will take care of launching the kernel reducing the data in sizes of blocks and then keeping; I mean keeping and it will just keep doing this on and on until you get to the final block and reduce it to a single value.

(Refer Slide Time: 21:55)

Reduction 1: Host Code for Multiple Kernel Launch

```
...//cudaMemcpyHostToDevice...
int threadsPerBlock = 64;
int old_blocks, blocks = (N / threadsPerBlock) / 2;
blocks = (blocks == 0) ? 1 : blocks;
old_blocks = blocks;
while (blocks > 0) // call compute kernel
{
    sum<<<blocks, threadsPerBlock, threadsPerBlock*sizeof(int)>>>(devPtrA);
    old_blocks = blocks;
    blocks = (blocks / threadsPerBlock) / 2;
}
if (blocks == 0 && old_blocks != 1) // final kernel call, if still needed
    sum<<<1, old_blocks/2, (old_blocks/2) * sizeof(int)>>>(devPtrA);
...//cudaMemcpyDeviceToHost...
```



So, going back and revisiting our host code, so this is the code that actually takes care of launching multiple kernels across different blocks. So, consider that we have threads per block equal to 64 and we have 2 of these variables; old blocks I mean, the blocks previously

to the reduction step and the blocks right now, so what we do is the number of blocks is total data size n divided by threads per block divided by 2.

Because of that global addition, we are considering the global addition scenario; 1 addition per thread, so first thing is we initialize the blocks value and put that value into the old blocks value and then inside this while loop, what we are doing is; we call this kernel sum with parameters blocks threads per block and the amount of shared memory that we want to pass it.

So, this would make a reduction and then you have this number of blocks which you store as old blocks and definitely, you have to reduce the number of blocks here now of course, we know the reduction factor is you reduce the number of blocks by the threads per block because every block gets reduced to a single value and because of one global memory addition get further reduces by a factor of 2.

So, that will be the new value of the number of blocks after one step of reduction by this kernel. I hope this is clear so, you are calling this kernel over blocks and threads per block parameter and finally, after the kernel returns, you have to recalculate the number of blocks that are there and that would be since each block gets reduced to 1 value so, blocks divided by threads per block.

And if you consider the situation that each thread before going to a reduction step is doing 1 global addition, so that would blocks actually also reduced by a factor of 2, so this is the final value of the updated number of blocks, you are going to keep on computing this as long as the number of blocks is greater than 0, if its final equal to 0 or this whole blocks is not equal to 1, then you launch the final kernel to reduce the remaining block to a single data point.

So, this is the host code that we skipped earlier so, with this now we have finished our discussions on the different possible reductions.

(Refer Slide Time: 24:24)

Reduction 6: Kernel

```
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n){  
    extern __shared__ int sdata[];  
    // reading from global memory, writing to shared memory  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;  
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];  
    __syncthreads();  
  
    // do reduction in shared mem  
    ...  
    // write result for this block to global mem  
    ...  
}
```



And so the final reduction, I will just reiterate here what we did was that we actually increase the amount of work to be done on the global memory by considering that okay, the grid size we actually hop over the grid size chunk, get into the other chunks of data and also perform an addition step block wise with a stride of block size for each thread, each thread is computing; is doing for its value of; unique value of i that the thread would compute for a corresponding block.

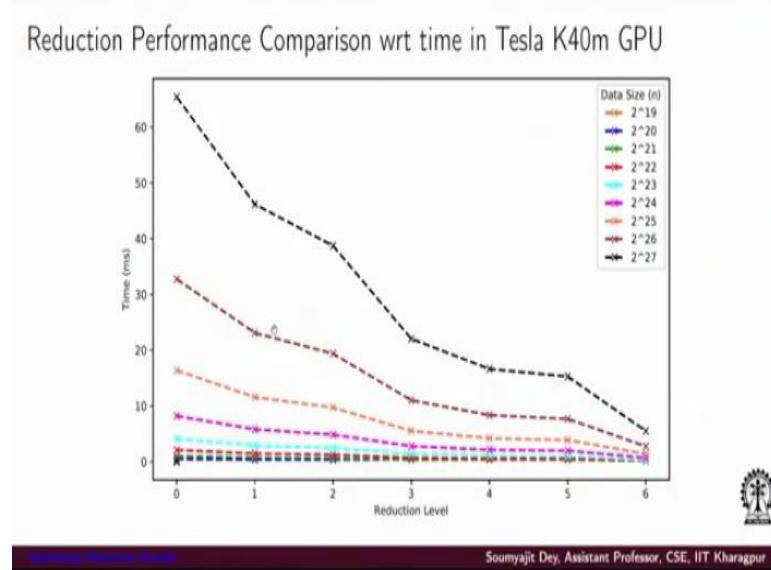
It will consider that corresponding location in the global memory and the next block and do the addition, so I mean I hope this is clear, the part thread activity which is very important. So, the first point as we mentioned was that each thread would be hopping over grid size chunk of data and doing 1 addition but the also, the important thing is what would it be doing inside 1 grid sized chunk of data, what it would be doing is; it will be computing 1 offset parameter i .

So, essentially it knows in which block it is, it will multiply by 2 for the global add considering and then multiply by block dimension get into its own block, then it will be get into this thread Id for that position and the corresponding offset position in the next block, it will be doing the addition inside a grid size chunk of data and then would again shift to the next grid size chunk of data.

This is how it would carry on the computation here okay, so with this we will add; we will actually end our discussion on performing addition in parallel to be specific and as you can

see that this addition can be other reduction operation, the same methods would actually be holding up here.

(Refer Slide Time: 26:23)

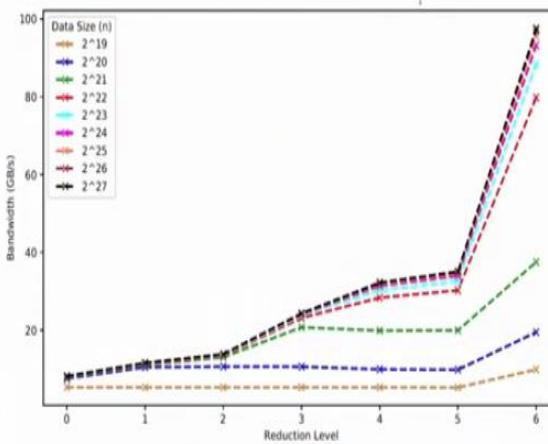


So, thank you for your attention, in the next lecture we will be actually go into a different topic some other algorithms, now just for your reference we provide some data here that how parallel reduction performs for the varying data sizes, so as you can see we are varying the data size here so, this black line has got the highest data size, so and for that we are showing that with the application of each reduction step, what is the decrease in execution time that we get.

So, this one gives is the actually I am highlighting the one with 2^{27} data size because you can see that maximum speed of you are possibly getting here and the experiment is done on Tesla K40 GPU.

(Refer Slide Time: 27:10)

Reduction Performance Comparison wrt bandwidth in Tesla K40m GPU



Reducing Data Size

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And similarly, now if we calculate the bandwidth utilization which is a primary reason for getting the speed up, as you can see for this 2 to the power 27 size, we have the maximum bandwidth utilization of course, it is closely followed by the other data sizes here okay, with this we will end our discussion today, thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology - Kharagpur

Lecture – 31
Optimising Reduction Kernels (Contd.)

(Refer Slide Time: 00:28)

Example of Applications on Reduction

- ▶ Bitonic Sort
- ▶ Prefix sum

Hi, welcome back to the lectures on GPU architectures and programming. In the last set of lectures, we were discussing the way the standard reductions can be optimized in with respect to GPU based parallelization. Now, we will go into some more algorithmic examples to be more specific, we will be talking about 2 examples; one is bitonic sort and the other is computation of a prefix sum.

(Refer Slide Time: 00:55)

Problem: Sorting

- ▶ Sort any random permutation of numbers in ascending or descending order
- ▶ Basic introduction to sorting networks
- ▶ Focus on a comparison based sort - Bitonic Sort
- ▶ Discuss how operations can be parallelized using CUDA.



So, let us get started with this, I mean, when we are talking about bitonic sort; the basic problem of sorting is, you are given as input any random permutation of numbers and you want to output a sorted set which is either ascending or descending in there with respect to their respective keys. As we know that there are a lot of well-known comparison based sorting algorithms.

So, in this case we will be choosing some algorithm, which is very well known for the amount of parallelization that it offers and so before going into this, I mean, the idea of what basically is bitonic sort, we let us start with an introduction of sorting networks and because that is a key idea which will be used in the context of bitonic sorts.

(Refer Slide Time: 01:45)

Sorting Networks

A sorting network is composed of two elements

- ▶ **Wires:** Wires run from left to right, carrying values (one per wire) that traverse the network all at the same time.
- ▶ **Comparators:** Comparators connect two wires. When a pair of values, traveling through a pair of wires, encounter a comparator, the comparator may or may not swap the values.



So, first of all what is a sorting network? So, essentially it is a computational model I would say, network which is doing a sort which is essentially being used for performing the function of sorting, we are assuming that say, for example consider that you have some wires, the wires are running from left to right and they are carrying values. So, each wire is carrying some physical signal value, let us say some numerical value in terms of bits like that.

Consider an obstruction here now, the values are traversing through the network from the left to the right and in between, we have some comparator circuits connected to the wires. So, whenever I have a value coming in one input of the comparator circuit through one wire and another value coming through another input of the comparator circuit to another wire, the circuit will make a comparison.

And depending on whether I want the max or the mean, it will, I mean, if I want that the maximum value should flow up or the maximum value should flow down accordingly, the comparator will work and swap the values across the wires, so that is the basic functionality here.

(Refer Slide Time: 02:58)

Comparator

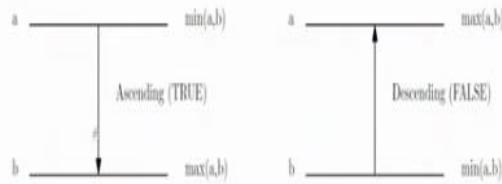


Figure: Comparator Function



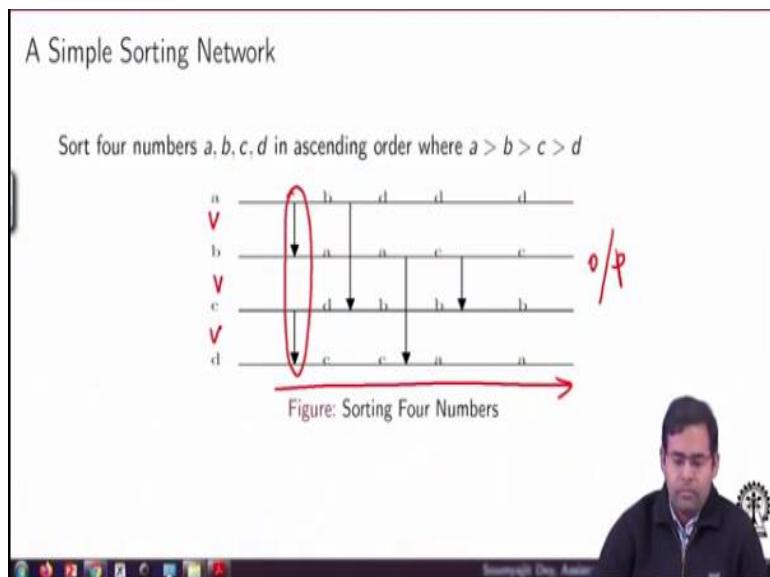
Let us take an example, so what we mean by this comparator operation? So, these are my wires running from left to right, they are being provided with this inputs a and b and this notation; this symbol down; this arrow pointing downwards, it is essentially telling me what I want is whatever is the input here, the comparator circuit is going to compare and find out what is the bigger value.

And it will push the bigger value to the downward and it will pop the smaller value to the upward wire. Now, off course, if a is anyway less than b, then nothing is going to change, if a is greater than b, then there will be a swap operation here. So, essentially the comparator ensures that whatever is the input at this point, I have the upper value to be less than the lower value that is the invariant that is going to be ensured in the output lines here, in this pair.

Now, similarly if we just reverse this arrow that means, we are looking for a comparator circuit which again compares a and b, the inputs and in the output, it is floating the maximum value in the upward wire and it is bringing the minimum value downward, so essentially we are saying that okay, we are trying to do a descending output here and here, we want a ascending output.

So, that is the primary goal of the comparator circuit, it is in this case, you can just remember like this, the maximum value follows the arrow that is a simple way to remember. The max value is always going to follow the arrow in this case, it is going to percolate down in this case, it is going to percolate up so, here again in the output, the invariant that would be true is whatever is the value that I have in the wire here is always greater than the value that I have in a wire here.

(Refer Slide Time: 04:57)



Now, what is the usefulness of these circuits and the sorting network we want to build? Now, consider a set of wires here, we have a more involved example, we are trying to build what we essentially mean by a sorting network. So, essentially it is a collection of this kind of wires and comparative circuits are placed like this. So, the important thing we need to understand here is the flow of time.

So, essentially we would mean that leaves the input and this is the time following which, I mean this is the flow of computation following the time axis and here we are looking for the output. So, now if you observe what is going on suppose, the input is already in ascending order sorry, the input is in this order essentially, it is a descending order but we want the output to be in the ascending order.

So, since a is greater than b and b is greater than c , so on and so forth, in the output d , being the smallest element, if we want the output in ascending order, d should come followed by c , followed by b and followed by a and we can just to a small chip that this sorting network can

realize this functionality, so let us see what is happening. So, we have given a and b as input now, since a is greater than b, so a is going to flow down here.

The max will follow the arrow, a comes there, b goes there similarly, c comes here, d goes here again, as we can see that a is definitely also greater than b by transitivity and so b is also greater than d here. So, d will go up and naturally, b will come down and similar behaviour we can take here. So, what is going to happen is after this again, we are comparing a and c here, so again c will go up and a will go down so on so forth.

Now, what is important to point out here is, you can look at the set of comparisons that are going on with respect to time and try and understand what are the activities that can be done in parallel, so essentially as long as I do not have a dependency between the inputs and outputs, those comparisons can easily be performed in parallel for example, here this set of comparisons, I can easily do them in parallel.

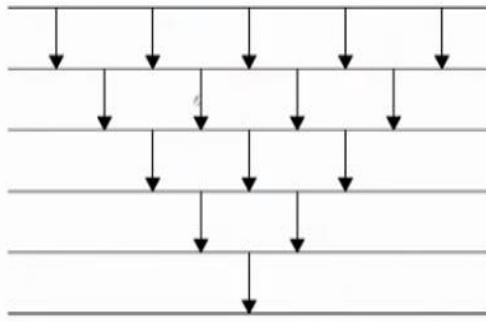
And can I do this too in parallel? Okay why not, because apparently there is no dependency between the inputs and outputs; because here you are comparing a and d and here you are comparing a, sorry, here you are comparing b and d and here you are comparing a and c, so even this could have been done in parallel. Now, what is happening is the switching of the values between the wires go on with respect to time.

If we see the flow of values, the circuit is going to ensure that at the output, I get the values in ascending order; I get d followed by c, followed by b, followed by a. So, one thing we can now understand, it makes sense to think like this that I can use this kind of a sorting network to represent the underlying algorithm of a sorting function; any sorting algorithm.

For example, bubble sort should have this kind of a sorting network representation or insertion sort should have a separate different, since it is a different algorithm, it should have its own sorting network representation.

(Refer Slide Time: 08:43)

Bubble Sort



Any comparison based sort can be done using a sorting network.

Soumyajit Dey, Asstt



So for example, this as we can see that this is a sorting network, which is going to do a bubble sort, well, we will not discuss anymore here, we will provide the slides you can just look into this and understand why we are saying this represents the bubble sort algorithm here. Of course, the generalization would be that any comparison based sort algorithm can be represented using this kind of a sorting network.

So, the way we have drawn the picture is to make you understand what are the operations we can actually do in parallel without any dependencies and in that way, bubbles are or any other sorting algorithm as you can see, if we take a sorting network representation, it actually reveals to us what are the operations that are actually possible in parallel; that is the most important thing.

(Refer Slide Time: 09:40)

Bitonic Sort

Bitonic sort takes place using two fundamental steps:

- ▶ Step I: Convert an arbitrary sequence to a bitonic sequence.
- ▶ Step II: Convert a bitonic sequence to a sorted sequence.

A Bitonic Sequence is a sequence of numbers which is first strictly increasing then after a point strictly decreasing. $a_1 < a_2 < \dots < a_m > b_1 > b_2 > \dots > b_n$

Soumyajit Dey, Asstt



So, coming back to our idea, here we want to talk about bitonic sort; so this example; this algorithm has got 2 fundamental steps. The first step is that you convert any arbitrary sequence of numbers to a bitonic sequence. Now, what is the bitonic sequence? It is a sequence of numbers which is first increasing strictly and then it is decreasing strictly for example, this so, it is increasing up to a_m .

And then it is from a_m to b_m , it is strictly decreasing; so this is a bitonic sequence. The first step of the bitonic sort method is to consider as input any permutation of values or key value pairs and convert it into this kind of a sequence and then the second step is well, you take this kind of a bitonic sequence as input and convert it into a sorted sequence. Just coming back once to this example, I hope from this it is clear to you that why we are trying to say that this is essentially a representation of bubble sort.

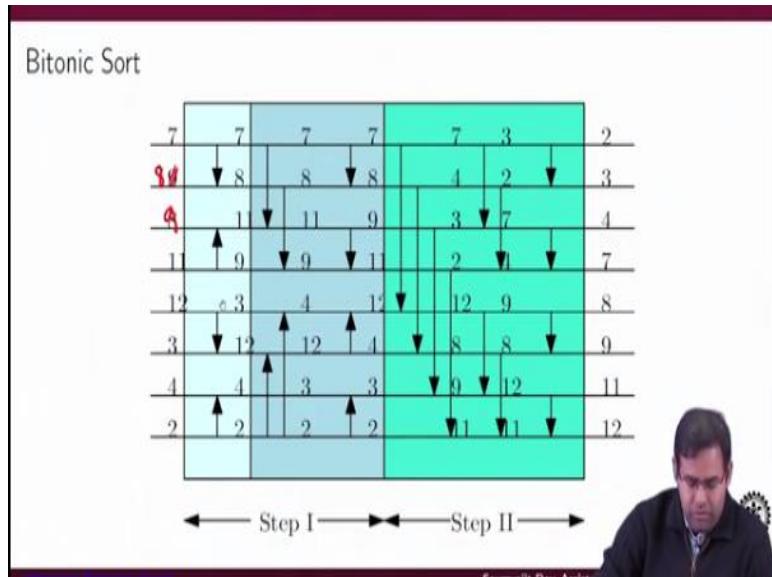
As you can see that if I just follow the track of these downward arrows, the max value is kind of flowing down, the next max value will also similarly like flow down. The way we are drawing this picture is we are trying to show which are the operations that can be done in parallel, so, maybe you can consider this kind of picture and try and figure out why this represents a specific sorting algorithm.

And you can also create sorting network representations of other sorting algorithms and figure out what is the parallelisation that is possible for that algorithm. So, coming back to this bitonic sort idea, so as we can understand the step 1 is to create the bitonic sequence, the step 2 is to take as input the bitonic sequence and then create a sorted sequence. The sort algorithm is actually going to implement these 2 steps in form of a and we shall be trying to view those methods in the form of a sorting network.

And then use the sorting network to identify, what are the operations that can be done in parallel similar to the previous examples of sorting algorithms, I will just repeat; so these are the 2 fundamental steps we are going to realize through a sorting network so, in step 1 we are going to convert, we are going to implement a sorting network which can take as input an arbitrary sequence convert the sequence to this kind of a sequence where it first increases and then it decreases, the sequence of key value pairs decreases.

And then, we have a stage 2 sorting network, which will take this kind of a sequence as input and it will provide a completely sorted sequence.

(Refer Slide Time: 12:50)



So, let us look into this for once, so let me just do a few corrections in this picture, here I should have 8 and this should be 9 in the input, I hope the rest is correct here yeah, okay. So, we are trying to give here a sorting network using which we are performing the bitonic sort. So, we have these inputs; 7, 8, 9, 11, 12, 3, 4, 2 as you can see, it is an arbitrary permutation and we are trying to mark out the different steps here.

So, I have a step 1 that means, as per our previous definition here, step 1 is going to create a sequence here where the values will first increase and then the values are going to decrease. Now, let us look at the network and see whether that is really happening or not so, in the first stage again, I will just repeat, this sorting network is trying to represent the operations with respect to time.

That means, at this time slot I can do these operations in parallel, in the next time slot I can do the following operations in parallel, in the next timestamp I can do the following operations in parallel. Now, of course in between operations happening at different time stamps, if there are no dependencies, then you can parallelise the operations across time stamps, we have to understand this, we will give examples on that.

So, first point I am just repeatedly trying to make is, what is the sorting network trying to show you, it is trying to show you that this is the flow of time and at each time instant what

are the operations that I can do, go to the next time instant what are the operations I can do and again I will just repeat, if there are operations sitting at different time stamps but they really do not have any input output dependency, they can also be parallelised.

So, just; let us just first look at here the activities that are happening and try and figure out whether the functionality of the sorting algorithm is actually going to be realized through this sorting network. Well, first of all as we can see, there will be no downward flow here because 7 is always; 7 is of course less than 8, so the max is already here nothing flows down, here the max is below, so it will flow up, 11 goes up and 9 comes down.

Similar thing here and nothing changes here, here what is happening; again, I have to compare between 7 and 11 nothing changes, 8 and 9 nothing changes but now 3 and so, now here, I have to compare 2 and 12 again, nothing changes and then I compared 2 with so sorry, so here we are comparing 2 and this value 12, so that remains same and then in this line, we are comparing the value of 2.

So, here we are comparing 2 with the value of 4 that is done and here, we have 2 and 12 and once I compare them again, there is no change that is going to happen and then again, when I compare here yes, so I will expect the maximum value to go up and it is already up here as I can see and just one minute, I think here so, the 4 is being here and yet again yeah, so the issue is yeah, this is from where it will start.

And so what I am going to get is; so I am comparing 2 and 12 nothing changes here and yeah now it is fine so, I am going to compare between 4 and 3 so, the 4 is the max it should go up following this so, 4 comes up and 3 goes down, so, pardon the corrections here I have to do, extremely sorry for that and now, when I continue so again, I go to the next stage, nothing changes here but here I have 11 which should come down, and 9 is going to come up.

And similarly, here 12 flows up, 4 flows down and then again, here nothing will change because 3 is the maximum and it is already here. So, now let us observe the outputs there because we are considering that step 2 finishes here. So, what are the outputs? As we can see in the output, I have 7, 8, 9, 11 followed by 12, 4, 3, 2 so, it is ascending and then is descending.

So, it is ascending here and then is descending here, so that means this is a bitonic sequence because if you go back to our previous definition, a bitonic sequence is a sequence of numbers which will first strictly increase and then it will strictly decrease after that point, so that is what is happening here. Then, we are going to the step 2 and in step 2, the definition is that okay, I am expecting the input in the form for bitonic sequence.

And I will transform it into either fully ascending order output or a fully descending order output. So, in this case we are trying to transform it to a fully ascending output and let us see whether it is happening yes, it seems so, for example I am going to compare now 7 and 12, they are already sorted, so then I compare 8 with 4, I exchange then I compare 9 with 3, nothing changes, then I compare 11 with 2 and so it changes within 2 and 11.

So that is done and then in the next stage, I compare 7 and 3, so again I have a change, I compare 4 and 2, again I have a change and here I compare 12 and 9, I have a change, I compare 8 and 11, nothing changes and then I am in the last change, where I compare 3 and 2 with a switch compare 7 and 4, again a switch 8 and 9, again a switch and 12 and 11, again a switch.

But what I get in the output as you can see is fully sorted in ascending order, so apparently these bitonic sort network that we have shown here does this job. Question is does it really do its job in general? Well, you can study a bit about this algorithm and look into its correctness proof which is there in almost all of the algorithms books that are available because given the scope that we have here, let us assume that it works.

But one fundamental thing we will do; we look into the algorithm and try and understand why it works without going into proofs and other techniques for showing the correctness of the method.

(Refer Slide Time: 20:57)

Recursive Structure

- ▶ If you look closely, Step I uses Step II recursively on smaller sequences.
- ▶ Step II can be used to sort in any order (ascending or descending). The order can be controlled using the comparator.
- ▶ Step I uses Step II in a way to construct subsequences that are bitonic in nature.

<http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm>



So, if we look into the intuition behind the algorithm what is really happening, observe something that step 1 is using step 2 recursively on smaller sequences. Now, why is that so? Let us go back to the definition of a bitonic sequence so, it is first going to increase and then is going to decrease, so then I would say that small as bitonic sequence would be to the just a collection of 2 numbers.

And then, if I apply my method of step 2, it is expecting as an input a bitonic sequence and it is going to provide as output in a sorted order ascending or at descending. So, essentially I can say that whatever is the method of step 2, it will take as input bitonic sequence and produce a fully sorted sequence. So, even for building the basic steps of step 1, I can apply step 2 recursively on the smaller sequences by starting with the smallest bitonic sequences which are of size 2.

I mean in that case for step 2, the only thing I will have is a single phase here, is that okay, you are going to; you are given the function bitonic sort which is expecting bitonic sort step 2 which is expecting the input to be as a bitonic sequence so, I can just say that the way I am going to draw up the algorithm is; so consider step 2, give it as input just as smallest bitonic sequence to different values.

And give the algorithm the order you want, that whether you want it to be an ascending order or you want it to be a descending order. What you get is output is 1 comparator, a smallest sorting network containing only 1 comparator. So, in that way I can keep on applying the step

2 recursively on smaller sequences from here and keep on constructing the different phases of step 1.

So, in that way we need to remember that for constructing step 1, I can look into it as constructing sub sequences by using smaller versions of step 2, if we look into this what is happening here; here step 2 is expecting an input which is the 8 size bitonic sequence. If I look into the method here, just this sub part, if you look into this sub part, then I can say that well what is the input and this input I mean, if this is an input; 4 size input, here you have a bitonic sequences of output.

Because it is ascending and then it is descending, and then if you look into this part you are essentially applying step 2 on an input of size 4, this is your step 2 on an input of size 8, here you are applying step 2 on an input of size 4. In that way if you generalize, you can look into every part of step 1 and start saying that this is nothing but a smaller instance of step 2, so this is the larger step 2, this is the smaller instance of step 2.

And this is the even smaller instance of step 2, which is nothing but a single instance of a comparator. So, we will use this recursive formulation to generate the algorithm.

(Refer Slide Time: 26:07)

The screenshot shows a video player interface. On the left, there is a code editor window titled "Recursive C Program" containing two functions: a comparator function and a bitonic merge function. On the right, there is a video frame showing a person speaking. Hand-drawn red annotations are overlaid on the video frame. The annotations show a sequence of four boxes labeled $a[1]$, $a[2]$, $a[3]$, and $a[4]$. Arrows indicate a comparison between $a[1]$ and $a[2]$, and another between $a[3]$ and $a[4]$. A bracket below the boxes is labeled $m=2$. To the right of the boxes, there is a small diagram of a T-junction with a downward-pointing arrow.

```
//Comparator
void compare(int i, int j, boolean dir){
    if (dir==a[i]>a[j])
        exchange(i, j);
}

//Step II
void bitonicMerge(int lo, int n, boolean dir){
    if (n>1){
        int m=n/2;
        for (int i=lo; i<lo+m; i++)
            compare(i, i+m, dir);
        bitonicMerge(lo, m, dir);
        bitonicMerge(lo+m, m, dir);
    }
}
```

First, let us look at a recursive C program which can perform the bitonic sort by using the idea we just discussed so, this is our basic comparator, it is considering 2 inputs i and j and it is given a direction. Now, look at the way we are writing the program here so, if the direction

value is 1, ai is greater than aj, so you are giving ai and aj as input and you are doing the comparison.

If the direction is 1, if ai is greater than aj, then you are going to exchange, so you are exchanging i and j, essentially in the output lines what you have; if ai is greater than aj, then you do the exchange and you get effectively the value here this and the value here ai. So, what we get here is the bigger value is flowing down so, this is the equivalent sorting network representation.

Because since ai is greater, you are doing the exchange, so the bigger value is flowing down and this is the network that you have. So, this way this small comparator program represents this sorting network with a downward direction. Now, look at the merge step and let us understand why it works, why do I say that the merge step is important?

(Refer Slide Time: 28:29)

Recursive C Program

```
//Step I
void bitonicSort(int lo, int n, boolean dir){
    if (n>1)
    {
        int m=n/2;
        bitonicSort(lo, m, ASCENDING);
        bitonicSort(lo+m, m, DESCENDING);
        bitonicMerge(lo, n, dir);
    }
}
```

A video player interface is visible at the bottom, showing a person speaking and various control buttons.

Because we understand that fundamentally, we are going to use this recursive formulation because we have understood the underlying recursive structure of the problem, if we are given a bigger; if we have to implement a bitonic sort algorithm, what we will do is; we take the whole input, divide it into 2 parts, these are entire input divide the input into 2 parts, for the first part you apply the bitonic sort algorithm with the direction ascending.

So, the values will be increasing so, the values increase, for the next part you again apply the bitonic sort algorithm with the direction reversed, so the values would actually be decreasing downwards, so here you have an ascending sequence followed by a descending sequence that

gives you the bitonic sequence, so this is essentially your step 1 and then you have step 2 which is essentially the bitonic merge.

Because already we have discussed, it expects ascending followed by descending and it is going to provide a fully ascending or descending output depending on what is the requirement. Now, we have also discussed that the step 1 is nothing but a generalization of step 2, I can keep on applying step 2 recursively on sub structures to realize step 1, which means it is fundamental to understand the bitonic merge problem.

And all we need to do is; we will be recursively calling the bitonic sort on smaller instances and finally, the bitonic merge function is the function which is going to do the job for us because we have already identified I am just repeating here that step 2 or the merge is going to be the fundamental thing effectively that is doing the sorting for you on the smaller instances and just flowing it out on the larger values.

So, if we look into bitonic merge in the general case so, you have given a low and a high essentially, the starting point and the ending point of values and you have given the direction, and you are asking me to do the merge, so what is really going to happen? So, now to better understand this, let us first look into the merge sorting network, as you can see this is the merge step.

The first thing we do is a sequence of comparisons how many; half of the size of the inputs, there is a number of comparisons we are doing, what is the stride of the comparison; again half of the size of the input, so at that size stride, I am doing 4 comparisons here and that is essentially followed by what; again a smaller instance of bitonic merger and the smaller instance of bitonic merge here, now this is important.

Let us understand again, the recursion present inside the bitonic merge step, so this is very; this is really something we need to understand. First point is we understood that step 2 is the most fundamental thing, step 1 is nothing but a recursive application of step 2 because what is happening in step 2 is; you are considering as input a bitonic sequence and then you are providing as output and the other sorted sequence.

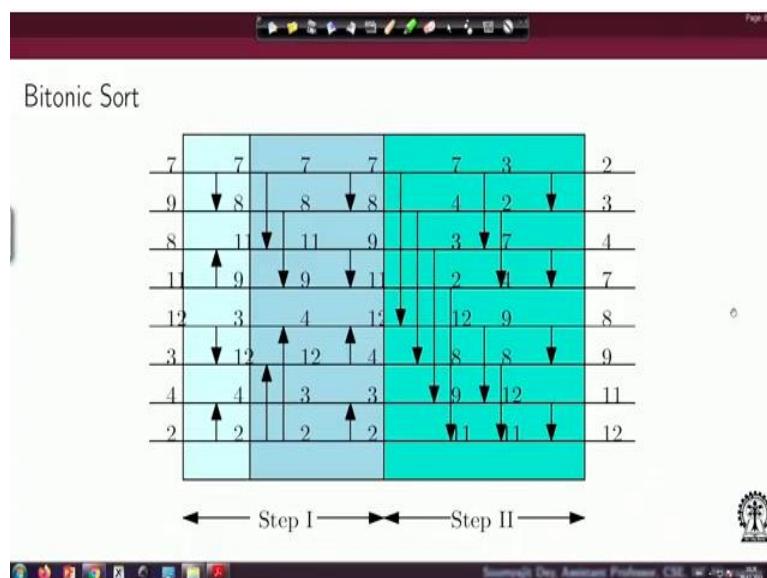
And when we apply step 2 on smaller instances and combine them that is essentially giving me step 1. So, with this understanding we will end this lecture and discuss the details of step 2 in the next lecture, thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture – 32
Optimizing Reduction Kernels(Contd.)

Hi, in the last lecture on bitonic sort we have been discussing about the recursive structure.

(Refer Slide Time: 00:30)



So what we identified was that step 1 in the bitonic sort is nothing but application of smaller instances of step 2 and then combining them to create the step 1 and the next thing was we are trying to figure out what is the step 2? And if you just recall from our further discussions.

(Refer Slide Time: 00:53)

```

//Step I
void bitonicSort(int lo, int n, boolean dir){
    if (n>1)
    {
        int m=n/2;
        bitonicSort(lo, m, ASCENDING);
        bitonicSort(lo+m, m, DESCENDING);
        bitonicMerge(lo, n, dir);
    }
}

```

Page 8 / 10



[Comparing Previous Slides](#) Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

We figured that the bitonic sort algorithm can be broken into 2 parts. So you apply bitonic sort on the smaller part. You break the total input into 2 parts apply bitonic ascending followed by bitonic sort descending on the first part and the second part respectively to generate a bitonic sequence and then apply the bitonic merge which is essentially your definition of step 2. So this first two steps generate the bitonic sequence then you apply bitonic merge that is the step 2.

(Refer Slide Time: 01:21)

```

//Comparator
void compare(int i, int j, boolean dir){
    if (dir==a[i]>a[j])
        exchange(i, j);
}

//Step II
void bitonicMerge(int lo, int n, boolean dir){
    if (n>1){
        int m=n/2;
        for (int i=lo; i<lo+m; i++)
            compare(i, i+m, dir);
        bitonicMerge(lo, m, dir);
        bitonicMerge(lo+m, m, dir);
    }
}

```

Page 9 / 10



[Comparing Previous Slides](#) Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And so the point we are trying to make now is that, okay, what is step 2 or what is the fundamental behind the bitonic merge process? Now when we went back to our example here we figured that in the bitonic merge essentially what we have is a sequence of comparisons as we

can see. So if the input size is 8 we are applying 4 comparisons here 1 stripes of size 4; so half of the size.

So this sequence of comparisons are followed by smaller instances of bitonic merge that is the important thing. Now if you look into these parts, so we have a sequence of 4 comparisons. Then a set of steps. Now if you look into these parts, I can say that the overall thing is a bitonic merge of size 8 and these two are nothing but bitonic merge of size 4; so from this recursive structure.

We can say that the merge step can again be written as a sequence of comparisons followed by 2 calls to bitonic merge on a reduced set. With this understanding if we look into the program; here the pseudocode. So you have this botanic merge with a low and a high and you are given the direction essentially now what will happen is you will make those sequence of comparisons then make the two calls to be precise, as we saw for this part and this part you are making two calls.

But the calls are going to happen on two separate parts on the half sized inputs. So from lo to $m = n/2$, the $high/2$, and then for the rest half. Important is to understand the merge process. The direction is same. So in these three calls in all the comparisons and in all the merge course the direction remains same. Again I will repeat in all the comparisons and in the merge process the direction remains same. If you look in here that is true. The direction is same as the desired merge process direction of the outer call.

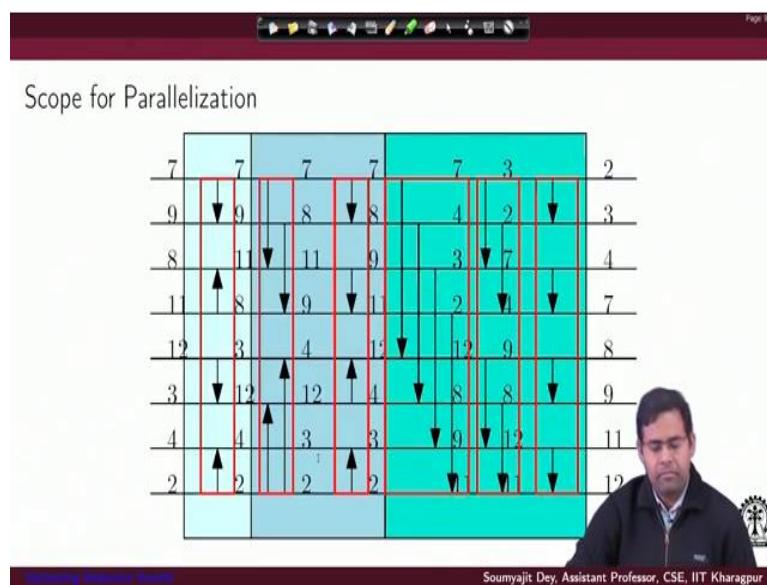
So here the outer call is expecting a bitonic sequences input and is going to provide an ascending output. So for that we make a call to merge of size 8 with ascending output reduction and here we do the comparisons in that direction and for the sub calls to the smaller merge processes again the direction does not change. So this is the most important thing to identify with respect to step 2 that I can simply write the merge process recursively like this.

All we need to do we will have first this set of comparisons. The comparisons are happening with a half size stripe. So from the start value, you make comparisons up to the midpoint, $lo + m$; there is a midpoint. As you can see from the start you are making comparisons 1, 2, 3, 4 up to the

midpoint. Each of the comparisons are considering this data with a data at the stripe size which is half of the overall width of the data.

So you are comparing the i^{th} value with the $i + m^{\text{th}}$ value you know this m is the stripe size and the stripe size is again $= n/2$ so half of the data width as I have been saying. So I hope this is now clear to you that we have a sequence of comparisons here and this comparisons will have their stride as defined and then after the comparisons we just make calls to consecutive bitonic merge. So once we execute this overall program we have the bitonic sort done.

(Refer Slide Time: 05:36)



But now the question is what about the CUDA implementation? So first thing we have to figure out is, okay, let us have a look what are the parallelizations. So as we can see the sorting network provides me the view that with respect to at the same time point what are the operations happening? So these are the operations that are happening in concurrently and here again these are the operations which have no dependency and I can make them happen concurrently.

So in red, we are trying to figure out which are the concurrent operations that we have and as we can see there are lot of concurrencies present inside each stage. So first I have step 1, inside step 1 I have 2 parts. So first I have this single comparator and then I have this idea of again applying the step 2 in a larger context and then again applying the bigger merge process here. So I hope this is clear that why even inside step 1 I am using 2 different colors.

So the first part essentially tells me applying step 2 in the smallest context and then here you are applying 2 instances of step 2 in a relatively large context and then you are applying step 2 in the bigger possible context here. So looking at the parallelization as we can see I can define the per thread activity like this. So I can run these parallel threads and each thread in this part of the computation of step 1 they can perform this comparisons in parallel.

So I would require 4 threads here half the number of threads as the number of data points. So each thread can perform 2 loads and then do the comparisons put the value in the shared memory. And then the threads can just do a sync threads here, progress here effectively do this comparisons and then progress and then progress so on and so forth.

(Refer Slide Time: 07:51)

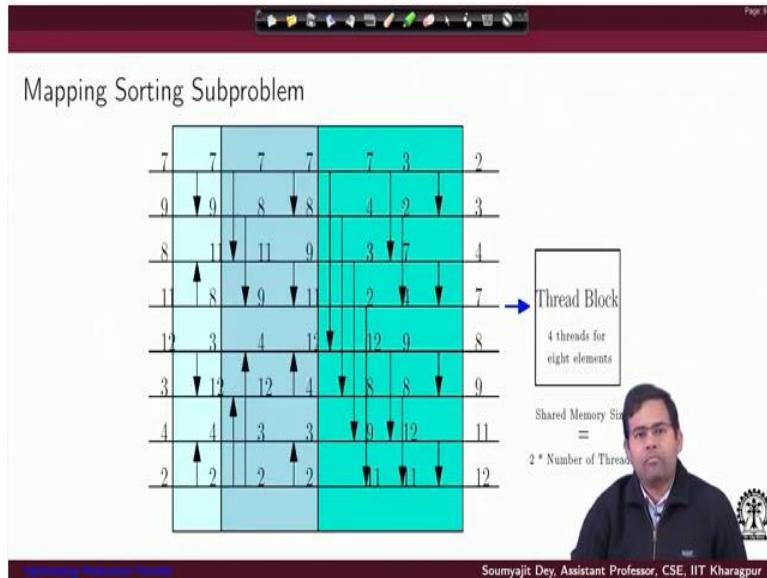
Formulate Parallel Solution

- ▶ Associate every cuda thread block with a sorting subproblem.
- ▶ Merge results from each SM to solve the original sorting problem.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

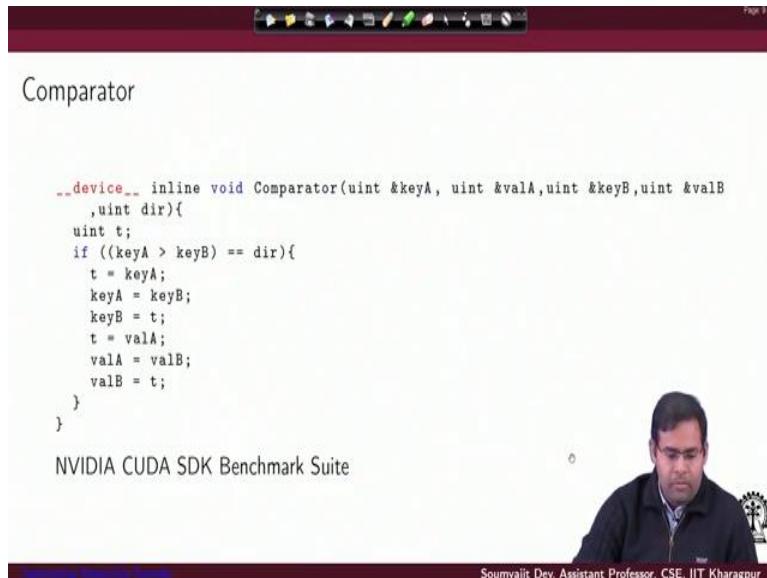
So overall we can summarize like this that associate every cuda thread block with a sorting subproblem. So essentially if we have a significantly large sized array to sort you define cuda thread blocks for each of the sub arrays and then you apply the merge on the results for the different thread blocks that you derive. So at the end you have to actually merge results from each of the SMs and solve the original sorting problem.

(Refer Slide Time: 08:25)



So fundamentally we need to also identify that what is the size of the thread block? As we discussed that since 4 threads are going to do the job in this case. So if I have an 8 element input I would require half the number of threads okay but what is the shared memory size is going to be same as the input size here for each block. So it is going to be twice of the number of threads.

(Refer Slide Time: 08:52)



Now these are simple codes that we have taken from NVIDIA CUDA SDK benchmarks and so first thing we highlight here is the comparators code. So what we have is 2 inputs now we are writing a general situation here. So we have 2 inputs with key value pairs. We have two inputs with key value pairs and we also provide a direction and then for the inputs we compare the key and then we compare it with the direction.

Now as you can see that the idea of this program is similar to the C program comparator that we had. So essentially we are checking it with the direction if $i > j$ and there is a 2 case. Then we do the exchange so we are considering downward arrow sorting networks by this example here and of course if the direction is otherwise then it will represent the upward arrow sorting network. So as we derived earlier considering dir as 1 I have this, considering dir as 0 I have the reverse.

So we compared the key and then we compared it which we just compare the result of this comparison I mean 1 or 0 with the direction and accordingly we decide to swap the key and swap the value. So this is just an extension of the original comparator circuit considering key value pairs.

(Refer Slide Time: 10:41)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Sort in Shared Memory". Below the title is a code snippet in CUDA C:

```
__global__ void bitonicSortShared1(uint *d_DstKey, uint *d_DstVal, uint *
    d_SrcKey, uint *d_SrcVal){
    //Shared memory storage for current subarray
    __shared__ uint s_key[SHARED_SIZE];
    __shared__ uint s_val[SHARED_SIZE];

    //Offset to the beginning of subarray and load data
    d_SrcKey+=blockIdx.x*SHARED_SIZE+threadIdx.x;
    d_SrcVal+=blockIdx.x*SHARED_SIZE+threadIdx.x;
    d_DstKey+=blockIdx.x*SHARED_SIZE+threadIdx.x;
    d_DstVal+=blockIdx.x*SHARED_SIZE+threadIdx.x;
```

In the bottom right corner of the slide area, there is a video feed of a person, identified as Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur. The video feed has a small circular logo in the top right corner.

So we are going to do the sorting based on the key values. Now we provide the implementation of bitonic sort on shared memory. So we will have each thread loading 2 values to a shared memory and proceeding through the different stages of the sort. So we define 2 of these shared memory segments one for the storing the key values and one for storing the one for storing the keys and one for storing the values here.

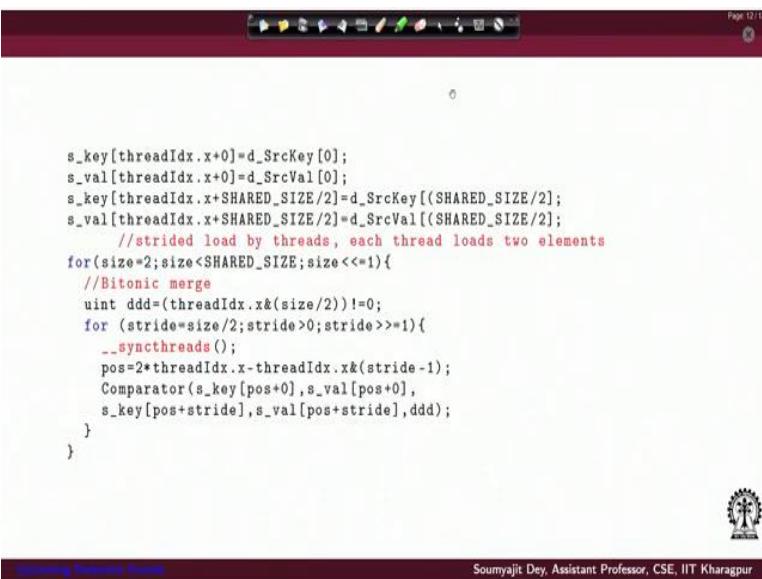
And then we have to figure out what is the per thread activity? That means we need to figure out okay each thread is going to store load and store 2 key value pairs. So they are going to work in which area of the memory. So for the 2 threads the first thing you do is you calculate the offset to

the beginning of the subarray that means inside which of these blocks a thread is going to do the load activity, it has to identify that.

So what we do is we check the block ID. So whatever is the block ID you I mean we are considering x dimensional and one dimensional data here so you just multiply the block ID in the x direction with the shared size and then if you do a plus thread Idx dot x you get the offset that means inside the shared memory corresponding to this thread block what is the offsetted location in the shared memory where this thread is going to load the key, load the value and also the other key and the other value.

We just call them as destination and source key values. So this part of the code actually performs the computation of that offset. So these are block and you just compute the offsets of the key value pairs. Because you are considering for each block you have 2 shared memories so every thread is going to load 2 of these keys and 2 of these values and considering only 1 dimensional data here.

(Refer Slide Time: 13:06)



The screenshot shows a presentation slide with a dark header bar containing icons for back, forward, search, and other controls. The main content area displays a block of C++ code. The code is a bitonic merge algorithm, likely for GPU memory access. It includes comments explaining the strided load by threads and the bitonic merge process. The code uses variables like `s_key`, `s_val`, `d_SrcKey`, and `d_SrcVal`. The slide has a footer with the text "Copyright Material - Review Only" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

```
s_key[threadIdx.x+0]=d_SrcKey[0];
s_val[threadIdx.x+0]=d_SrcVal[0];
s_key[threadIdx.x+SHARED_SIZE/2]=d_SrcKey[(SHARED_SIZE/2)];
s_val[threadIdx.x+SHARED_SIZE/2]=d_SrcVal[(SHARED_SIZE/2)];
    //strided load by threads, each thread loads two elements
for(size=2;size<SHARED_SIZE;size<<=1){
    //Bitonic merge
    uint ddd=(threadIdx.x&(size/2))!=0;
    for (stride=size/2;stride>0;stride>>=1){
        __syncthreads();
        pos=2*threadIdx.x-threadIdx.x&(stride-1);
        Comparator(s_key[pos+0],s_val[pos+0],
                   s_key[pos+stride],s_val[pos+stride],ddd);
    }
}
```

So once we have figured out that this is the location from which we are considering the threads to load the values into the shared memory. So now the pointers have been shifted to the offsets. So now you can just access the location with this as the base shifted by 0 for the source key and

the source value and you write it back to the shared memories location only with the thread Idx offset.

I hope this is clear as we can remember that each shared memory is being defined in a power block basis each shared memory segment that we have here is there in a power block basis. So first each thread figured out in the global memory what is the offset or from where it is going to load the data? Is going to load 2 of the data points and then it figured out using the thread Id based offset into the shared memory that where the locations they are going to write.

So this is my shared memory from the global memory I have figured out what is the data I will write and then in the shared memory for this block just by using the thread Idx based offset and figuring out per thread blocks. This is the location where I am going to write the key and so this is the shared memory for keys and similarly I have the shared memory for values. So in that I will also figure out where to write the value and so on and so forth.

Now as we see that every thread is going to route 2 loads to the shared memory both for threads and values. So the second load has to happen it has tried of with an offset of shared size/2. Because it has first figured out its location the best location where it is going to load using thread Idx inside the shared memory block and then you add that thread Idx with half size of the shared memory block to figure out what is the other location of where also is going to load the key.

And that it is guessed from this d source key I mean similarly how do I get the load data address from the global memory? Well I have already shifted this to this d source key is now pointing to the location from where you have to just load the first data point. So that is why you can access it by index 0. So you can just put the index as shared size/2 and that would give me the other location in the global memory from where I am going to load the second data point.

So just so this is the second location from where I am going to load the other data point. So first thing we did was we calculated the offsets that means what is the location in the shared memory where I am going to load the data and then we just loaded the keys and the values at the respective locations with the offsets as provided.

(Refer Slide Time: 17:06)

```
s_key[threadIdx.x+0]=d_srcKey[0];
s_val[threadIdx.x+0]=d_srcVal[0];
s_key[threadIdx.x+SHARED_SIZE/2]=d_srcKey[(SHARED_SIZE/2)];
s_val[threadIdx.x+SHARED_SIZE/2]=d_srcVal[(SHARED_SIZE/2)];
//stride load by threads, each thread loads two elements
for(size=2;size<SHARED_SIZE;size<<=1){
    //Bitonic merge
    int ddd=(threadIdx.x&(size/2))!=0;
    for(stride<size/2;stride>>0;stride>>=1){
        __syncthreads();
        pos=2*threadIdx.x-threadIdx.x&(stride-1);
        Comparator(s_key[pos+0],s_val[pos+0],
                  s_key[pos+stride],s_val[pos+stride],ddd);
    }
}
```

Now we come to the second phase that now I have just figured out that where the loads are going to happen in the shared memory by the thread and each thread is just doing that 2 loads from the global memory to the shared memory. The next thing that is supposed to happen is we are going to enter the step 1. We are going to enter into the step 1 in which we are going to build in parallel the bitonic sequence followed by.

In step 2 we are going to figure out how this bitonic sequence is going to be arranged in the completely ascending or descending order. As we have seen from our earlier algorithm that well all that we will be needing to do is we will first have the sequence in the step 1 and step 2. So here first we show the part for the bitonic merge here that is happening for the different segments that are going on.

Because we have already identified that even for step 1 we just need to carry out the bitonic merge process at the different segments that are there of the program. Sorry so before carrying out any of this comparisons we need to figure out that what should be that direction of the comparison and of course what should be the positions where the comparison operation has to be performed.

So for that we introduce 2 variables here this variable ddd is kind of computing what is the direction? And this variable pos is trying to figure out what is the position, where the comparison has to be performed. Now if you have a loop into the sequence of operations that we do here. So let us try and figure out that how to assign this comparisons in as a per thread activity.

Because till now whatever we have done is to figure out how each of these 4 threads are going to load the values. So the next phase is to figure out how each thread is going to be orchestrating the comparisons that are to be done. So what we do is okay for performing the comparisons.

We let thread 1 do the first comparison and then we let thread 2 to do the second comparison and then we let thread 3 do the third comparison and thread 4 does the other comparison or maybe I can just use the actual Ids of the threads sorry which would be 0, 1, 2, 3. So these are the per thread activities and then coming to the next part what we do is?

In the next phase of operations so if you may recall this entirely is my step 1 inside step 1 I am essentially executing step 2 first with I mean one-fourth of the input size and then I am executing step 2 again with half of the input size. So here this is my definition of the per thread activity and then I have this is the part that would be done by thread 0 and then this is what will be done by thread 1 and we will have the following to done by thread 2 and 3 respectively so on and so forth.

So as you can see what is happening is in the first part we have threads doing the comparisons at a specific location and the next thread is doing the comparison at another location with the direction continuously flipping. Then when we come to the next part what we have is the threads again doing the comparisons at a bigger stride for 2 threads the directions do not change and again for 2 threads the direction do not change.

So somehow we need to figure out that how this effect can be brought in. So for that if you check the way we are actually computing the direction is basically a function of the thread id and the size value. So we put a for loop here which is trying to figure out that okay in which stage we are operating of step 1.

So let me just reproduce the picture here we are just elaborating on step 1 here and as we see that step 1 has 2 parts. So this is the full picture of step 1. Now since I am working with an 8 size input in step 1 we have already figured out that what we do is here we run step 2 on half the size. So I can just write this is step 2 on half the size here and similarly step 2 on half the size here and then we run step 2 on one-fourth of the size 4 times.

So essentially I can say the number of times I am running step 2 is going to be controlled by the outer loop and inside each iteration of the outer loop I am doing the number of comparisons which is controlled by the inner loop I hope this is fine. So the number again I will repeat so the number of phases I would say of applying step 2 in different stride size is being controlled by the outer loop.

For example here you are executing step 1 on an input of size 8. For that you have to execute step 2 on half the sized input twice and step 4 step 2 on one-fourth the size of input 4 times. The fact that you are going to use step 2 in 2 different configurations is being reflected by the outer loop. So the outer loop is controlling in what configuration I am going to apply step 2.

First when I run the outer loop it is setting the size variable at 2 and it is saying that okay you are going to apply step 2 with the comparisons done between 2 consecutive elements okay and that the fact that these comparisons will be done. So essentially you are applying step 2/4 that means you are going to apply step 2 on input of size/4 4 times essentially we will be doing 4 number of comparisons that is going to be controlled by the inner loop.

Look at the next iteration of the outer loop you are increasing size/2. Once you increase size/2 what is happening is? So again I will just repeat once you increase size/2 what is happening is? You are reconfiguring your way of running step 2 and you are saying that okay now I will not be comparing I mean okay I am going to run step 2 again at configuration where it is input size/2 and that is realized again by the inner loop with suitable choice of size and direction.

So if I just loop that how the values of size and stride are changing so initially when I start well I have size = 2 and stride as 1 and so with that configuration since stride is okay by the way the

stride is computed here and that comes out to be 1 because size = 2 and all we are doing is we are just running the inner loop and in each iteration of the inner loop we are actually going to increase this value of stripe.

So starting from the initial value so what we have here is we start with sizes 2. So when we figure out that okay in that case stride would be 1. Now since stride would be 1 i would get in here and here I am going to execute the comparison operation by all the 4 threads. So I will execute 4 different comparisons. So they are all done in 1 shot by the 4 threads in parallel and then I am trying to decrease the size.

So that would mean since the integer variable size stride 2 was already 2 and stride sorry stride was already 1 if I decrease by having it with 2 it goes to 0 and the loop does not execute anymore. So in that way I have this 1 iteration of the loop and inside this 1 iteration of the loop I have all the threads executing 1 comparison in that way I get 2 of the 4 of the comparisons done in parallel and the comparisons are done on adjacent values which is dictated by the stride variable and the size value was set to be 2.

So now if the next iteration when I come back for the outer loop I would have size as increased by this factor of 2 because of this left shift and but still it is smaller than the shared size so I come to the other part of the step 1. And here I am now going to execute these 2 different parts of step 2, 1 for have the input size 1 here and the other here. Let us look at how this is done getting realized.

So now I have said the size value as 4. Since size value has been said as 4 I am now going to execute with stride at 2 because stride is size/2 and then again the loop will again execute with a stride value of 1 and then the stride will go to 0 and it wont execute anymore. So when I execute with size as 4 and stride as 2, let us see what has happened okay. Again I will get into this loop and all the threads will execute comparison.

But now the comparisons are going to happen with the stride of 2 that is why you get these 4 comparisons happening at a larger stride size I mean part on my picture and maybe we look at

the original picture. So I am talking about these 4 comparisons here. As you can see they are happening with a stride size of 2 yeah. And then in the next iteration of this loop well I will again enter into the next iteration.

Because I will half the stride let it become 1 which is still greater than 0. So now I will again perform 4 of this comparisons on address and data points and their directions would be like this. So this loop ran with for 1 iteration in the first size essentially it executed 4 different comparisons. Essentially all of them were the atomic instances of step 2 running on one-fourth the size of input that is the step 2/4 that I am trying to say.

So essentially and that covers all the 8 inputs in 4 pairs. Here again I am running it I am just repeating. So step 2/2 sized that means step 2 on the input size of 4 that is 2 of the instances and the way it is being realized is that well we set the size variable again now as 4 and with the size variable as 4 we get 2 possible stride values that is why this inner loop is now going to iterate twice.

In the first iteration is going to perform 4 comparisons, in the next iteration again 4 comparisons, in the next iteration in the first iteration the comparisons are at strides of 2, in the next iteration the comparisons are going to happen at strides of 1. So in that way this code is actually achieving the overall idea of performing the step stage 1 or step 1 of the bitonic sort. Now observe the different ways the directions have changed.

So as you can see here the direction was we have repeated I am just repeating this direction was continuously switching. In the next stage the direction remains same for the first 2 threads and then switched for the next 2 threads and that pattern continued here. To realize this switching activity of the direction we do a bitwise end based realization of this value using thread Id and size.

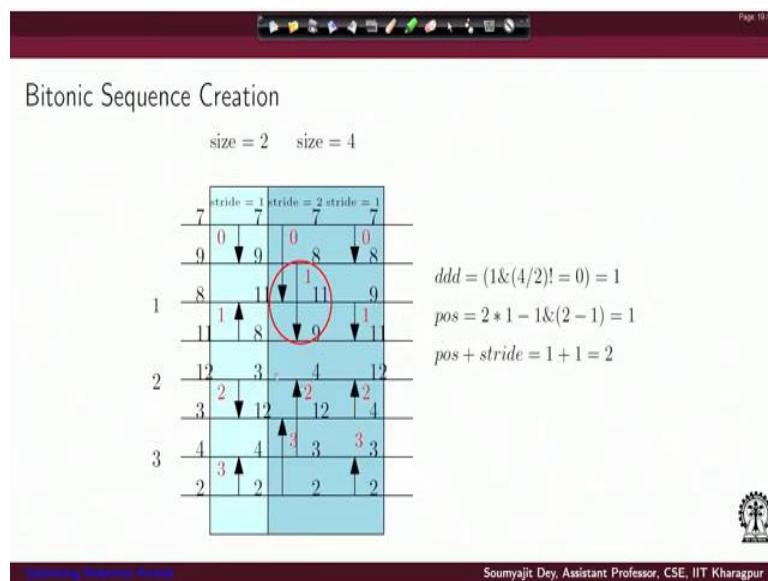
If you can check these values there you will see that how this expression is actually able to I mean reproduce this idea of direction switch where when I said the size as 2 the direction switches very frequently. When I said the size as 4 the direction switches up for 2 threads I mean

the first two threads have the same direction and the next two threads have the other direction. That is how the direction switch activity happens.

And then for the position again I have this expression which actually takes care of identifying for each thread Id what is the starting position? Because for thread Id 1, this has to be the starting position for thread Id; this followed by there is a switch in the starting position, yeah, so this is also something interesting just the comparison positions for each of the threads the use for thread Id 0.

For the next thread the starting positions actually change. These are the starting positions for the next thread and for the other threads for the thread with Id 2; again it is this and for the other thread is this. So these are the different starting positions that we have. Now to achieve this sequence of positions as a function of the thread IDs we use this access expression. So I mean you will need to check how this actually reflects those switches of position by the thread Ids.

(Refer Slide Time: 35:13)



Now this is also an example through which you can identify how everything is going on. So here, we have actually noted down in the as a par thread activity that thread Id 0 is performing, this operations thread Id 1 is performing, this comparisons thread Id 2 is performing, this comparisons thread Id 3 is again performing this comparison so on and so forth. And we have also done some example computations of triple d position, position + stride etc.

(Refer Slide Time: 36:03)

```
//sort in opposite directions odd/even block ids
uint ddd = (blockIdx.x + 1) & 1;
for(stride=SHARED_SIZE/2;stride>0;stride>>1){
    __syncthreads();
    pos=2*threadIdx.x-threadIdx.x&(stride - 1);
    Comparator(s_key[pos+0],s_val[pos+0],
               s_key[pos+stride],s_val[pos+stride],ddd);
}
__syncthreads();
d_DstKey [0]=s_key[threadIdx.x+0];
d_DstVal [0]=s_val[threadIdx.x+0];
d_DstKey [SHARED_SIZE/2]=s_key[threadIdx.x+SHARED_SIZE/2];
d_DstVal [SHARED_SIZE/2]=s_val[threadIdx.x+SHARED_SIZE/2];
}
```

Yes, now we come to the other part of the code where we are kind of trying to figure out what is the step 2 here. So if you see in the step 2, you have this uint which is basically the direction. Now in step 2 first of all the important thing is we already have figured this out that the direction does not really change. So you compute that direction and it just remains the same.

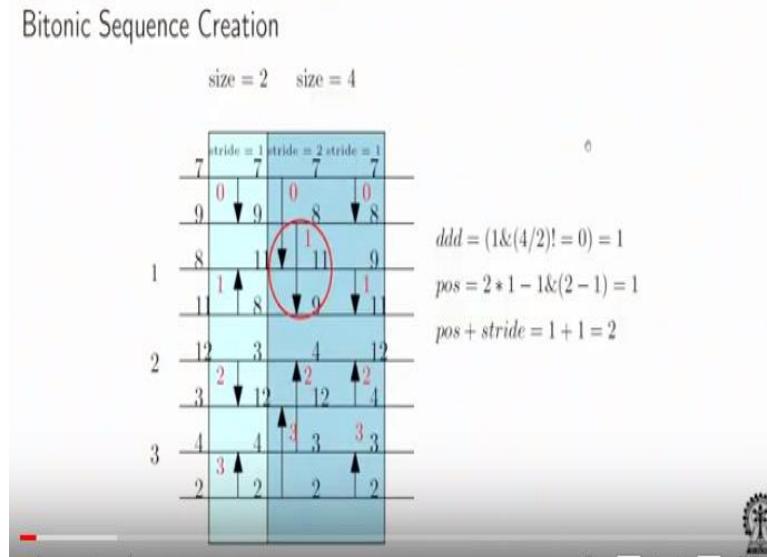
So and then the thing that keeps on changing is at what stride value you keep on performing the comparisons. So maybe we will take this up in the next lecture and with this we will end the current lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture – 33
Optimizing Reduction Kernels(Contd.)

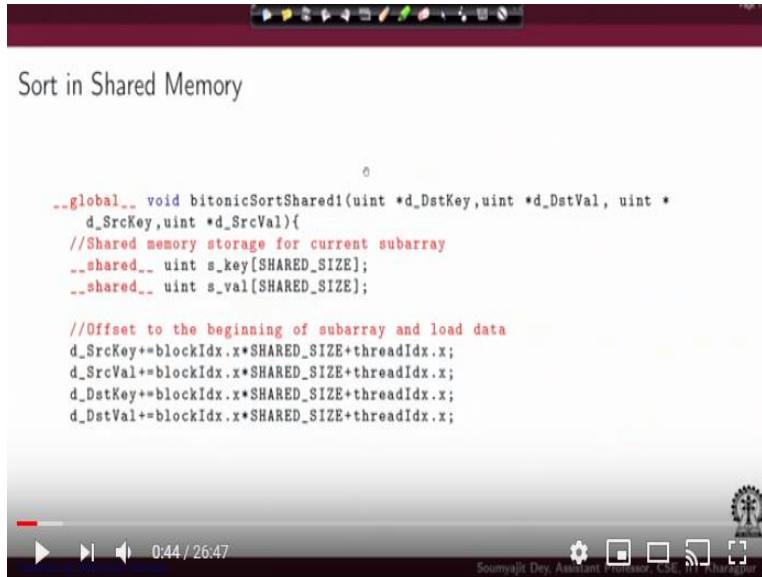
Hi, welcome to the lecture series on GPU architectures and programming. So if you remember in the last lecture we have been discussing about Bitonic sort.

(Refer Slide Time: 00:33)



And our discussions progressed up to the part where we are generating the Bitonic sequence.

(Refer Slide Time: 00:42)



The screenshot shows a video player interface with a dark theme. The title bar says "Sort in Shared Memory". The main area contains CUDA C code for a bitonic sort algorithm. The code uses shared memory to store subarrays and performs strided loads and bitonic merges. The video player has a progress bar at 0:44 / 26:47 and a status bar at the bottom.

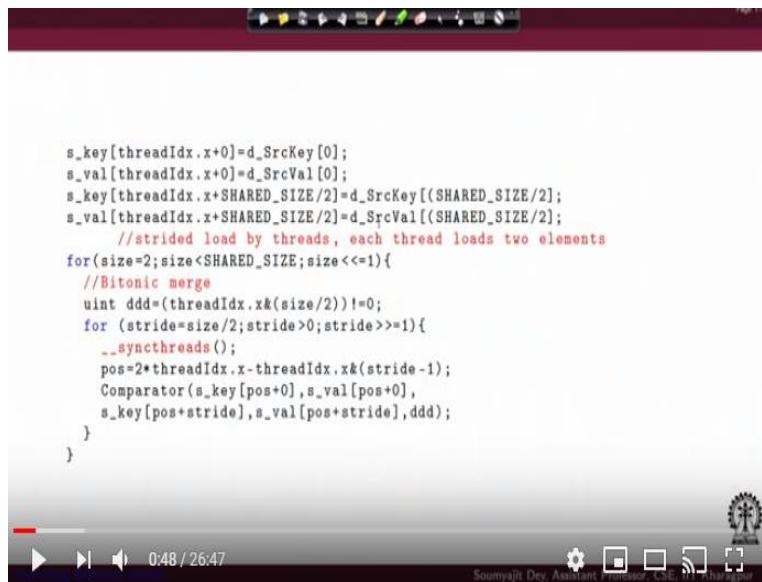
```
--global__ void bitonicSortShared1(uint *d_DstKey,uint *d_DstVal, uint *
d_SrcKey,uint *d_SrcVal){
//Shared memory storage for current subarray
__shared__ uint s_key[SHARED_SIZE];
__shared__ uint s_val[SHARED_SIZE];

//Offset to the beginning of subarray and load data
d_SrcKey+=blockIdx.x*SHARED_SIZE+threadIdx.x;
d_SrcVal+=blockIdx.x*SHARED_SIZE+threadIdx.x;
d_DstKey+=blockIdx.x*SHARED_SIZE+threadIdx.x;
d_DstVal+=blockIdx.x*SHARED_SIZE+threadIdx.x;

s_key[threadIdx.x+0]=d_SrcKey[0];
s_val[threadIdx.x+0]=d_SrcVal[0];
s_key[threadIdx.x+SHARED_SIZE/2]=d_SrcKey[(SHARED_SIZE/2)];
s_val[threadIdx.x+SHARED_SIZE/2]=d_SrcVal[(SHARED_SIZE/2];
//strided load by threads, each thread loads two elements
for(size=2;size<SHARED_SIZE;size<<=1){
//Bitonic merge
uint ddd=(threadIdx.x&(size/2))!=0;
for (stride=size/2;stride>0;stride>>=1){
__syncthreads();
pos=2*threadIdx.x-threadIdx.x&(stride-1);
Comparator(s_key[pos+0],s_val[pos+0],
s_key[pos+stride],s_val[pos+stride],ddd);
}
}
```

And just a small recall; so this was the CUDA code we were showing for the Bitonic sequence generation.

(Refer Slide Time: 00:47)

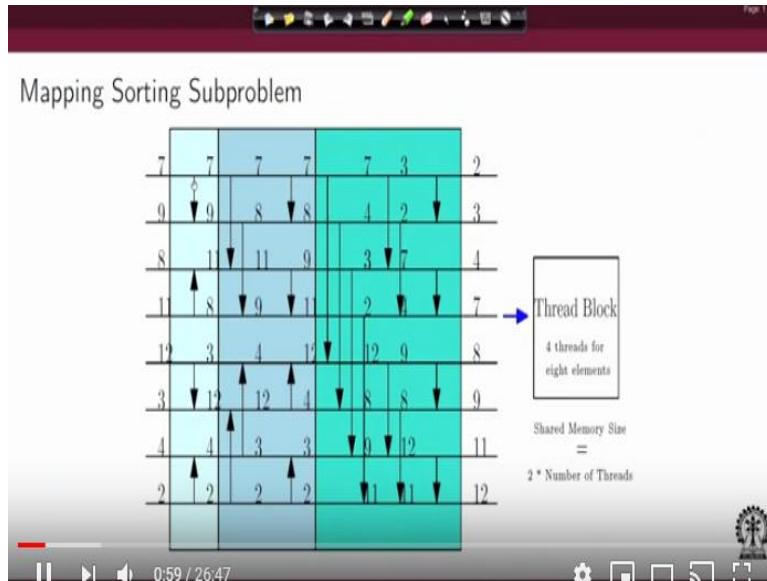


The screenshot shows a video player interface with a dark theme. The title bar is not visible. The main area continues the CUDA C code from the previous slide. It shows the bitonic merge loop where threads compare and swap values based on their stride. The video player has a progress bar at 0:48 / 26:47 and a status bar at the bottom.

```
s_key[threadIdx.x+0]=d_SrcKey[0];
s_val[threadIdx.x+0]=d_SrcVal[0];
s_key[threadIdx.x+SHARED_SIZE/2]=d_SrcKey[(SHARED_SIZE/2];
s_val[threadIdx.x+SHARED_SIZE/2]=d_SrcVal[(SHARED_SIZE/2];
//strided load by threads, each thread loads two elements
for(size=2;size<SHARED_SIZE;size<<=1){
//Bitonic merge
uint ddd=(threadIdx.x&(size/2))!=0;
for (stride=size/2;stride>0;stride>>=1){
__syncthreads();
pos=2*threadIdx.x-threadIdx.x&(stride-1);
Comparator(s_key[pos+0],s_val[pos+0],
s_key[pos+stride],s_val[pos+stride],ddd);
}
}
```

So each of the threads were responsible for bringing two values into the shared memory. Then each of the threads,

(Refer Slide Time: 00:57)



as we can see from the previous picture here that each of the threads are tasked with the following operations. Like if I look at the thread id 0 then if I try to figure out that what are the values on which that threads id is working? And so we have a nice example here on the 1st part that threads id 0 is bringing two values and in the 1st phase of operations thread id 0 is doing a comparison here, whereas thread id 1 is doing a comparison here. Thread id 2 is doing the next comparison like that.

All of these are happening in this constant I mean in this with the variables of size =2 and stride =1. As you can see the stride is 1, here, for the comparisons and the size =2 denotes that essentially we are doing a bit unique Merge which size 2 here. I mean just between two elements there that would be the first phase of the Bitonic sequence generation creation.

In the second phase again we will have size =4 and we will just apply the same logic? But now we will try to do it with, since we are doing it size =4 then well have two strides; stride=2 and stride=1 and in this strides threads id 0 would be tasked with doing these two comparison; these first comparison here with stride=2.

And then when the iteration of the inner loop goes to the next iteration then thread id 0 would again we will be doing another comparison with a reduced stride of 1. Because in the inner loop, as we have discussed earlier, the stride is decreasing by half. It is starting from size by 2. So we

start from stride 2 and then we moved to stride=1 and similarly when we started with size =2 the stride was already half to 1 and there was no further progress?

So the outer loop is controlling the size and the inner loop is reducing the stride starting from size by 2. And in that way I have different phases of the Bitonic sequence creation as we have discussed that recursively; these are all basically Bitnoic merge operations at different levels.

So I mean, just as an example as we can see, that here we have a small Bitonic sequence getting generated 7 increases to 8 and 11 increases to 9 and then again this is being going to Bitonic merge stage here in the step 2. So in that way we have the above code here which is realizing the Bitonic sequence generation part using these two cascade of loops. So overall I mean, off course the lecture slides will be with you, if you take a closer look you will find as we have discussed many times earlier each threads is loading two fellows into the shared memory at an offset of share size by 2. Then each thread gets into this loop cascade, then each thread in each stage of the loops. So the outer loop is controlling the iterations over size values, inner is controlling the number of strides for a specific size value and that defines for what is the par threads activity.

So for 0 id threads, this is the activity; for the threads with id 1, this is the activity for threads. With id 2 we have again marked activities. So the ids are marked here in red. As you can see similarly for thread with id 3 these are the activities and so on full so forth. Now if you look into the code, we also had these variables ddd and position which were actually trying to compute the direction of the comparison and the position at which the comparison would happen.

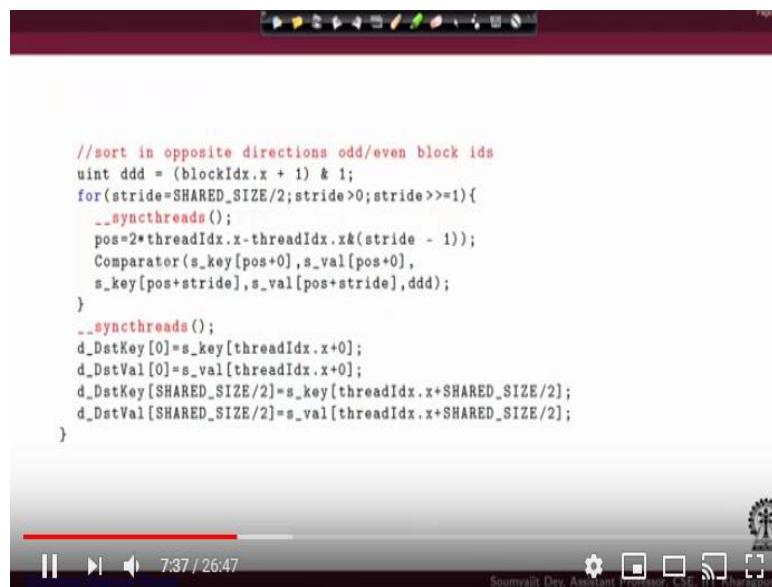
Just to do a check that they are really doing their job. If we take an example here for this specific threads with id 1 in the size = 4, stride =2 case that how is this thing really happening? You can see that a, this is how the direction, is being computed as downwards. Using that Bitwise and operation and similarly you can also compute the position and position plus stride locations here.

So just have a look this an example, we have tried to mark out for you. Now coming to the next part of the code assuming that we are okay with this sequence generation idea. Then the other part comes in where we are just trying to do the overall merge. So as we can say that the Merge

is a more fundamental thing but here what we have to do is, we have to do the merge over a large sized Bitonic sequence.

So the first thing is inside the merge phase, there will be no change of direction. Because now my goal is fixed whenever I am applying smaller versions of marge recursively inside the sequence generation phase, off course, there will be a change of direction as it is happening here. So some thread as you can see 0 is always computing, the comparison side is indication is down; for 1, the comparison direction changes; for 2, the direction again changes; for 3, there is no change in direction and these are controlled by these variables. But when we come to the overall merge stage it is a global objective. That you either want to do a completely ascending output or you want to compute a completely descending output.

(Refer Slide Time: 06:28)



So the direction is computed once and for all. And then you enter into the loop cascade for doing the different margins for doing the different sequence of comparison operations. Now if we just go back to our original picture here. So this is my merge stage. So in this merge stage, as you can see that, we have a validation that directions are all same. We are trying to do an ascending output basically to all the arrows are downward.

We want the Max to all flow down. Now what is happening is; we have to identify what is the parts thread activity. So for each threads there will be a downward comparison happening at

different stride values and at different data points. So one thing is common the stride value will increase by 2; as you can see a stride 4 comparison, stride 2 comparisons, stride 1 comparison.

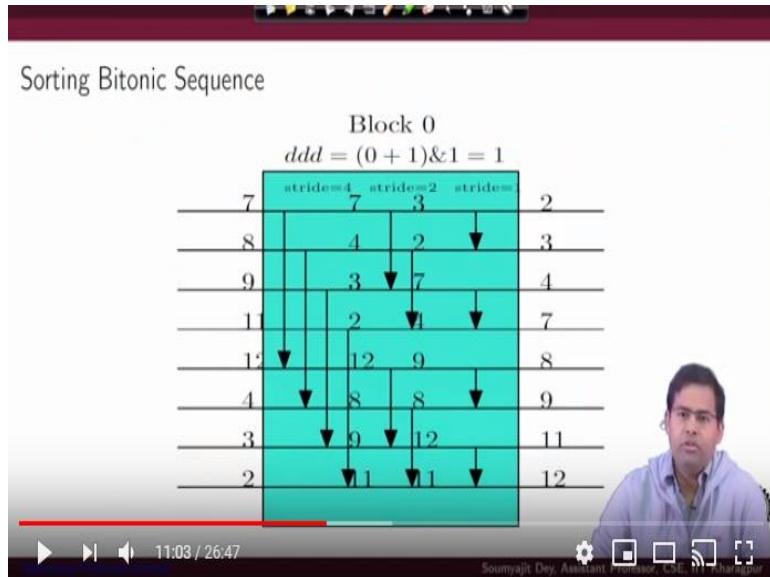
Similarly for each threads stride 4, stride 2 and strike 1 comparison. The issue is how do we distribute the different comparisons to be done as a par thread activity. So again that is being; this is the loop which is controlling these sequence of stride values for part for each threads. So the loop iterations for the threads are spread over the X axis in that picture. And here for each threads you are figuring out what is the position. Once you figured that, okay, I will do a comparison at this stride value for the thread.

The next thing you do is; you figure out what is the position of the comparison. Once you figure out the position of the comparison by this computation. The next thing is you just apply the comparator at that position with that stride and the direction that has been already pre decided. So with this loop the output will be red in the shared memory. Then what you have to do is; since each threads has been taxed with loading two values.

Now the threads will also write back to values to the global memory both the keys and the value. So essentially the position index computes the location for which the threads will be doing the comparisons like for thread 2, the locations would be here this and this like that. Right in that way for each of the threads, the positions would be computed with different strides; the comparisons would happen.

So thread 1 would have these three operations. Similarly, three comparisons for each of the threads. And finally when the loop iteration ends for all the threads, I would have these values in the shared memory which now will be returned back to values per thread in the global memory. So we have the example here like so if you look at here this is the Bitonic sequence creation. So this is a sequence 7,8,9,11 and then 12, 4,3,2; ascending and then descending.

(Refer Slide Time: 09:35)



Right now let us look at the merge operations here. So you have these; that earlier output as the input and then we are showing that first 4 block 0 that direction has been decided as ; so everything downwards and then for different stride values. We are just trying to show here. using the example, how the values change. So for thread 0 when these activities is done, 7 and 12 nothing will change.

Then it is looking here at 7 and 3. So 7 will come down and then again its just looking here and it will make 3 come down here and 2 go up, in that way with all the other values also being getting swap through the comparators, you have the final output at here as output of the sorting network. So that would be our overall discussion on Bitonic chart implemented as a CUDA program.

So this actually gives you a nice overview how to use a sorting network and try to create a parallel implementation of that and that will be useful for you to do sum; and you can try with some large sized arrays how to make these implementations and you can study things like how shared memory and other issues can be effecting the computation. I mean there are many interesting issues there.

Like for example how do I really configure the shared memory? Is it really affecting the computation speed? Should I compare the shared memory as a larger shared memory with a smaller element cache or a smaller shared element cache? What is really going to help me in this

case? So these are important questions you may ask yourself. You can try in a CUDA based system for your purpose.

(Refer Slide Time: 11:23)

All-Prefix-Sums

The **all-prefix-sums** operation takes a binary associative operator \oplus , and an array of n elements.
[a_0, a_1, \dots, a_{n-1}],
and returns the array
[$a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-1})$].

Example: If \oplus is addition, then the all-prefix-sums operation on the array
[3 1 7 0 4 1 6 3],
would return
[3 4 11 11 15 16 22 25].

Now, we will come to the other example that we have; which is on prefix sum. So to understand what is a prefix sum operation; essentially, it is a difference. The difference between reduction, I mean, parallel reduction that we did earlier and prefix sum is that earlier when we were doing parallel reduction you are given the sequence of values and all you are doing is; you are only interested in this value.

So given the operator, you are interested, assuming that just does simple addition; it can be any other associative operator, you are interested only in this expression. But now we are seeing that okay no I also want the other prefixes being operated like this. So essentially I am looking at an output which is a_0 followed by a_0 then assuming this addition could I am saying it can be any associative operation a_0 at 1 and at the last it can be $a_0 + a_1 + a_2$ like that; and finally the addition of all the numbers.

Let us assume that this is what I want instead of only computing the total sum. So in that way if this is the input in a prefix operation, it should return me this entire sequence. Earlier when were doing a parallel reduction as human as a simple addition operation, I was only interested in this

25 value. But now I am saying that no I want an output at a computer which is containing the sum of all the prefixes.

So in the i th location I have the sum of values from $a_0 + a_1 + a_2$ up to $+a_i$.

(Refer Slide Time: 13:07)

Inclusive and Exclusive scan

All-prefix-sums on an array of data is commonly known as *scan*.

- ▶ **Inclusive scan** - a scan of an array generates a new array where each element j is the sum of all elements up to and including j .
- ▶ **Exclusive scan** - a scan of an array generates a new array where each element j is the sum of all elements excluding j .

The exclusive scan operation takes a binary associative operator \oplus with identity I , and an array of n elements
[a_0, a_1, \dots, a_{n-1}],
and returns the array
[$I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})$].

Now the issue is how can these be accelerated? Can the usual technique that we applied for computing the normal parallel reduction; is it directly applicable here? What are the ways we accelerate and all that? Now one obvious question is here that we can have two possibilities. One is inclusive scan and Exclusive scan. So inclusive scan would mean that when you are producing the output array then at any point any j th location in the output array you are interested in the sum from a_0 up to a_j .

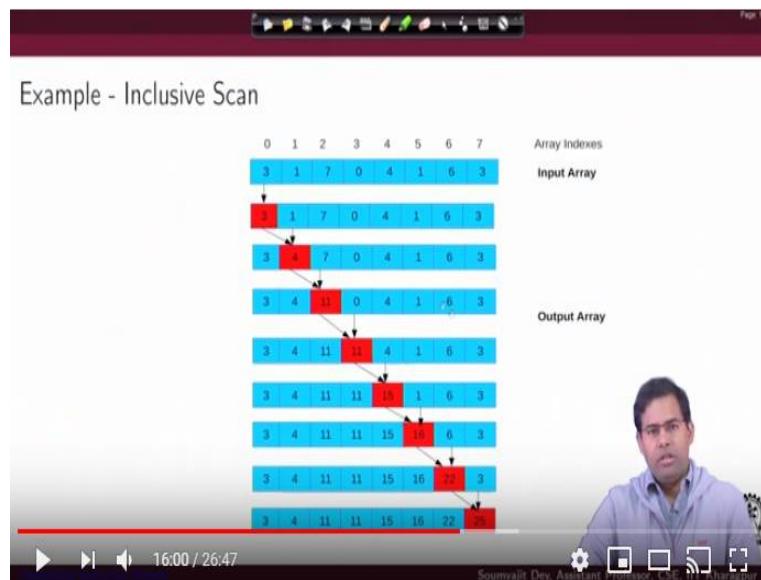
Whereas if I say it is an exclusive scan you are saying that okay you are interested in $a_0 + a_1$ like that up to a_{j-1} . So essentially we are seeing that okay whether I will include the element in that location in the sum or whether I should exclude it and I will just put there the value which is for the location up to the previous one. So if I am doing an exclusive scan then considering this a_0 up to a_{n-1} as my input the output that I am looking at is at the initial position, I am not having a_0 .

What I am having is i which is the identity of the operation that is being given here. Considering that the operation is addition the identity would be as 0. So I would write as 0 here, in general, I

I am just reading the identity as the operator i the identity of the operator under question. So here considering addition, I will have 0 in the next location. There is 1st location, I am just having a0; in the second location, I am having a0+a1.

So in the second location, I am having the sum of the first 2 elements; in the third location, I am going to solve the first 3 elements and so on so forth. Whereas if I have done an inclusive scan in the second location, I would have put a0+a1+a2 like that.

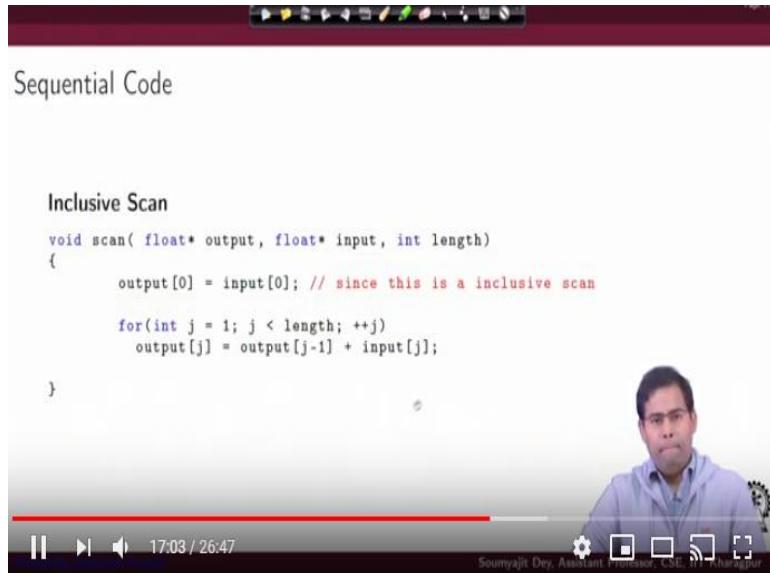
(Refer Slide Time: 15:18)



So this is a example by figure that if I am doing an inclusive scan; so this is the situation for an exclusive scan. Now what would be the situation for an inclusive scan? By the way I mean you may be wondering why we are suddenly using this word scan here, because in general this operation is popularly known as a scan of the data under question; a scan of an array. So when I say it is an inclusive scan the picture like ideally we are looking like this.

Suppose, this is your input array and your output array is being computed through the stages like this that $3+1$ is 4 and $4+7$ is 11 like that.

(Refer Slide Time: 16:05)



So if we just take an example of the code for inclusive scan it is going, I mean this is a normal sequence. Shells code, off course, it is just a simple follow; please take a second and have a look. All we are doing is you are accumulating the values in the output array. So in the j th iteration you are computing the output j as simply the previous iterations output plus the current location j value for that input array right input $j+1$ currently accumulated value in output $j-1$.

This term is necessarily going to give you the output j value so this is nothing but the way you are sequentially iterating over. So that is essentially that code is basically the behaviour like this that one thread of computation either editing like this and providing that output referrals. I hope this correspondence is clear to you. So you know sequential code that is out of program flow will go.

You are just accumulating the value here and you are just considering the previous accumulated value and so that is, let us say in this iteration you are considering the previous accumulated value output arrays location; here in the current location of the output array $j=1$. So this is your output $0 + \text{input } 1$. Then output $1 + \text{input } 2$ should give you output 2 , output $2 + \text{input } 3$ should give you output 3 so on so forth.

(Refer Slide Time: 17:44)

Sequential Code - Complexity

- ▶ Code performs exactly $n - 1$ adds for an array of length n
- ▶ Work complexity is $O(n)$
- ▶ Very large n , motivate parallel execution

17:59 / 26:47

Soumyajit Dey, Assistant Professor, CSE, IIT Roorkee

Now off course, it is very easy to understand that these are completely sequential code. So the work complexity is pretty high for such a simple operation and since there is lot of possibilities for parallel execution.

(Refer Slide Time: 18:03)

Parallel Prefix Sum - Hillis/Steel Scan (Algorithm 1)

```

for d = 1 to log2n do
    forall k ≥ in parallel do
        x[k] = x[k - 2d-1] + x[k]
    
```

We can just do things in a smarter way. So consider initial parallel implementation. So this is Hillis or Steel scan; consider this algorithm here. So what we write here is a parallel version of this for loop, we are trying to save the operations we are going to do in parallel. So for every location k , we are saying that okay there will be some parallel additions going on and the number of stages of the parallel additions will be $\log n$.

So we will be doing $\log n$ number of stages of parallel addition and in each stage, what are the better additions we would be doing? It would be that every k th value should be summing up its own value with x_{k-2} to the power $d-1$. So d is the current stage value. As you can see its quite simple its initially $d=1$. So I am just saying that okay every thread and I should be summing up its location value with the previous value that is.

So if d is 1 so then $x_{k-1}+x_k$. So this is what is going on. If we just look at the data flow here; so these are initial array for $d=1$ what we are doing is for every threads I am just summing up its value with the previous locations value. So off course, I would need half the number of threads as is the number of locations.

So this gives me the sum. These are the partial sums that I am computing. For x_1 location I have the summation x_0+x_1 . For x_2 location I have the summation x_1+x_2 like that. So I hope this is clear for let us sum x_6 the sum x_5+x_6 and so on and so forth. And as I go to $d=2$ let us see what happens. So now in the second stage my stride is increasing. For the threads I am increasing my stride.

First of all how many threads are now operating? If you see, I have started initially with a 8 sized array and the number of threads I am operating is a basically 1 less than then the total number of locations. So since my total number of locations is 8, I am operating here with 4 threads. Off course, that is required because every thread is going to add its own locations value along with the previous proceeding locations.

Own location means if I am assuming access expression we use identity. So just location x for tid ; x has tid^{th} location that is what I would say. So initially I really do not have a half number of threads of the array because every threads is going to add its own location plus the previous location. So if the total is N here the number of threads would be $N-1$ that are active here. Now what happens in the next iterations as we can see with $d=2$. And so here we will get $k-2$.

So every location we will start adding up with one hop. We will start doing a work in stride of 2. Now if that start happenings, how is it first of all helping me meet the objective? So again if we

just consider then what are the total number of threads that are required here? So first of all that would be N-2 because I have 6 threads operating here. But the issue is; why is this working fine?

So let us inspect that, so here I have the value x_0+x_1 , here I have the value x_1+x_2 . Now if I add this with x_0 I will get $x_0+x_1+x_2$ here. If I look in here I have x_0+x_1 , here I have x_2+x_3 operating at a stride of 2. If I add, I get $x_0+x_1+x_2+x_3$ so x_0 up to x_3 . Similarly, if we just check x_4, x_5, x_6, x_7 add them of to x_4 to x_7 . Now if we go to the next stage which is $d=3$ and that should be the last stage here because your N is 8 and $\log N$ base 2 would give you $d=3$ and now you say that okay now what is my stride value?

So since d is 3 so it is 2 to power 2. So that means now the threads would operate at a stride of size 4. So you start with a stride of 1 and you are just multiplying your strides right by 2. And of course this should not work. Why? Because first of all let us again write the number of threads you have active.

So here you are working with $N-4$ threads is just subtracting the number of strikes the number of threads that are basically subtracting the stride value from N because every threads is operating with that stride value. So you can surely having that many strides left as simple as that. So now if you look at the correctness of the algorithm is very easy to figure out because here I have essentially computed the sum up to x_0 to x_2 .

Now if we look at a half like this; so where do I have the sum from x_3 up to sum the next 4 values x_3, x_4, x_5, x_6 sitting here. What is their distance? They are sitting at the half of 4. So if I consider these two I get x_0 to x_6 . So that is the inclusive prefix sum up to this point. If I consider this point and this point, what do I really have here? I am having x_1 to x_4 I have here just the x_0 .

So I will just add them up and I would get x_0 to x_4 . So I would get an x_0 to x_4 here. Just take another example and another location. So as we can see that here I have x_0 to x_1 and look at half of 4 from here. So from here you have x_0 to x_6 so if you just add them up you have x_0 to x_2 to x_5 and so you will add up and get x_0 to x_5 . You can easily write a formal proof of this it is quite easy to figure out. So essentially we can understand that this algorithm is working right?

We are not going into the formal proof because that would more be a part of algorithm course here, but all were trying to introduce here is how the algorithm works. It is easy to figure out illustrated at its neighbourhood values at the shortest distance and then add in the next iteration. It goes to add the next the values on already accumulated in the neighbourhood. And next it goes on increasing its horizon.

And finally every threads will end up the way, the accesses are designed; the threads will end up computing values from the initial up to each location. So with that, we will end this lecture and in the next lecture we will try to figure out the more refined versions of this algorithm and how this can be implemented and what are the technical issues with this. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

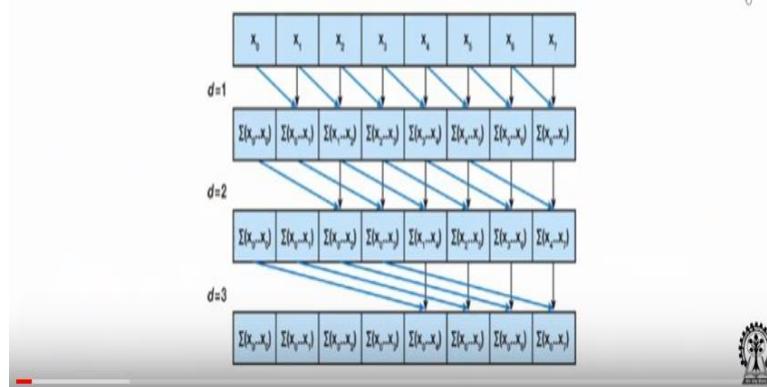
Lecture – 34
Optimizing Reduction Kernels(Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming.

(Refer Slide Time: 00:30)

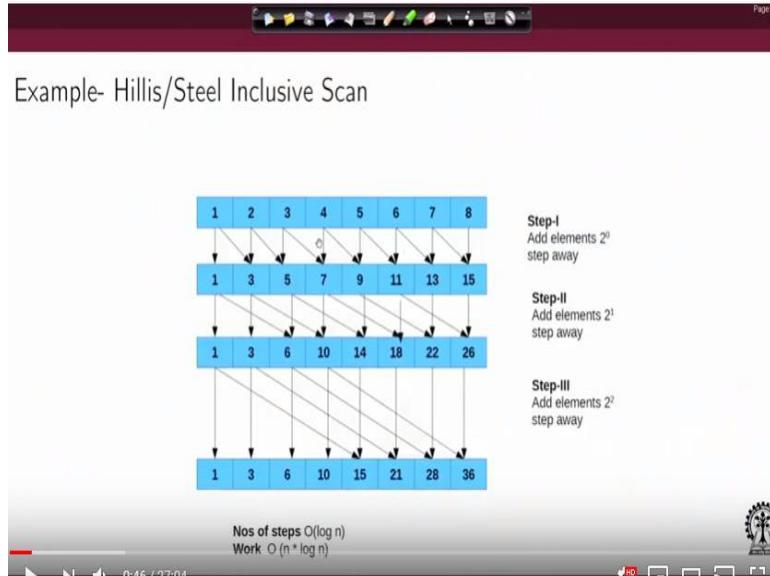
Parallel Prefix Sum - Hillis/Steel Scan (Algorithm 1)

```
for d = 1 to log2n do
    forall k ≥ in parallel do
        x[k] = x[k - 2d-1] + x[k]
```



So, in the last lecture we have been discussing about some algorithm which can perform parallel computation of prefix sum and we argued that why this algorithm would work.

(Refer Slide Time: 00:43)



But let us look at also a picture which just signifies and highlights the properties of the algorithm we discussed. Initially, step 1 is adding elements that steps of at a stride of 2 to the power 0, 2 to the power 1, 2 to the power and like that. So in each iteration the number of threads that are active is decreasing from $n-1, n-2, n-4, n-8$ like that. The number of steps here is the $\log n$, off course, because that is a depth.

(Refer Slide Time: 01:20)

- ▶ The algorithm performs $O(n \log_2 n)$ additions operations.
- ▶ Remember that a sequential scan performs $O(n)$ adds. Therefore, this naïve implementation is **not work-efficient**.
- ▶ Algorithm 1 assumes that there are as many processors as data elements. On a GPU running CUDA, this is not usually the case.
- ▶ Instead, the *forall* is automatically divided into small parallel batches (called *warps*) that are executed sequentially on a multiprocessor.
- ▶ The algorithm 1 will not work because it performs the scan in place on the array. The results of one warp will be overwritten by threads in another warp.

And what would we analysis here; first problem is that sequential scan performs the order of n adds and this implementation apparently seems to be naive. Because we also have that many adds there. The more important point here is that if we are trying to implement these directly in

just the way we talked about the loop; if we just implement this kind of a loop in form of a CUDA program, it is not going to work.

Why? First problem is here we are assuming that we have as many compute codes as are the number of active threads. Why would I say that? You see unless, we assume that we are going to have a problem. Why? Because in case its not so, for example in a GPU that is really not going to be so considering a big array; you have launched lots of such threads for doing this prefix scan operation and you do not have many threads.

So you are now going to do the processor when you are trying to implement this for a loop in parallel that means its going to, I mean, how will really you execute this as a CUDA program? The first thing you will do is you do not have such a parallel loop, you do have a kernel where inside the kernel you would have only this. So first of all, this is not a representation of the CUDA code. This is just like a way we are trying to tell that how the parallel algorithm would work.

I mean that is something we would like to make clear here. We will say that this is a for loop inside the for loop. I am trying to say that this is the activity that is going to happen in parallel. But when we talk about the corresponding CUDA implementation we do not have something like this. We just define a par thread activity and the par thread activity will be defined that okay each thread will be executing this outer for loop and this statement that is how it will work.

So then we will be launching those many threads in parallel and they would be divided into warps. And those warps are at the mercy of the hardware scheduler because the warps will be scheduled based on the hardware schedulers algorithm and we do not have any control there to which warps goes faster and which warps get slowed down. And due to that we will also have problems. What is the problem? The problem is that the results of one warp can get overwritten by the threads of another warp.

The reason being, here the way we have presented the algorithm, things are happening in place. So as you can see that we are writing, every thread is writing its results on the same array. So the

moment this computation gives divided across warps there would be locations where you need data you need data across warps.

So unless we have some synchronization primitive or we have some way to control that, how the works interact among each other in this naïve implementation. If I just do a paradise and here in a naive way, I mean, where every work is warp is working in place things are going to be very bad. So what is the option in that case? How can I really make the warps work in such a way that they do not override results, computed per parallel partial results, computed by other active warps.

(Refer Slide Time: 05:09)

A double-buffered version of the sum scan from Algorithm 1

Algorithm 2

```

for d := 1 to log2n do
    forall k in parallel do
        if(k ≥ 2d) then
            x[out][k] := x[in][k-2d-1] + x[in][k]
        else
            x[out][k] := x[in][k]
        swap(in,out)
    
```

The diagram shows two horizontal rectangles labeled 'in' and 'out'. Red arrows indicate data flow between them. A red bracket groups the 'if' and 'else' statements, and another red bracket groups the assignment statements. A red circle highlights the condition 'k ≥ 2^d'.

One option is that again we are just writing the pseudocode here. I mean, we are trying to give the activity description which you have to implement as a CUDA code. Try it; let us first understand that how this scheme will ever that problem is called a double buffered version of the algorithm.

Why double buffer in the algorithm; One example we just showed that in place implementation which, we argued that, is not going to work in case of work based scheduling of the GPU. If we consider this double buffered implementation we are saying that, hold on; Let us consider that we have two such arrays in and out. Updates by the thread will always happen on that out array. So what we will do is first thing we will figure out what is the depth.

So in the CUDA code this thing will need be there. These all parallel activity, these are parallel thread activity, every thread falls into the loop; in the loop first it has to figure out what is that depth at which does the operating. If the depth is 1 you have n-1 threads working. If the depth is 2 you have n-, sorry, if the depth is 0 you have n-1 threads working.

If the depth is 1 n-2 threads, if the depth is 2 n-4 threads like we have already discussed. So that decision is taken by this condition. If so based on that depth I have inactivity at certain parts and threads starting from the rest of the positions are going to work with this much of stride. So this value is actually deciding, what is the stride? As we have already discussed earlier that the stride is equal to, I mean, n-stride is the number of threads that would be working but what are the threads going to do?

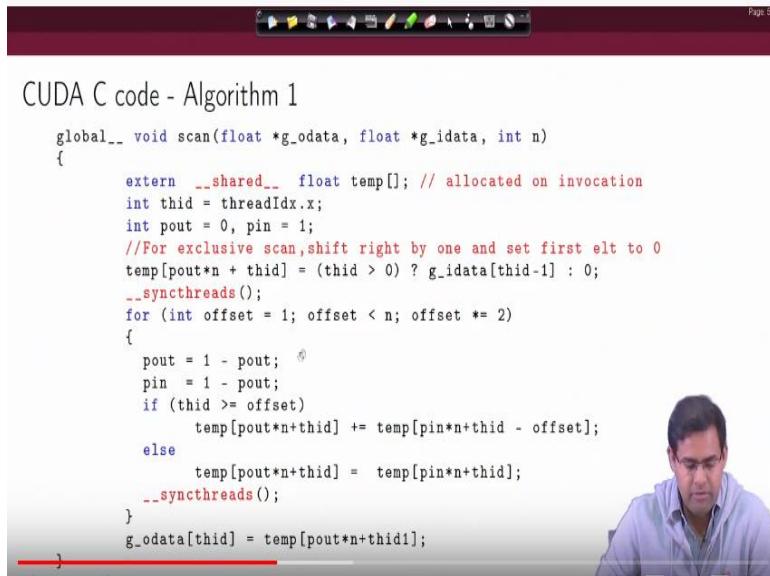
They are going to consider the data values from the in array and do the computations and updates on that out array. So the threads will do their computation on the out array. If those threads do not belong to the portion where we are doing the computation like; if we go back, if the threads are here, if the threads are from here, I mean, with 3 ids of here then at some stage they are not going to work.

For example this thread will stop working here. Only these threads will remain active. This thread will stop working after d=1. So that is the condition which is being signified by this, if we are seeing that okay just do the same thing. Just copy data from in and do the updates in the out buffer. If you are not in the working set, the thread id, then do not do anything but still just completely update things in the out buffer

So the thread either computes that means does the addition or it just updates data from the in buffer to the out buffer. At the end of this you do a swap. That means exchange the values from in into the out. That means after this you have a consistent view of whatever this threads has done in the in buffer once this loop is processed by some warp.

So in that way now the earlier problem can be mitigated and we can have a CUDA implementation.

(Refer Slide Time: 08:44)



```
global__ void scan(float *g_odata, float *g_idata, int n)
{
    extern __shared__ float temp[]; // allocated on invocation
    int thid = threadIdx.x;
    int pout = 0, pin = 1;
    //For exclusive scan, shift right by one and set first elt to 0
    temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
    __syncthreads();
    for (int offset = 1; offset < n; offset *= 2)
    {
        pout = 1 - pout;
        pin = 1 - pout;
        if (thid >= offset)
            temp[pout*n+thid] += temp[pin*n+thid - offset];
        else
            temp[pout*n+thid] = temp[pin*n+thid];
        __syncthreads();
    }
    g_odata[thid] = temp[pout*n+thid];
}
```

So this would be the CUDA code here. So as you can see so, here, this is the loop inside which the above computation, we discussed, would be done. So the thread id is first checking whether its beyond the offset or not. In that case it will add otherwise, it will just simply copy. We are discussing it at a high level. Every thread would synchronize at the end of the iteration of the loop. That means every thread synchronizes with all the operations inside one depth level and then the entire loop goes to the next level like that.

Once the entire loop is traversed by the threads in the warp, they are just going to copy back from out to in. And then again somebody some other warps which is progressing, if they require the data they will get the data from the modified in. So this is how this algorithm is going to work. But yet we can say that this algorithm can be further improved in by using a better technique where we can have to do lesser number of adds.

(Refer Slide Time: 10:03)

Parallel Prefix Sum - Blelloch Scan

The algorithm consists of two phases:

- ▶ Reduction Phase: we traverse the tree from leaves to root computing partial sums at internal nodes of the tree
- ▶ Down Sweep Phase: We traverse back up the tree from the root, using the partial sums to build the scan in place on the array using the partial sums computed by the reduce phase.

So let us see that how possible improvement can be done. So this is possible using this Blelloch scan technique here. The idea is to build a balanced binary tree on the input data and you sweep it to and from the root. So you first do our normal reduction and then you do what we call as our down sweep. And if we do that essentially after the downstream operation, we can get all the prefixes computed.

We will see with an example what we really mean here and it should be easy for us to actually mark out the advantages out of this method. So this algorithm will have two phases. In the first stage we have a normal reduction phase just like a normal reduction computation. If we assume addition just like a normal reduction addition examples we did earlier. So we traverse the tree from leaves to root computing partial sums at internal nodes of the tree.

Now why suddenly have we started talking about the tree? Essentially if we look at a reduction operation and if you want to create a sequence of operations, we can say that we are first doing computations at stride =1 and then stride =2 like that. And that I can represent in the form of a tree. So let us say this as initial nodes, we are just trying to draw a normal production picture here.

So you do the sums here, the local sums we are doing; then you go 1 up effectively this is what we are getting. If we just try to create a graph here based on how the values are flowing then

essentially each thread does a computation at the stride 1. Then you do a computation like this like that. So I can say that thread id, all the thread ids, would progress like this.

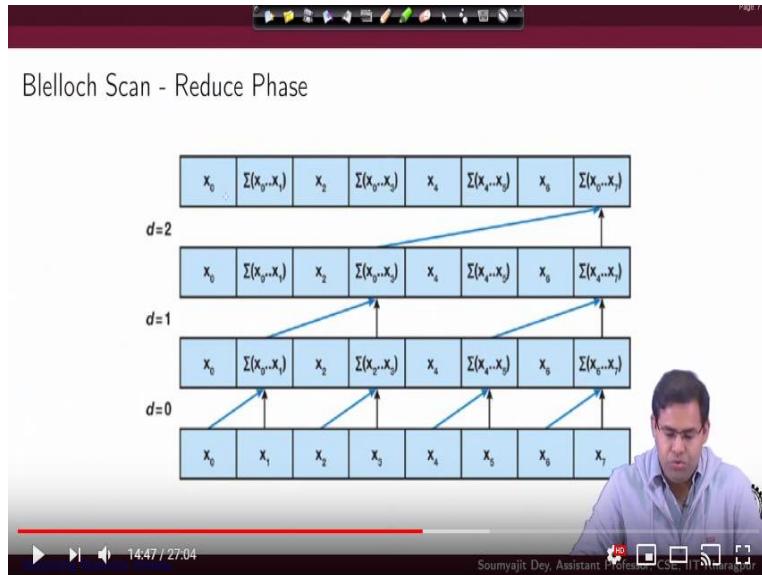
This is the initial computation and then the way I am organizing here, I am trying to say, that okay then these two values are going to be added like that. I mean this is one way of looking at. But then at the end of the reduction phase, what do we really have? Okay let me just redraw it again. So if we create a tree of this computation at the end of the reduction phase we have the overall sum sitting here.

So after this phase what we will do is; we will do a down sweep. That means in terms of the data structure it traverse back from the root to the lower level. And what we do is so what initialise this root with a 0. Suppose I have some value; here value 1. So I will copy this with value here. This value comes here and this value 1 plus the existing value here goes to this point.

So if I draw a better picture, I will start with the root being initialized to an identity. If it is an addition then it is 0, we pass it here and whatever the value was here you add it up with this value and put it here. So in general, if I said this is value 0 and this is value 1 you put value 0 here;

you add value 0 and value 1 and put it here. So this is the operation you keep on performing downstream to get the final prefix sum computed and it really works. So let us see how it works in terms of a parallel implementation.

(Refer Slide Time: 14:39)



So a good idea would be to look at the picture here, because it would convey the situation clearly. So first of all, what we really have at the end of a reduction phase; we understand that at the end of a reduction phase at this position, we would have the final sum computed of the entire array. Now if we look at the each different phases of reduction like this is the tree structure, I was talking about earlier. So considering this as the lower level notes, I can say that here I have computed; I have computed in intermediate value.

These values are again getting summed up by this thread. Again these are intermediate values. They are getting summed up here and I am creating a tree like this with this being the root, note containing the entire sum. Now just try and observe what are the partial sums that we have right now.

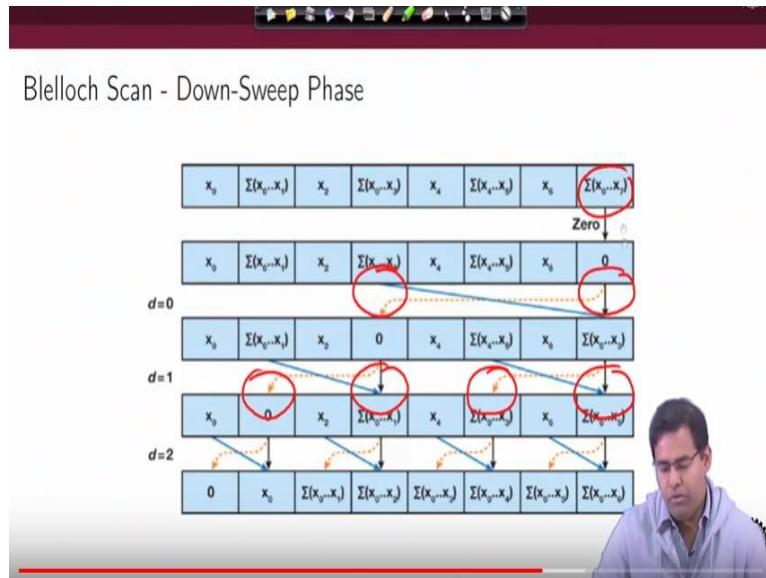
So at alternate locations, I have no sums but just the value corresponding to that location; at alternate locations and other alternate locations, what I have, is the partial sums. So here I have summation up to x_7 alternate location I have summation up to x_5 then in alternate location summation up to x_3 summation up to x_1 .

I have to figure out a smart way such that I am able to compute the prefix sums, off course, the only prefix sums which is ready with me right now is this. And, off course, the other alternate locations because this is also prefix sum valid prefix sums are there. So we have them ready but

what is not there is the prefix sums in the alternate locations which I have to figure out efficiently.

Right now the way we will do, it is as we explained earlier in this picture that okay, we will do a down sweep and perform the operation that I just defined that start with the identity from the root copy that in the left child take the left child's value and add with the root's value and put it in the right child and let us see that why that works.

(Refer Slide Time: 17:07)



First let us just observe the figure. So what we will do is here, in the down sweep phase, we will put the identity here which is 0. And then we pushed back the 0 here okay and then this the input of the down sweep phase. First thing we do is we remove this and we will just remember this and push a 0 here in the down sweep phase and put the 0 here.

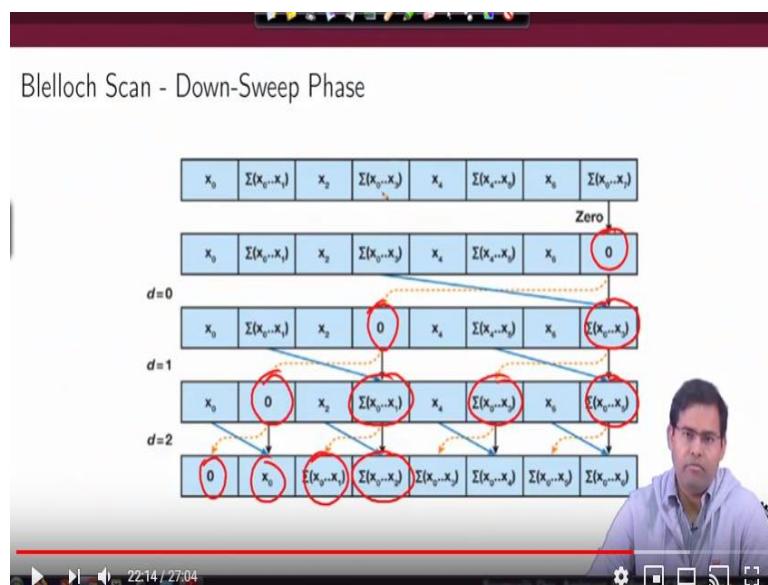
So this is the left child where we copy the value and; what was the original content of this location? It was x_0 up to x_3 so do an addition with this value with x_0 to x_3 sum. So in that way in the right child you put in the summation x_0 to x_3 . Just to repeat this thing in parallel that is the good thing were going to do. So again from this points where we made the modifications in the tree you just keep on doing the same operation.

So you again copy the current locations value, which locations; the locations where you just made the updates the left and right child, you copy it to the left child whatever is the left child value, the summation x_0 to x_1 , add it up with this value that you just copied put it in the right child. So that would give you the summation x_0 to x_1 here and give you a 0 here.

If you propagate it, you get the 0 here and essentially here you are adding 0 with the x_1 that was already there, 0 with this would give you x_0 here. When you copy from here you put in x_0 to x_1 and the right child value that, originally, was x_2 . So you just add the right child, sorry, the left child value and put it back to the right child. So, this is what you get.

Again I will just repeat the algorithm. What we are doing here copying the value from the root to the left child.

(Refer Slide Time: 19:10)



Let me just redraw these locations again. So you are propagating it here and, what was the original content? This one you are adding, you are with this, and putting it here. Just follow this you are propagating it here in the left, whatever was the original content; this was the original content of the left. You add it up with whatever you propagated and put that updated value to the right. Now you see why this works.

So here are propagated x_0 to x_3 and you are adding up this with the original content which was up to x_4 to x_6 , sorry, x_4 to x_5 and in that way you are getting summation x_0 to x_5 here. You are putting in summation x_0 to x_5 here, you adding with original location x_6 and here you are getting x_0 to x_6 again, if I look here so whatever you copied here was summation x_0 to x_3 , you copy that to the left child x_0 to x_3 ; original content of left child was x_4 you add it up to get summation x_0 to x_4 .

So if we just followed this operation at the end what you get is basically an exclusive scan. So $0, x_0, x_0$ to x_1 like that up to summation x_0 to x_6 . So in this way with this parallel sums getting computed; you end up the alternate locations prefix sums been computed. you already had something here x_4 to x_5 and then x_0 to x_3 , x_0 to x_1 , x_0 to x_7 like that. But now you have for every location it done.

So initially after the reduction phase your status was only for the root from the last location or I would say the root location of the tree. You had the total sum done that was the correct prefix sum for that location. But for the other locations, there were sums which were actually for certain sub string, sequence of locations and in the alternate locations you just had individual values.

But by doing this operation you are now able to get when the down sweep phase ends, you are now able to get a continuous sequence of prefix sums for the entire array. So I would just repeat the operation that we did once this value computation of the reduction phase is done. Reduction phase gives you the prefix sum only for the last location; and for the other locations, you have at alternate locations, the only value that was there. And at alternate locations you have some subsequence sums that have been computed.

So now you start the down sweep phase. So you initialize this last location with a 0 and you start propagating this to each left child and whatever is the content of the left child add it with whatever you propagate and put it in the right child. And in that way when you in the down sweep for the requisite number of depth, so $\log n$ depth, then you end up getting the array with the actual prefix sums.

(Refer Slide Time: 22:50)

Blelloch Scan - Down-Sweep Phase

```

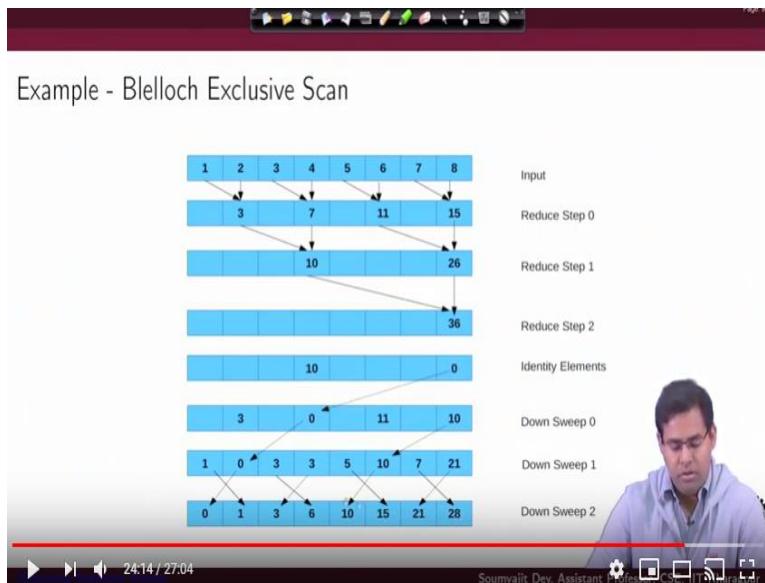
 $x[n - 1] := 0$ 
for  $d := \log_2 n$  down to 0 do
    for  $k$  from 0 to  $n - 1$  by  $2^{d+1}$  in parallel do
         $t := x[k + 2^d - 1]$ 
         $x[k + 2^d - 1] := x[k + 2^{d+1} - 1]$ 
         $x[k + 2^{d+1} - 1] := t + x[k + 2^{d+1} - 1]$ 

```

Okay, if you just look into the parallels pseudo-code here. So essentially you are learning this loop up to $\log n$ depth. Inside each iteration, depending on the depth here what you do is; essentially, you are storing the current locations of value, a temporary variable. Then you are propagating this value so you are storing so, sorry, this is the left child value which you are storing in the temporary variable.

Then in the left child you are storing the current locations value. And then in the right child you are updating with the current locations value which is this and the previous remembered left child value the operation that we just discussed and this is going to happen in parallel inside a depth. And then were going to the next step.

(Refer Slide Time: 23:40)



Just look at the picture. So now we have a overall picture of the Blelloch exclusive scan. So as you can see that we have the reduction done at the reduction phase then I just have this location containing the inclusive sum value but from the others I just have the either the original values or I just have a subsequent sum value. And then I perform the down sweep starting this location with 0 and actually utilizing all the other location values. And we see that in parallel, we are able to compute the value for the prefixes as an exclusive scan at the end.

(Refer Slide Time: 23:40)

```

__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern __shared__ float temp[]; // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;

    temp[2*thid] = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];
    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();
        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset *= 2;
    }
}

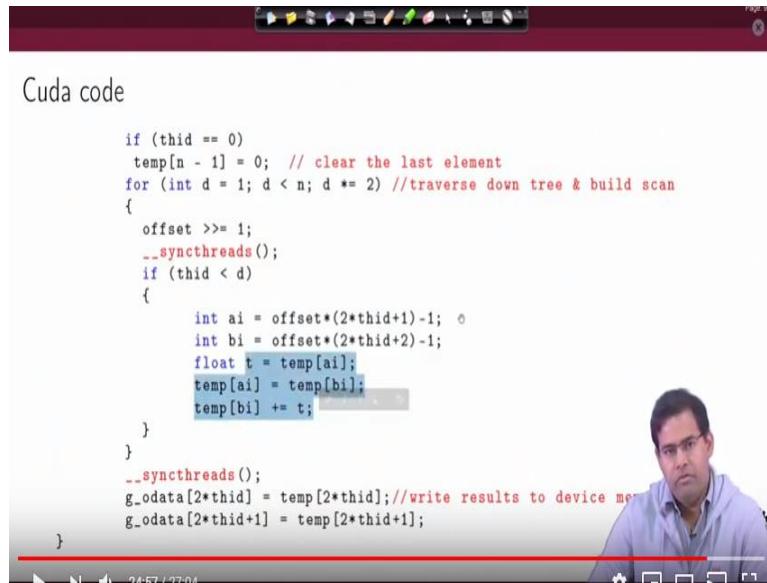
```

So just a representation of the CUDA code here, which would be easy for you to figure out. Given that, we have already discussed that what is exactly is going on. So this is the loop where

you are performing that, so this is basically the part where you build the sum through the original reductions here.

So this is the part where you are doing the reduction. I mean we are not getting into the details here. It is kind of repetitive.

(Refer Slide Time: 24:46)



Cuda code

```
if (thid == 0)
    temp[n - 1] = 0; // clear the last element
for (int d = 1; d < n; d *= 2) //traverse down tree & build scan
{
    offset >>= 1;
    __syncthreads();
    if (thid < d)
    {
        int ai = offset*(2*thid+1)-1;
        int bi = offset*(2*thid+2)-1;
        float t = temp[ai];
        temp[ai] = temp[bi];
        temp[bi] += t;
    }
    __syncthreads();
    g_odata[2*thid] = temp[2*thid];//write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

And then once the reduction is done, this is the next loop where we are performing the down sweep. As you can see, this is where we are performing that operation of getting values from the roots to the left child and adding up the left child values with the right child and reaching the right child and at the end doing a `__syncthread` across all the threads in the block; and then finally updating that global memory.

(Refer Slide Time: 25:13)

Reduction Conclusion

We have learnt how to-

- ▶ Understand CUDA performance characteristics
 - ▶ Memory coalescing,
 - ▶ Divergent branching,
 - ▶ Bank conflicts,
 - ▶ Latency hiding
- ▶ Use peak performance metrics to guide optimization
- ▶ Understand parallel algorithm complexity theory
- ▶ Identify type of bottleneck and
- ▶ Suitably optimize the algorithm

So with this we would like to end our discussions on certain parallel reduction algorithms. So what we have; we like to end up with the following conclusions that we try to understand the performance characteristics of these different optimizations. Like how memory coalescing, handling divergent branches, resolve bank conflicts and trying to hide the latency by making more threads do their work makes sense. And to create performance awareness programs.

So we explored all these techniques in the context of parallel reduction and then we also started looking into other parallel algorithms as an example and their corresponding CUDA implementations. What we did not really do is explore how, I mean, what are the performance issues in those algorithms? I will actually leave those for people who are interested and maybe some of those will also be part of our assignments.

So these are the issues we have to think that how to make the other algorithms that we discuss perform well, I mean, how to create low latency versions of those codes. What are the issues with those codes; what are the good and bad things with respect to memory operations latency, cache conflicts or whether they really have some bank conflicts and all that.

So please look those aspects for the other algorithms that we discussed apart from parallel reduction like in bitonic sort and also the one in prefix sum operations. And with this we

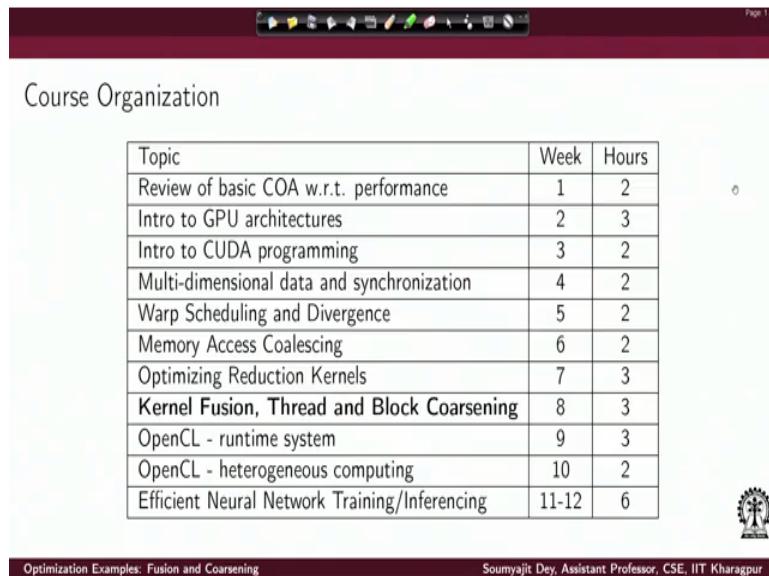
would like to end this topic and in the next lecture, we will be starting a new topic here. Thank you for your attention.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture – 35
Optimizing Reduction Kernels(Contd.)

Hi, welcome to the lecture series on GPU architectures and programming. So today we will be moving over to a new topic which is some different other kinds of optimizations, primarily, the optimizations will be discussing are fusion or fusing multiple kernels and coarsening of threads in kernels. So they are popularly known as fusion and thread coarsening has two possible optimization techniques.

(Refer Slide Time: 00:54)



The screenshot shows a presentation slide titled "Course Organization". At the top, there is a toolbar with various icons. Below the title, the slide content starts with a table:

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

At the bottom of the slide, there is a footer bar with the text "Optimization Examples: Fusion and Coarsening" on the left and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur" on the right, along with the IIT Kharagpur logo.

So coming to this topic on kernel fusion and thread and block coarsening.

(Refer Slide Time: 00:59)

The screenshot shows a presentation slide with a dark header bar containing icons for navigation and search. The main title is 'Recap'. Below it is a list of four bullet points:

- ▶ Multi-dimensional mapping of dataspace
- ▶ Synchronization
- ▶ Warp scheduling
- ▶ Divergence

In the bottom right corner, there is a small circular logo of IIT Kharagpur. At the very bottom of the slide, there is a footer bar with the text 'Optimization Examples: Fusion and Coarsening' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right.

So this will actually require you to do a small recap of the following topics like, how to do a multi-dimensional mapping of dataspace, concept of synchronization of threads, concept of warp scheduling and divergence.

(Refer Slide Time: 01:17)

The screenshot shows a presentation slide with a dark header bar containing icons for navigation and search. The main title is 'Recap'. Below it is a list of five bullet points:

- ▶ Memory Access Coalescing
- ▶ Tiling
- ▶ Matrix Multiply, Convolution
- ▶ GPU optimization techniques

In the bottom right corner, there is a small circular logo of IIT Kharagpur. At the very bottom of the slide, there is a footer bar with the text 'Optimization Examples: Fusion and Coarsening' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right. A video feed of a person is visible in the bottom right corner of the slide area.

And also how memory accesses get coalesced across, I mean, across different thread inside a warp and the concept of tiling and also we will see that the primary workloads will be talking about here would be like matrix multiplication, convolution which are the workloads on which we maybe focusing a bit like that.

(Refer Slide Time: 01:43)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'GPU Optimization techniques'. Below the title, there is a list of optimization techniques: 'Some examples of GPU Optimization techniques-' followed by a bulleted list: '▶ Reduction', '▶ Fusion', and '▶ Coarsening'. In the bottom right corner of the slide area, there is a small video window showing a man with glasses and a dark jacket. At the bottom of the slide, there is a footer bar with the text 'Optimization Examples: Fusion and Coarsening' on the left and 'Soumyajit Dey, Assistant*' on the right.

(Refer Slide Time: 01:48)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Loop Fusion'. Below the title, there is a bulleted list of benefits: '▶ Classical compiler optimization in programming', '▶ Improve performance by reducing off-chip memory traffic' (with sub-points: '▶ reduction of cache miss', '▶ better control of multiple instruction', '▶ reduces branching condition'), and '▶ Operates by fusing iterations of different loops when those iterations reference the same data'. In the bottom right corner of the slide area, there is a small video window showing a man with glasses and a dark jacket. At the bottom of the slide, there is a footer bar with the text 'Optimization Examples: Fusion and Coarsening' on the left and 'Soumyajit Dey, Assistant*' on the right.

So getting into this topic of fusion, let us just do a recap of what is classical loop fusion? This is essentially a classical compiler optimization which helps to improve performance of a program. The primary reason for the performance benefit is that if I do a loop fusion operation it may be possible to reduce the off-chip memory traffic. Essentially you reduce the number of accesses and you can reduce the cache miss.

You can also warp, I mean, obtain better control on multiple instructions. And also it helps in reducing the number of branch condition checks that are required. And we will see that why that is so; we will just revisit the idea of loop fusion in this case. So the way fusion of loops work is

that you have two independent loops. You fuse the iterations of these different loops and it helps in case the iterations are such that they are referencing the same data element.

(Refer Slide Time: 02:58)

The screenshot shows a presentation slide titled "Loop Fusion". The slide contains two blocks of C code. The first block, labeled "//Before Fusion", shows two separate loops: one for array 'a' and one for array 'b'. The second block, labeled "//After Fusion", shows a single loop where both arrays are updated in each iteration. The code is as follows:

```
//Before Fusion
for (i = 0; i < 300; i++)
    a[i] = a[i] + 3;
for (i = 0; i < 300; i++)
    b[i] = b[i] + 4;

//After Fusion
for (i = 0; i < 300; i++)
{
    a[i] = a[i] + 3;
    b[i] = b[i] + 4;
}
```

At the bottom left of the slide, it says "Optimization Examples: Fusion and Coarsening". At the bottom right, it says "Soumyajit Dey, Assistant" and features a small profile picture of the speaker.

So if we look at this kind of an example; here we are just trying to give an example of loop fusion. We are not saying that in this case I will obtain a usefulness which is exactly that we are referring the same data element or not. But let us see what is the advantage we get? So this is a simple loop fusion example. We have two loops here. As you can see in the first loop, i iterate a total of 300 times.

Again in the second loop i again iterate in total of 300 times while in each iteration of the loops I am accessing the array elements of ai or bi. So alternatively I can fuse the loops and I can perform these updates of ai and bi together inside a single loop. So what is the good thing about this fusion here? If you check the original code which was before fusion here we are having the condition checks in the first for loop as well as the condition check separately in the second for loop that gets avoided.

So that is the advantage which we are leveraging in this case. Maybe in case of other examples which we will soon see that suppose I have two loops which are effectively, I mean, in the same iteration they are effectively referencing the same data then it does not make sense to bring to do

that in separate loops and we can just bring those computations inside the same loop provided there are no dependencies and the main thing that holds is in the same iteration same value of i.

I am accessing the same data a_i and maybe a_{i-1} stuff like that in the two loops and that is what we fuse and we get an advantage. We just went to cache access which is fairly understood.

(Refer Slide Time: 04:47)

Kernel Fusion

- ▶ An optimization technique applied to a group of GPU kernels to increase efficiency by decreasing execution time, power consumption
- ▶ GPU kernels cannot be scheduled once launched in device
- ▶ Kernel fusion can rearrange and schedule the kernels from the host side
- ▶ Kernels using the same or different data array(s) can be replaced with a single kernel call
- ▶ The new kernel aggregates the code segments of the separate kernels

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant

Now coming to our specific topic of fusion when we are talking about GPU kernels. So again before getting into this, I would just like to tell you this is just a very simple representative example. It does not highlight all the advantages of loop fusion. It highlights a few. But you can do a study of compiler literature or program optimization literature which is classical and find there are lot of advantages that loop fusion actually brings into play.

In today's lecture, we will primarily be studying this idea of loop fusion from the perspective of GPU kernels. We will study that how the fusions can be done. What are the different possible ways to do the fusions? I mean, in terms of handling the fusion at the thread level or the block level and we will talk about architectural implications and other things in detail later on.

So kernel fusion is basically the kind of fusion where we are going to group a set of GPU kernels together and rewrite a single kernel. And the idea of using this is that if I fuse multiple kernels together it may potentially increase the efficiency by decreasing execution time, power

consumption etc. Now why is this advantages? Let us first discuss certain issues that we have with GPU kernels.

For example, whenever I am launching a GPU kernel after setting up suitable launch parameters. I have no control on how the different threads get scheduled. It is all decided by the hardware scheduler. So I have no control on how the threads gets schedule and typically when 1 kernel is running I wont have the flexibility of running launching another kernel as long as the kernel 1 which was running does not finish.

In modern days NVIDIA and other support concepts called concurrent kernel execution through which I can launch multiple kernels together. But still I would say that we are not having absolute control of how the different kernel threads, threads belonging to different GPU kernels gets schedule once launching the device. Now this is something on which you can have final control if you perform a static optimization like kernel fusion on the input kernels.

So what can be done is, inside the kernel fusion operation I can rearrange and schedule multiple kernels from the host side. That I will be able to control that which threads in which kernel execute fast for which thread in which kernel later on like that. If I fuse them following some suitable semantics. So this is something, we like to see that how such fusion of kernels can be done.

Also the other idea would be that if I am using kernels with same or different data arrays they can be replaced with a single kernel call, that is the fundamental thing. Instead of having multiple kernel launches, I am going to launch 1 kernel if I am fusing multiple kernels together.

(Refer Slide Time: 08:08)

Advantages

Increase efficiency by -

- ▶ data reuse using on-chip memory improves performance
- ▶ reducing off-chip memory data traffic
- ▶ reducing global memory data transfers
- ▶ reducing kernel launch overhead
- ▶ utilising maximum threads in GPU

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant Professor

And we will first study what are the advantages out of fusing multiple kernels together and writing a fused kernel. It may happen that in case I have data reuse using on-chip memory and if that is a property of the individual kernels that they actually are accessing data from the same memory from the same data elements. Then by fusing them I am able to leverage the reuse of the data through the hierarchy of cache and shared memory on primarily cache as there is on-chip memory to the primary and also the shared memory which is there in the SM through the memory hierarchy of the GPU and that would improve the performance.

Now, off course, there are side effects; what happens is that if I am making better use of the on-chip memory I reduced the off-chip data traffic and that is going to provide me, essentially, that is what we mean by reducing the global memory data transfers. The number of data transfer from the global memory will reduce and that reduces the kernel launch overhead. Because I instead of launching multiple kernels, as we have repeated that, I am launching less number of kernels.

Because each time when you are launching a kernel; you have to setup launch parameters. You have to set up the GPU run time system of CUDA will set up internally suitable data structures and that also consumes space and time. So that is the kernel launch overhead which you can bypass if you are launching a single kernel by fusing two or more number of individual kernels which the original programmer may have written.

Now the other advantage that may happen is you can utilize maximum threads in the GPU that means, suppose you are launching single kernels at a time and each of the kernels are not completely occupying all the SPs in the GPU. Then you are not really using the full parallelism offered by the GPU, instead you can perform suitable fusion and utilize maximum threads in the GPU.

(Refer Slide Time: 10:21)

Limitations

Kernel fusion does not always result in performance improvement:

- ▶ architectural resources like the capacity of on-chip memory and registers are limited
- ▶ reduction of off-chip memory data traffic is feasible upto a certain limit
- ▶ overhead in identifying fusible kernels
- ▶ overhead in defining a scalable method to search for the optimal rearrangement of fusible kernels
- ▶ may introduce divergence

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant

Now what are the limitations? Now this kernel fusion may or may not result in performance improvement. So when does it not result in performance improvement, for example let us also understand that architectural resources like the capacity of on-chip memory and registers inside each of the SMs; they are limited in number. If I fuse kernels, then essentially each kernel would place the higher requirement of on-chip memory and registers.

That is good; that does not help performance as long as I have enough on-chip memory and registers to support the execution of a fused kernel. But in case the requirement becomes too high, then the fusion is not really advantageous and it may lead to further overheads and reduction in speedup. Now again in a similar way we will have all the other points. For example, this idea that I would get reduction in off-chip memory traffic that is also feasible up to a certain limit.

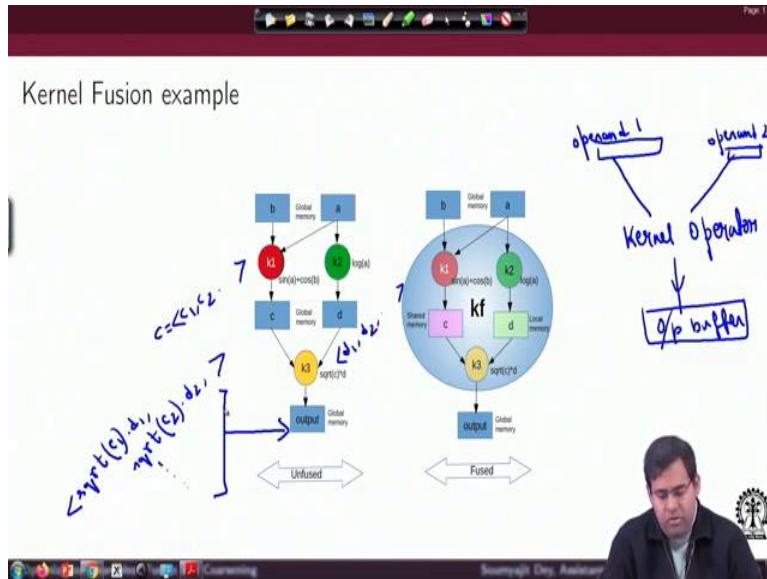
Beyond which that advantage will also go, I mean that advantage will also be lost and the other thing is when you are fusing kernels there may be an overhead in case the kernels do not have an identical data, I mean, data access patterns. There may be overheads in terms of fusing the kernels. These are the few things which will also study. Also, in general, it may be difficult to search for optimal rearrangement of fusible kernels.

So essentially first thing is I have to identify which are the kernels, which can be fused to our advantages. Then the next problem can be that what should be if there are many kernels which I am deciding to fuse and the next problem is more related to scheduling of the kernels that can there be a suitable arrangement of the threads which use me the best possible fusion. Now that can also be a decision which may be quite complex in general.

Let us not get to that. And also in certain cases if I am fused in process of doing the fusion, I am incorporating too many conditionals inside the fused kernel code and it becomes very complex that may also introduce divergence which was not the case earlier. So we have to understand that essentially this is an optimization where we are trying to exploit the entire parallelising the entire compute and memory bandwidth available with the GPU with respect to doing the fusion.

But if we overdo it, it can actually create some issues and speed of gains may not happen. So one has to decide on doing the kernel fusion based on his architectural knowledge and the usability of the GPU parallelism.

(Refer Slide Time: 13:15)



So what we will do is we will provide some examples and we will classify this idea of kernel fusion into several formal classes and figure out in which cases which kind of fusion is advantageous. In this figure we are just providing a basic simple example of kernel fusion. So we are just trying to show that in the global memory we have some data segments labelled with b and a.

So let us consider that it is a big data chunk sitting in b and also similarly in a. And we are considering 2 kernels k1 and k2. So this is more like a task graph. So k1 is a kernel task graph or I would say it is a DAG representing the partial order relationships between the operators and the operant. The kernels are the operators. So the k1 and k2 are the kernels which are the operators. They are operating on the data operants.

Like in this case k1 is a kernel an operator which is going to perform a $\sin a + \cos b$ transformation on data inputs from b and a. So these are the buffers containing that data. Sequential data of b and sequential set of data essentially a big a large buffer or containing the data in a. K1 will launch multiple threads in parallel and for each data item in b and a it will simply compute a $\sin a + \cos b$.

And this sequence of values k1 is going to write in an output way for c. So this entire dependency is highlighted here in the form of our task graph. So you have I would let us just

define the task graph here you have operant 1, you have operant 2, you have a kernel operation and here you have the output buffer. These are the input buffers. So follow I have this k1 which is following these kinds of a structure.

And then similarly I have k2 which is working on a and providing me as an output buffer d containing the sequence of log a. All the components of a and in that way in the global memory once k1 and k2 have executed we are saying that we will have 2 output buffers c and d filled with the outputs of k1 and k2. And then I want to execute k3 which is essentially going to take the contents of c and d and is going to perform a further complex operation is going to do a square root of c and multiply by d and output that in the global memory.

So c let us say c is containing the values like that and this is containing the values like that. So here so this is what you get in output buffer sequence of this kind of values after k3 executes. So this is like a task graph. We are trying to show that we have kernels k1 and k2 processing input data buffers b and a and producing the output in the global memory output buffer. This is the situation when we are doing an unfused execution.

What happens when I fuse these kernels? So essentially we can replace these 3 independent kernels k1, k2 and k3 and create 1 kernel called kf. So then what would happen is? Kf would be a single kernel whose inputs would be the global memory buffers b and a. So it will copy data from b and a, it would compute sin a + cos b and log a. And it would have the log a in the local memory and it is just 1 possible implementation.

It would be storing the sin a + cos b data elements in the shared memory. This again I am just saying this is an example and then it will perform all the operations of k3 on these data points and then do the global.

(Refer Slide Time: 18:18)

The screenshot shows a video call interface. At the top, there's a toolbar with various icons. Below it, the title "Code snippet for Kernel 1" is displayed. The main area contains C++ code for a CUDA kernel named "process_kernel1". The code calculates a linear index "i" based on block and thread indices, and then performs a computation involving sine and cosine functions. At the bottom of the video frame, there's a dark bar with the text "Optimization Examples: Fusion and Coarsening" on the left and "Soumyajit Dey, Assistant /" on the right, along with a small profile picture.

```
__global__ void
process_kernel1(float *a, float *b, float *c, int datasize) {
    int blockNum=blockIdx.z*(gridDim.x*gridDim.y)+blockIdx.y*gridDim.x+blockIdx.x;
    int threadNum=threadIdx.z*(blockDim.x*blockDim.y)+threadIdx.y*blockDim.x+
        threadIdx.x;
    int i=blockNum*(blockDim.x*blockDim.y*blockDim.z)+threadNum;
    if (i<datasize)
        c[i]=sin(a[i])+cos(b[i]);
}
```

So this is the code snippet I have for kernel 1. As you can see that you first find out what is the total number of blocks by this statement, total number of threads by this statement then you use this total number of blocks and total number of threads to create i which would be the total I mean exactly the total number of threads that have been launched. So I hope this is clear. So I am finding out what is the block number; sorry.

Let me just repeat this part. So block number so essentially what you are doing is we are multiplying this block Id z with grid dimension on x and y . And then you are doing an addition with block Id x dot y times the grid dimension in x direction and then you are adding block Id x . So in an effect all you are doing is you are considering for part thread. What is the corresponding block?

I mean if I just linearize the total set of blocks then what is the block and which it belongs? And then I am finding out what is the thread number for it inside the corresponding block and then if you just execute this kind of a statement you figured out what is the global idea of the thread? Essentially the relative position of the thread across the entire input data space.

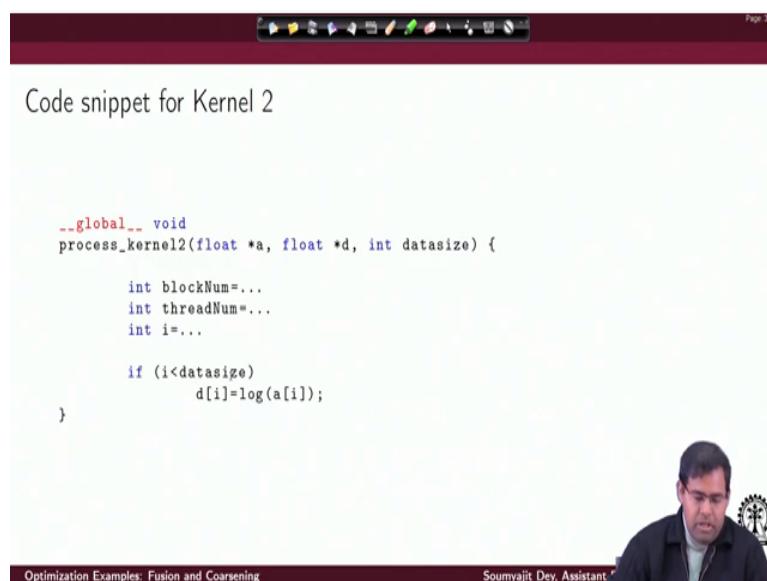
And of course we require that considering that I have these 2 input buffers a and b which are 1 dimensional arrays containing the data a_i 's and b_i 's. I mean of course they can be in other I mean in other setting fundamentally they would all be linear and that is their primary reason why you

would like to know the global thread Ids. So once you identify the global thread Id. You use that global thread Id to have a linearized access to a.

You just access ai, you just access bi. I mean when is that a valid access? As long as i is less than a data size which is a variable that tells you up to what position in both the buffers a and b. You are supposed to contain I mean they are supposed to contain a valid data. So you compute the global thread Id, you use the global thread Id to compute the linearized position or offset inside the array similarly the location from b.

And then you are simply doing a sin ai + cos bi to figure out what should be ci from the with respect to the input buffers.

(Refer Slide Time: 21:03)



Code snippet for Kernel 2

```
__global__ void
process_kernel2(float *a, float *d, int datasize) {

    int blockNum=...
    int threadNum=...
    int i=...

    if (i<datasize)
        d[i]=log(a[i]);
}
```

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant Professor

And similarly if I can just do it for kernel 2. I just can find our block number, thread number and then again compute the global Id. I will just see that whether it is going to access something valid inside the data size and then I will just consider the corresponding position in the input buffer a for using a linearized access and stored the data into the output buffer di. So I do I provide a code snippet for kernel 1 a code snippet for kernel 2.

(Refer Slide Time: 21:34)

Code snippet for Kernel 3

```
__global__ void
process_kernel3(float *c, float *d, float *output, int datasize){

    int blockNum=...
    int threadNum=...
    int i=...

    if (i<datasize)
        output[i]=sqrt(c[i])*d[i];
}
```

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant Professor

And then I also have a code snippet for kernel 3 where I am using i as the global thread Id and using that I am doing this computation of $\text{sqrt}(c_i) \times d_i$ and producing the result to output i.

(Refer Slide Time: 21:48)

Code snippet for fused kernel

```
__global__ void
process_fused_kernel(float *a, float *b, float *output, int datasize) {

    __shared__ float c[blockDim.x * blockDim.y * blockDim.z];
    float d;
    int blockNum=...
    int threadNum=...
    int i=...
    if (i<datasize) {
        c[threadNum]=sin(a[i])+cos(b[i]);
        d=log(a[i]);
        output[i]=sqrt(c[threadNum])*d;
    }
}
```

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant Professor

So of course it is very simple to fuse all these sequence of instructions to a fused kernel. All we need to do is we will have a similar implementation compute block number thread number compute the global thread Id. If it is less than the data size you just use it to compute c, you just use it to compute d and then use c and d and apply the last operation to compute the output i value.

So this is kind of an example of fusion of kernels and as you can see that what is the advantage out of it? Now once we understand what how a fused code should look like you can see that is just a sequence of the kernel operations that have been performed. Now what is the control I have here? I can control this sequence. So this is important. Let us observe what is the sequence we are doing? So we are first computing c then we are computing d and then we are computing output.

Now when we look at the DAG it is not serializing k1 or k2. It is just saying that okay you can you have to execute k1 you have to execute k2. Once both finished then you have to execute k3. So both are valid execution orders here. When I am fusing the kernels I can do. Since if you look at the dependency structure of k1, k2 and k3 it is something like this which means if I just serialize I can have k1s operations followed by k2 followed by k3 or k2s operations considering that I have serialized them in the kernel code and this.

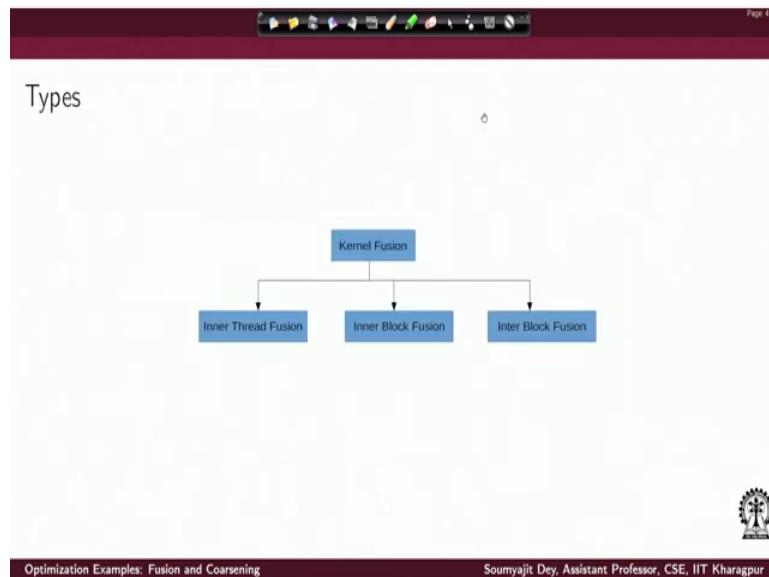
So when you look at the fused kernel I believe the log is in k2. So what we are doing is here? For each thread I am doing k1s operations followed by k2 followed by k3s operations. This is also a valid order. So if we just execute like this the original kernel 3 arrangement of 3 kernels essentially I leave it to the system to figure out how to execute the kernels? What should be the sequence followed by each thread and all that.

But when I am fusing that sequence is set by the programmer. That is one possible observation here. That observation is let us identify what is the advantage in this case out of fusion? Well in this case it is a big advantage because each of the kernels are not doing that bigger computation but there is lot of global memory reads and writes. So k1 and k2 are reading from b and a. Then they are writing back to c and d and then k3 is reading from global memory again and writing back to the global memory.

If I fuse k1, k2 and k3 then what happens is? I have a global memory read from b and a and I have a global memory write which is at the final output. In between I just will have I will just have shared memory and local memory writes and reads. So that would significantly increase my execution time sorry significantly decrease my execution time and provide a significant speed up

here just because I have avoided lot of unnecessary global memory transactions. So in this case the kernel fusion operation was found to be advantageous.

(Refer Slide Time: 25:26)



Now while that was simple example and earlier we have discussed that it there are certain advantages of the fusion operation. There are also certain disadvantages of the fusion operation. The first thing we will do at this point is; we will classify the different types of fusions, we will formalize this classification different types of fusions possible among GPU kernels and then we will speak about what are the different advantages and what are the advantages and disadvantages of each type of fusion.

So kernel fusion primarily can be classified into 3 types. One is inner thread fusion, the other is inner block fusion, and the third one is inter block fusion. So maybe we will end this lecture with this introduction of this classification and in the next lecture we will go into the detail of each of these type of fusion optimizations. Thank you for your attention.

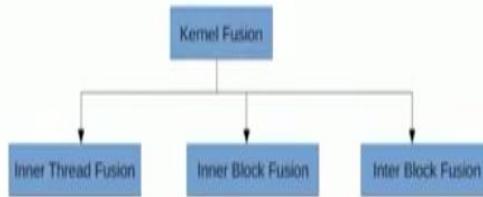
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture - 36
Kernel Fusion, Thread and Block Coarsening (Contd.)

Hi, welcome back to the lectures on GPU Architectures and Programming. So in the last lecture, we have just started the discussion on kernel fusion.

(Refer Slide Time: 00:32)

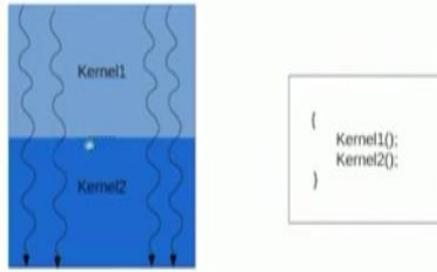
Types



One of the possible optimizations in GPU programs, and we have classified the different kinds of fusion optimizations that are possible. Just to recall, they were inner thread fusion, inner block fusion, and inter block fusion.

(Refer Slide Time: 00:47)

Inner Thread Fusion

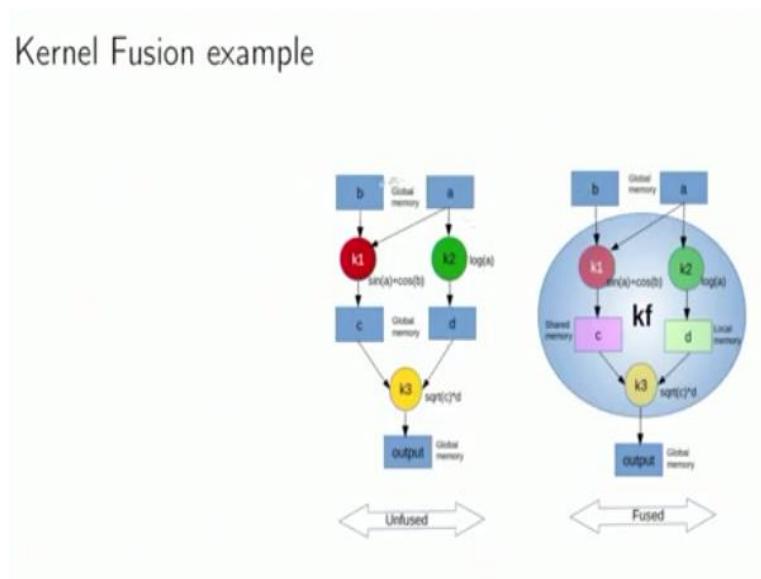


So let us get into the detail of what is inner thread fusion. So essentially, when we speak about inner thread fusion, suppose I have kernel 1 executing on some data segments and computing some output, which is to be then again passed to kernel 2. So consider that there is a dependency graph, where you have an input buffer. So this data stream would be operated by kernel 1, it would be returned to some output buffer and then it would be again read by kernel 2 and then it would be output again to some output buffer.

All we are doing is essentially just fuse this part, create one kernel, and let each of the threads perform the operations of kernel k1 followed by the operations of kernel k2. So essentially we are allocating the job per thread from kernel 1 followed by kernel 2 to a single kernel, where each thread performs the operations of kernel 1 followed by kernel 2. So if you remember the example we took earlier, let me just go back to that task graph here.

(Refer Slide Time: 02:15)

Kernel Fusion example



This was the task graph of kernels and we fused all of them and this was my version of the fused kernel.

(Refer Slide Time: 02:23)

Code snippet for fused kernel

```
__global__ void
process_fused_kernel(float *a, float *b, float *output, int datasize) {

    __shared__ float c[blockDim.x * blockDim.y * blockDim.z];
    float d;
    int blockNum=...
    int threadNum=...
    int i=...
    if (i<datasize) {
        c[threadNum]=sin(a[i])+cos(b[i]);
        d=log(a[i]);
        output[i]=sqrt(c[threadNum])*d;
    }
}
```

As you can see that the final fusion is at the thread level. We compute the global thread ID. We examine whether it is inside the valid data size, then I just perform per thread, the sequence of operations, which were to be performed by each of the three kernels individually. I have just written down those operations in a sequence. Without violating the sequence that was already provided by the partial order, that was enforced by the original kernel DAG.

So this is in a nutshell the idea of kernel fusion, when we do inner thread fusion. So you just write a par thread activity, which is a sequence of the two kernel codes.

(Refer Slide Time: 03:07)

Inner Thread Fusion

- ▶ Combines computation of two kernel into single thread
- ▶ Suitable for both dependent and independent kernels if dataspace size is same
- ▶ Let, $S_{th,i}$ represents the size of threads in a thread block $S_{blk,i}$ represent the size of blocks in kernel i ($i = 1, 2$)
 -
- ▶ For fused kernel -
 - ▶ $S_{th} = \max(S_{th,1}, S_{th,2})$
 - ▶ $S_{blk} = \max(S_{blk,1}, S_{blk,2})$
- ▶ Not suitable if -
 - ▶ Kernels not having same thread/block space
 - ▶ Results in unbalanced workloads between threads



Now the issue is; how do I decide on the threads per block and the number of thread blocks while designing an inner thread fused kernel. So you are essentially combining the computation of the two kernels at the single thread level. It is suitable for both dependent and independent kernels. Dependent kernel would mean that the first kernel's output is the input for the second kernel, as we saw in the task graph. So every edge in that graph is like a dependency edge.

So suitable in that case and it is also suitable for independent kernels, but there is a constraint, that the data space size has to be sent, so that we can use the same data space size to define identical thread blocks for the fused kernel. So consider $S_{th,i}$ represents the size of threads in a thread block. So that is the thread block size and $S_{blk,i}$ represents the size of blocks in the kernel or essentially I would say the number of blocks in the kernel.

Let us just write this; represents the size of thread block. So this is the number of thread per block and $S_{blk,i}$ represents the number of blocks in kernel, I write. So when I fuse them, I have to set the number of blocks and number of threads per block. I need to identify, what is my number of blocks and I need to identify what is my number of threads per block.

So when I fuse the two kernels to create a single kernel, I choose S_{th} , which is the size of thread block as the max of the individual thread block sizes and I choose the number of blocks as the maximum of the number of blocks in the individual kernels. Now this is important because, off course, when I am fusing two different kernels, they may be considering different data space sizes, I mean, they may be considering different ways in which the threads are arranged in the data space, although the data space are just same.

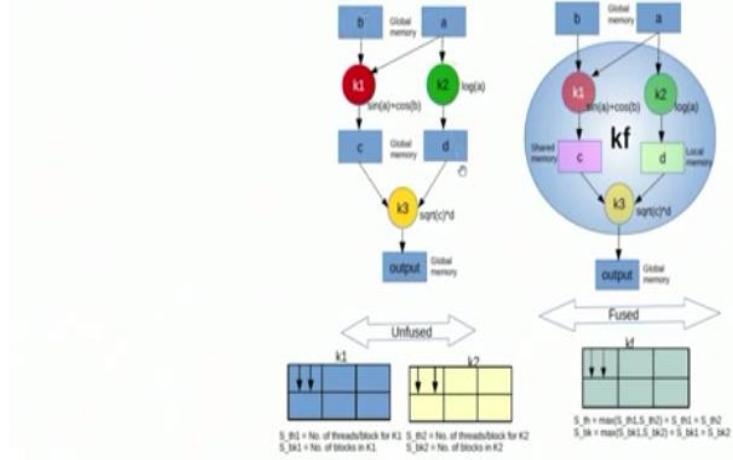
So I write the fused kernel, I have to choose S_{th} , the thread block size and I have to choose S_{bk} , which is the number of blocks. So in that way, when I am doing an inner thread fusion, I set these parameters by doing a max in each case and the issue is; this is not a good optimization in case the kernels do not have same thread or block, I mean, the arrangement of threads and blocks, and threads per block space is very different.

So if the kernels are not having the same threads per block in the data space, then it is not a suitable optimization. Primarily, the reason is that it would result in an unbalanced issue of workload, I mean the kernels different thread blocks, we will have different amount of work to do and that would be issue with respect to exploiting the parallelizing of the GPU in general.

So we will actually prefer this kind of an optimization, if the number of blocks and number of threads per block, primarily the number of threads per block are kind of similar across the different kernels. Let us move to an example and try and figure out why this is really so.

(Refer Slide Time: 07:48)

Inner Thread Fusion Example



Consider this situation, so I am doing an inner thread fusion. So this is original unfused version and this is the fused version. Now this is an example where we are showing that the number of $S_{th,1}$, the number of threads per block, that is 2 and $S_{bk,1}$, number of blocks is 6 and we are just considering a very ideal situation, where everything is same for $k1$, $k2$ and I am just fusing them to create the fused kernel kf with S_{th} and S_{bk} as the number of threads per block and the number of blocks. So it is just a max of $S_{th,1}$, $S_{th,2}$ and max of $S_{bk,1}$ and $S_{bk,2}$. Since they are all same, I get the same values here.

(Refer Slide Time: 08:42)

Inner Thread Fusion Example

Fusion of dependent kernels with same dataspace size and thread/block size

```
//Unfused kernels
k1(a, b, c, n):
    i = global threadId
    c[i]=sin(a[i])+cos(b[i])
k2(a, d, n) :
    i = global threadId
    d[i]=log(a[i])
//Fused kernel
kf(a, b, out , n):
    local c,d
    i = global threadId
    if(i<n)
        c=sin(a[i])+cos(b[i]);
        d=log(a[i]);
        output[i]=sqrt(c)*d;
```



So when I am going to fuse, the important thing here is these kernels, if I am considering $k1$ and $k2$, they are independent, but then $k3$ depends on $k1$ and $k2$. So it is not that all the kernels are

independent. They have some ordering as is enforced by the DAG structure here. So as long as we are fusing dependent kernels with the same data space size and threads per block size, the code is very simple.

So let us say, these are the unfused kernels and we are just trying to provide. So as you can see that we are just providing a structure of the kernel code, let us say k1 structure is like this that you just compute the global thread ID. Again, this is not the valid kernel code, we are just providing a structure of k1 and structure of k2 here. So for k1, you compute the global thread ID, do the operation for k2, you compute the global thread ID and do the operation.

When you fuse them, you will have some local variables to hold intermediate results. So for that you should have some more variable c and d here. Compute the global thread ID as long as it inside the data space size. You compute the first operation of k1, store it in local variable c, compute the operation of k2, store it in local variable d, use c and d to compute the final output. So you use c and d to compute the final output, which would be square root of the first kernel's output multiplied by the second kernel's output.

That is the final value, which is the final fused value. So this was a very easy piece of code, simply because the threads per block and the number of blocks are all same across the kernels. So things look very simple here.

(Refer Slide Time: 11:07)

Inner Thread Fusion Example

Fusion of independent kernels with same dataspace size and thread/block size

```
//Unfused kernels
k1(a, b, c, n):
    i = global threadId
    c[i]=sin(a[i])+cos(b[i])
k2(d, e, n):
    i = global threadId
    e[i]=log(d[i])

//Fused kernel
kf(a, b, c, d, e, n):
    i = global threadId
    if(i<n)
        c[i]=sin(a[i])+cos(b[i])
        e[i]=log(d[i])
```



So now consider this simple situation, where you have the same data space sizes, but now we have some different data space size, but same threads per block size. So again I am just recapitulating here. This is the inner thread fusion example and this is the very simple situation and this picture is just showing. So unlike this program, which was of total fused kernel, this picture is just showing how the k1 and k2's fusion will look like, very similar, just missing the code for k3 here.

But this is very simple and the fusion is quite simple for k1 and k2. So difference with this is here, I also have k3 in the final output. Here I am just showing the example of fusing k1 and k2 here. The two simple operations are done, but now consider the situation where you have a big difference in the data space size. So suppose the situation is that you have same number of threads per block, but you have a difference in the number of blocks.

So that is what the picture is showing to convey. So k1 is operating like this. So for k1 the block structure is this. You have two threads per block, in total you have 6 blocks, but k2 is operating on a larger size d here. So here, for k2, you have this 4 blocks, but the threads per block is same. Now how do I fuse them? Because I am going to produce the output c and e. So as you can see, we have now taken an example where the data space size is varying.

But since I do not have k3 in this example, I just have k1 and k2, I do not have a problem, because k1 and k2 are independent kernels. So as long as they are independent kernels, I can always use them together, even if the data space size is different, as is the case here. The data space size is different here, but in this example we have the same structure of threads per block. That is also important in this case.

Since the thread per block is same, it does not change even in the fused kernel, if I follow the formula, it is equal to whatever was for the k1 and whatever was k2, is same, but since I am doing a max over block size, the number of blocks is now same as the kernel with the larger block size, the number of blocks. So when I am fusing the kernels k1 and k2, which are independent for the output kernel, the kf, I have the larger number of blocks, where the thread per block will be same as the two origins.

So I hope the point is clear. When we are fusing all the three kernels together, we had the requirement that data space size has to be same, but now since k1 and k2 are independent kernels, I do not have the requirement because k3 is not there to work on the final output and since they are independent, they can actually have different data space sizes, which is the situation here and when I am fusing them, so it is again quite simple.

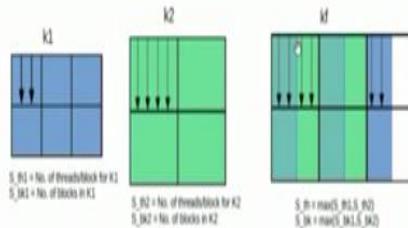
So if i is less than n, then I have this sequence of values. So the program automatically takes care of the situation, that since I have considered the larger number of blocks in my definition of kf. So automatically, the fused kernel will launch the number of threads similar to whatever is required for processing. It is going to launch the number of threads which is similar to the requirements of k2, which had the increased number of blocks and so there will be enough threads to process the data buffer d.

Apart from some of the threads, which are lying in these blocks, all the other threads will also be processing the data buffers from a and b, as was the requirement for k1. So I hope this is clear. Since the two kernels are independent, the fused kernel code is also very simple in this case.

(Refer Slide Time: 15:44)

Inner Thread Fusion Limitation

Fusion of independent kernels with different data size and thread/block size:
NOT SUITABLE: Due to unbalanced workloads between threads



Now coming to the limitations of the inner thread fusion, if I am fusing independent kernels with both different data size and threads per block size. Now it may be unsuitable in this case, again I will just come back here. Here, we had the number of blocks differing across independent kernels, but the number of threads per block was same. That was a good thing and that lead to a very simple code. So this was my code for the inner thread fused kernel.

Sorry, we keep this earlier. This is my code of the inner thread fused kernel considering difference in the overall data size, due to the difference in the number of thread blocks, but both of them having the same threads per block size. So as you can see, that you compute the global thread ID and then you have to see that out, whether the global thread ID would be inside the range of both the kernels, or is it outside the range of kernel 1, but inside the range of kernel 2.

If it is inside the range of both kernels, which is n1 here, then I should actually compute whatever is the operations for both k1 and k2. So then, I am inside thread ID i, which is lying in this common region and otherwise, I will go to other region, where I have to only execute the part for kernel 2. So again we will just recall that these are independent kernels with unbalanced number of blocks, but same amount of threads per block.

So the different data size is there due to the unbalanced number of blocks. Now consider the situation where I have different number of blocks and also different number of threads per block.

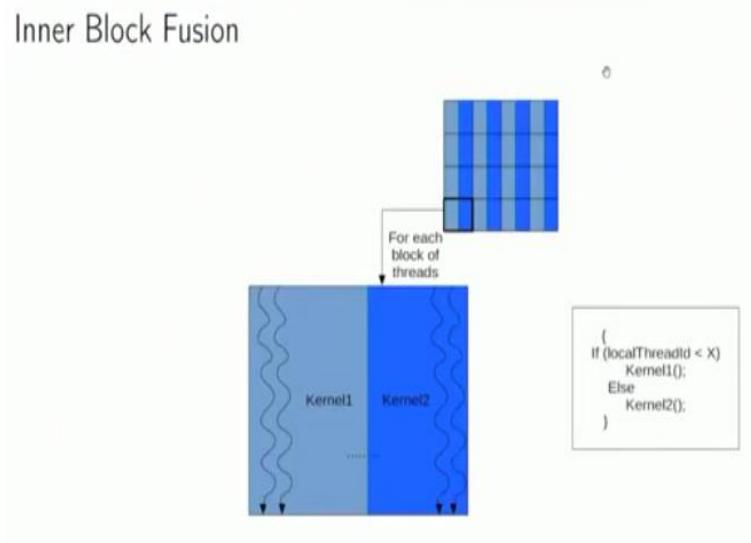
Now that may be bad thing, even for independent kernels, because consider k1 and k2 with these arrangement. So 2 threads per block and 6 blocks and 4 thread per block and 4 blocks. So what are the kf I am going to have 6 blocks and in 6 blocks, some of the blocks, I have full fusion.

So I have 4 threads. In some of the blocks, I have only 2 threads because if we notice the first block, I should have 4 threads, 2 of the threads will execute code for k1 as well as k2, 2 of the threads will execute only the code for k2 and the same pattern will continue for the other three blocks also here. I hope this is clear. Why these may not be a good solution in this case?

First of all the classification of this case is unlike the earlier case, where I had same number of threads per block, but varying blocks, here I have varying threads per block as well as varying number of blocks. Now due to that, when I am fusing, due to the issue of varying number of blocks, I will have some blocks where there is no activity of kernel 2, which has less number of blocks. So such blocks are these 2 blocks. For the other blocks, I have unbalanced execution.

For example, all these 4 blocks, as you can see, there are 2 threads. So each block has 4 threads, which is the max of the threads per block of k1 and k2 and inside these 4 threads, there are 2 threads, which are executing code for both k1 and k2 and 2 threads, which are executing the code only for k2. So this leads to a lot of unbalanced execution, which may not be good.

(Refer Slide Time: 19:36)



Now this leads us to another possible way of fusion. I hope by now, this idea of inner thread fusion is clear to you and this would actually lead us to the other possible ways, in which we can fuse and they are inner block fusion and inter block fusion. So before going into them, let us again summarize the good and bad things of inner thread fusion.

So when you do a thread level fusion of two different kernels, if they are independent kernels with differences in both the number of blocks as well as threads per block size, it may lead to a bad situation, completely unbalanced workloads and that may not be suitable. In some cases, it is okay. For example, the data size is different, but because the number of blocks are different, the threads per block are same.

So in that case, the code does not have too much operate and there is significant amount of balance in execution of the majority of the blocks. This was the picture for that. And when I considered the total fusion of the blocks, our demand was that okay, there should be absolutely same data space size and thread block size and we now coming back to this part, it would make sense. So this is the situation where I am fusing independent kernels.

As long as I consider only k1 and k2, they are just independent kernels. If we go back to the earlier example which was this. So here I was using k1, k2, and k3. So there was also dependency, which means, I can start the code for k3 only when k1 and k2 both finishes. Now the issue is as long as I have identical number of threads per block and identical number of blocks, fusion of dependent kernels is simple in this case while considering inner thread fusion.

But now as we have understood now, what is the implication of differences among independent kernels. So this was my picture of independent kernels getting fused, where they have differences in number of blocks as well as threads per block. So we understood what is the issue here? Now one can easily visualize that how the issue becomes even difficult when you are trying to use this concept of fusion in the case where you have dependencies among kernels and there is no balance in terms of equality of threads per block and the number of blocks, while executing dependent kernels, if you are going for fusion.

So as you can understand that if I am trying to fuse this with identical number of threads per block and number of blocks, things are very simple. my checks are very simple, just check on the global thread ID and execute all the elements, operations and sequence. If there is difference in the number of blocks in some of the kernels, it would be really difficult here. I need to do more fine grain control by introducing more number of these statements and that would be further complicated, if I have also variation in threads per block setting also.

That is why, in case of dependent kernels, it is even more complicated and we had an idea that by even looking at the situation for independent kernels. Because as we saw, as long as it was independent kernel with everything same, that is I have same number of blocks and same number of threads per block, that is the same data space size and same threads per block, then the code was so simple, just a check on whether I am inside the data space, the moment we introduced one difference, that is okay, let the threads per block remain same.

But let there be more number of blocks, so that there is a difference in the data size that introduces one “if” statement to check whether I am inside the common blocks or I am in the blocks only relevant to k2. The more I am going into this mess of the differences in threads per block as well as differences in number of blocks, it is only going to introduce more and more of conditionals, even for dependent kernels and the total program will get much more complicated with more primitives required in case of dependence.

Sorry, this is already sufficiently complex in case of independent kernels and that would require additional primitives, if I am going to use this concept in case of dependent kernels where I also have variations in threads per block as well as block size. So from this, the point to take home would be that when I am trying to fuse independent kernels, we should take care that at least there is similarity of threads per block size, even if there are differences in block size.

But if we have differences in number of blocks as well as threads per block, there may be significant absence of balance in the workload in the fused version and if we bring in the added complication of dependency among kernels, it would require more primitives, which would

complicate things even further. So we understand now when inner thread fusion is absolutely fine, as long as there is full identical arrangement of the data space across the different kernels.

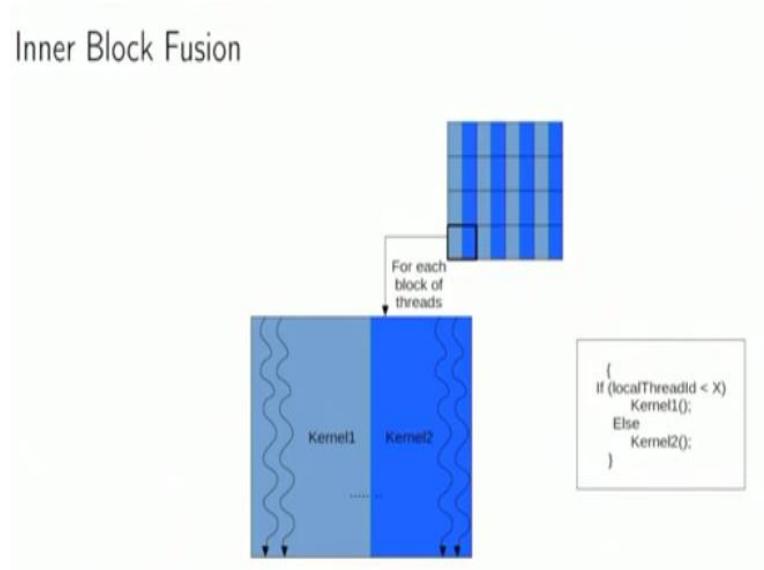
Even if that is not the case, if at least threads per block remains same, still things are okay for the case of independent kernels, and when I have variation in S_{th} as well as S_{bk} , then it is very difficult to achieve workload balance among threads for inner thread fusion even for independent kernels and it is better in those cases to look for other optimizations in respect to fusion, like inner block fusion or inter block fusion.

So with this, we will be ending this lecture and in the next lecture, we will be looking into more details of inner block fusion and that would be all for this lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture 37
Kernel Fusion, Thread and Block Coarsening (Contd.)

(Refer Slide Time: 00:30)

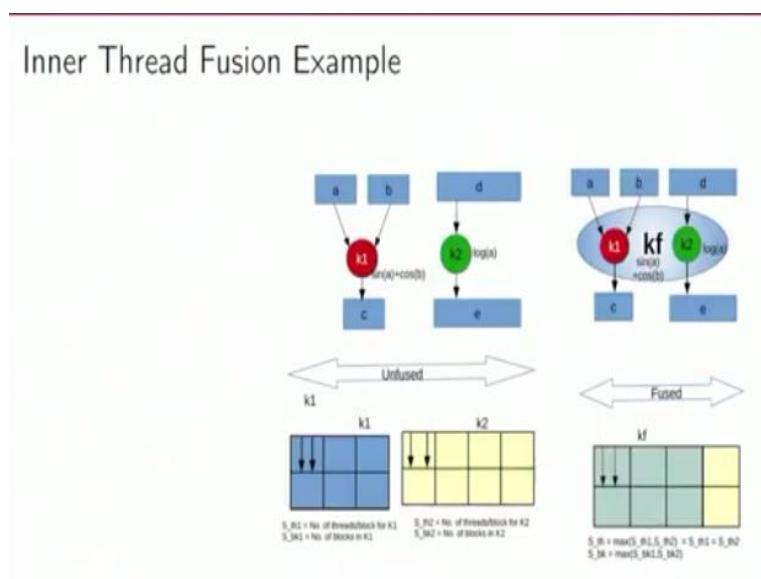


Hi, welcome back to the lectures on GPU Architectures and Programming. If you just recall, in the last lecture, we started with optimization that is how to fuse multiple kernels together and we covered the idea of inner thread fusion, one possible way to do the fusion of two kernels, and we figured out what is the relative advantage and disadvantage of doing inner thread fusion, when it is applicable and all that.

With that background, we will just move to the next possible way of doing fusion, which is inner block fusion. So the idea of inner thread fusion was very simple. We were just increasing the per thread activity by making the thread do work for two or maybe some multiple data points. So if you just look into the example that we had.

(Refer Slide Time: 01:19)

Inner Thread Fusion Example



So this was our sample program for inner thread fusion. We applied it on independent kernels in two cases, one was same number of blocks and same number of threads per block and then we said that okay, let us allow it to have different number of blocks for the two kernels while we keep the number of threads per block same and then we generalize the concept.

So essentially we define these two operations that for the fused kernel, the number of threads would be the max of the original number of threads per block and original kernel in the number of blocks would again be the max of the total number of blocks in the two kernels and we made each of the threads when they are launched for the fused kernel to perform the activities of both the original kernels.

(Refer Slide Time: 02:11)

Inner Thread Fusion

Fusion of independent kernels with different data size but same thread/block size:

```
//Fused kernel
//Let n2>n1
kf(a, b, c, d, e, n1, n2):
i = global threadId
if(i<n1)
    c[i]=sin(a[i])+cos(b[i])
    e[i]=log(d[i])
else if(i<n2)
    e[i]=log(d[i])
```

So this was the sample code. So ideally these two lines were the activities of kernel 1 and this line was the activity of kernel 2. Now in the fused kernel, they are both together or part of a single thread activity, based on whether they satisfy the requirements of whether they are actually varied operations for both of the kernels or not. Now coming here, we take different stands.

What we do is, okay, we do not fuse the kernels at that granularity of threads, but we start looking into the fusion at the granularity of a block. So what we do? If you are given two threads, you increase the block size of a fused kernel and you say that okay my number of threads in the block should be such that some of them will be doing the activity for kernel 1 and some of the threads in the block, we will be doing the activity for kernel 2.

So considering that these shaded areas represent activities for the individual kernels, when I am fusing them, this is my space containing the different launch threads and this granularity is showing the blocks now in the new fused kernel, where essentially this represents the original block size for k1 or kernel 1 and this is for k2, kernel 2 and they together form the new block for the fused kernel.

So we will just compute a global thread ID here, sorry a local thread ID here, that is I do not need to know what is the global thread ID, that is the ordering of the thread in a linearized

manner, but we are just interested in the offset or local thread ID with respect to this block. So I just look at what is the local thread ID. So just to recall how do I compute the local thread ID, so you will have the parameters dot x, dot y, dot z.

And you will like to use them with suitable multiplication factors in the x dimension, y dimension and z dimension and compute local thread ID here. So my point is the local thread ID does not represent the exact location of the thread in a linearized manner in the entire grid, but it just represents the linearization of the threads with respect to their ordering inside one block.

So with that, once we compute the local thread ID and then, we figure out that, okay what is the position of this thread inside this block. We decide that okay, whether to put the local threads with the threads with smaller local IDs to perform the activity for kernel 1 or kernel 2 and then we write the code and we will put in the activities like this. Suppose for kernel 1 the number of threads per block is x.

So for the fused kernel, we will just figure out whether the local thread ID is strictly less than this x or not and accordingly we will make the thread to do the job for kernel 1. If the local thread ID is equal to x or greater than x, then we will make it do the job for kernel 2. So these all things will work. We will have the work here for kernel 1 and then we will have the work here for kernel 2.

Just to note here and the difference earlier, the same thread was doing the job of both kernels. So we are doing a thread level fusion; inner thread fusion, but here we are doing a block level fusion. So we are adding more threads into the fused blocks, computing the local thread in the block that is the relative position of the thread inside the block. Using this relative position or local thread ID, we are computing whether this thread is supposed to do the activity of kernel 1 or kernel 2 and accordingly we are making it execute the code for kernel 1 or the code for kernel 2.

(Refer Slide Time: 06:33)

Inner Block Fusion

- ▶ Distribute computation of two different kernels among threads in single block
- ▶ For independent kernels with small block size(threads/block)
- ▶ Let, $S_{th,i}$ represents the number of threads in a thread block;
 $S_{blk,i}$ represents the number of blocks in kernel i($i = 1, 2$)
- ▶ For fused kernel -
 - ▶ $S_{th} = \text{sum}(S_{th,1}, S_{th,2})$
 - ▶ $S_{blk} = \max(S_{blk,1}, S_{blk,2})$
- ▶ Not suitable if -
 - ▶ S_{th} of fused kernel exceed upper bound of threads/block
 - ▶ kernels with synchronization statement



So if we just summarize it, essentially we will distribute the computation of two different kernels among the threads in a single block. For independent kernels with small block size or threads per block, this is the good idea. Why, because in case the sum of the threads per block for the kernels add up and cross the upper bound, then I cannot really do any inner block fusion. So this is a good idea for independent kernels with small block size.

So like earlier, let $S_{th,i}$ represent the number of threads in a thread block and S_{blk} represents the number of blocks in the kernel, where i is either 1 or 2 representing kernel 1 or kernel 2. Of course, I can generalize; I can have fusion of more than two kernels. So this is just a candidate example where I am showing how I will fuse two kernels. So for the fused kernel, we will have the total number of threads per block as the sum of these two.

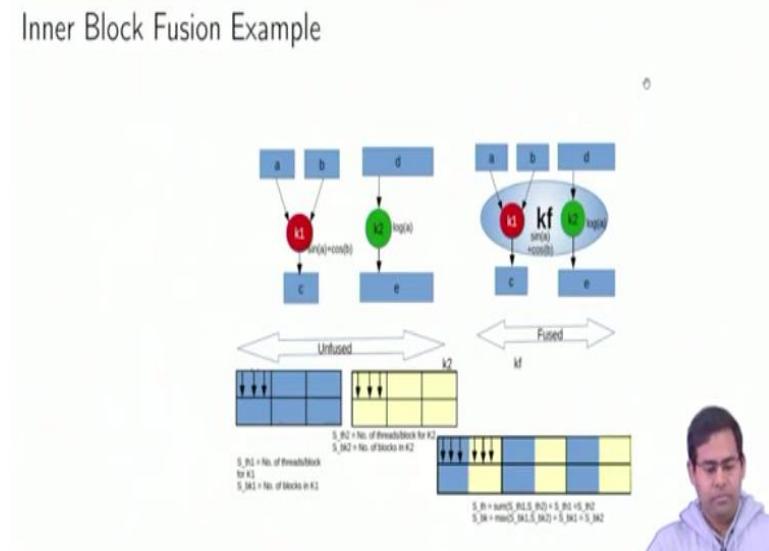
Because I will delegate some of the threads to do the activity for kernel 1 and I will delegate some of the threads to do the activity for kernel 2, like that and what should be the block size. First of all, see I am widening the blocks. I am not interested in increasing the number of blocks. So the block size should not be any sum or things like that, as is done with the number of threads, but the block size is rather the maximum of the block size, sorry.

The number of blocks is rather the maximum of the number blocks for each of the two kernels. So this seems, I have to cover both the kernels. So my number of blocks has to be the max of

$S_{blk,1}$ and $S_{blk,2}$. So when does this not work in general? Of course, if I am trying to fuse two kernels, where each of the kernels have lot of threads per block and the sum of $S_{th,1}$ and $S_{th,2}$ is greater than 1024, the upper bound and maybe for the higher level of architecture that would be 2048.

So whatever is the upper bound for the GPU, if it crosses the sum crosses the upper bound, then this is not working. So apart from the issue with this upper bound, the other issue is that kernels with synchronization statement, for them also the idea of inner block fusion would not be suitable, simply for the reason that a synchronization statement will force synchronization for all the threads in a block and it cannot work for a subset of threads, which will be part of a block.

(Refer Slide Time: 09:18)



So let us come to the example of doing inner block fusion of independent kernels. So we go back to our original examples of k1 and k2, fine. So when we are fusing these two, as you can see that for k1, the operation was $\sin a + \cos b$ and we are storing the value in the buffer c. For k2, the operation is $\log 2$ and it is stored in the buffer e and when I am fusing them, all we will do is, we will first define the number of threads and the number of threads per block, and the number of blocks.

So we will follow the method we discussed earlier. So we will increase the block sizes now by making it the sum of original block sizes and the number of blocks will simply be the maximum

of the original two. So the original two kernels are 6 blocks, so our fused kernel will also have 6 blocks and inside these blocks, I just have the double number of threads. Half of the threads are doing the job for kernel 1 and half of the threads are doing the job for kernel 2, as simple as that.

And which threads are doing the job for which kernel is demarcated here using the different colors. So the blue marked threads are doing the job for kernel 1, the yellow marked threads are doing the job for kernel 2, which also is marked in yellow, like that.

(Refer Slide Time: 10:46)

Inner Block Fusion Example

Fusion of independent kernels with same dataspace size

```
//Unfused kernels
k1(a, b, c, n):
    i = global threadId
    c[i]=sin(a[i])+cos(b[i])

k2(d, e, n) :
    i = global threadId
    e[i]=log(d[i])

//Fused kernel
//S_th1 = No. of threads/block for K1
//S_th2 = No. of threads/block for K2
//S_th = sum(S_th1,S_th2)
```

So looking into the program example, so these are the unfused kernels, k1, this is the k2.

(Refer Slide Time: 10:56)

Inner Block Fusion Example

```
kf(a, b, c, d, e, n):
    b = blockIdx
    t = threadIdx

    if(t<S_th1)
        c[t]=sin(a[t])+cos(b[t])
    else if(t<S_th)
        e[t-S_th1]=log(d[t-S_th1])
```

So when I am going to fuse these kernels using inner block fusion, so this is the pseudocode of my fused kernel. So first thing I will do is, I will compute the block ID and then I will compute the local thread ID. I will check whether this local thread ID. Again, let us remember, this is the local thread ID, so the offset inside the block. This is not the global thread ID. So we will check whether this local thread ID is less than the value at Sth1.

That would mean that this thread is supposed to do the job for kernel 1. Otherwise, we will check whether it is inside the boundary of the total number of threads per block for the fused kernel, because if this is not satisfied, then either the value is out of bound or the value is for the second kernel's work activity and accordingly the illustrative statement would fire.

Now let us just take another example. So here we have a bit of unbalance in terms of the number of blocks. So kernel 1 has got four blocks, kernel 2 has got five blocks, sorry six blocks. Kernel 1 has two threads per block and kernel 2 has got three threads per block that would mean in the fused kernel, we are going to have five threads per block and the number of blocks would be the max, which is 6 here.

So again we are showing that which thread is going to do the job of which kernel. We are again showing that using the highlighting scheme we discussed earlier. Observe that in the last block, what is happening, essentially in this picture, remember one thing, although I am not showing the arrows in the other blocks, I hope you understand that the threads are present here. So essentially, this pattern is replicating here, here, here, like that.

So just to avoid any confusion, let me just say just like this, this entire pattern is also present here, is present here, is present here, but observe that I am not going to have activity for kernel 1 here. So in this portion, there would be no activity for kernel 1. So what are you just have is only the k2 threads. Here I have threads for everybody, similarly here, but here I do not have any activity.

I hope you understand that as the data arrangement of the kernels become a bit different while fusing them, we are introducing an absence of balance, because when these blocks are going to

execute, there will be unused blocks, when waps will be formed. So essentially, we will be losing on the occupancy of the GPU and we will be not extracting that much parallelism. So whenever we are working with blocks or threads, which are widely varying in terms of their arrangement of data, the fusion operation may not be a good choice.

(Refer Slide Time: 14:22)

Inner Block Fusion Example

Fusion of independent kernels with different dataspace size

```
//Unfused kernels
k1(a, b, c, n1):
    i = global threadId
    c[i]=sin(a[i])+cos(b[i])

k2(d, e, n2) :
    i = global threadId
    e[i]=log(d[i])

//Fused kernel
//S_th = sum(S_th1,S_th2)
//S_bk = max(S_bk1,S_bk2)
//Let S_th2 > S_th1
//Let S_bk2 > S_bk1
```

So how do I handle the fusion of independent kernels, if the data space size is different, like this. Again, we have different number of threads per block here and also we have different number of blocks here. For that, let us understand that the code is going to be a bit different. Now these are the unfused kernels.

(Refer Slide Time: 14:50)

Inner Block Fusion Example

```
kf(a, b, c, d, e, n1, n2, X):
    b = blockId
    t = threadId
    if(b<S_bk)
        if(t<S_th1 AND b<S_bk1)
            c[t]=sin(a[t])+cos(b[t])
        else if(t<S_th)
            e[t-S_th1]=log(d[t-S_th1])
```

So when we are going to fuse these kernels, the first part is again same. So I have the block ID and I am going to have the local thread ID computed using the threadIdx and blockIdx parameters, that is of course fine. The next thing is, first the thread has to figure out that what is its block ID. Now why is this important? Because if the block ID is beyond the max, then of course, there is nothing to be done.

So just to remember, S_{bk} here is the number of blocks for the fused kernel. $S_{th,1}$ and $S_{th,2}$ are the threads per block setting of kernel 1 and kernel 2. $S_{bk,1}$ and $S_{bk,2}$ are the number of blocks for kernel 1 and kernel 2. So we will figure out whether I am really having to do some valid computation. So whether the block ID is less than S_{bk} . If so, then you get inside and figure out whether you have some job. This thread is going to do some job for the kernel 1 or kernel 2.

So the check would be that the thread local ID, whether that local ID of the thread is less than $S_{th,1}$ and whether the block ID of the thread is less than $S_{bk,1}$. Now just to understand that why do I need this, because earlier if you see when we are doing, I am just hoping that to the code and inner block fusion for kernels with identical data arrangement. So we just were computing the local thread ID and we are checking whether the thread ID is going to do the work for k1 or k2.

But now it is different, because we have a varying amount of number of blocks. So I should not only check for a given thread that whether it is part of its local thread ID is inside the data space for kernel 1 or it is inside the data space, is beyond the data space for kernel 1, but inside the data space for kernel 2. Only doing that check is not going to be valid. I additionally need to check whether the block of the thread is a valid block ID for k1 or a valid block ID for k2.

For example, if the block ID is this one, 012, so then in the x dimension of course, then I can understand that even if the thread ID is inside the thread boundary for kernel 1, kernel 1 does not have any computation for this block ID. So we will have a check with $S_{th,1}$ as well as $S_{bk,1}$ and then only do the computation for kernel 1. Otherwise, we know that okay there is no work for the kernel 1 and we will just check whether there is inside the boundary for the total number of threads per block. If so, there should be some activity for kernel 2.

(Refer Slide Time: 17:39)

Inner Block Fusion Limitation

Upper bound for threads/block is architecture dependent

- ▶ S_{th} of fused kernel must not exceed this upper bound

CUDA does not support synchronization for partial threads in a block

- ▶ kernels with sync() statement like reduction kernel not applicable for this type of fusion

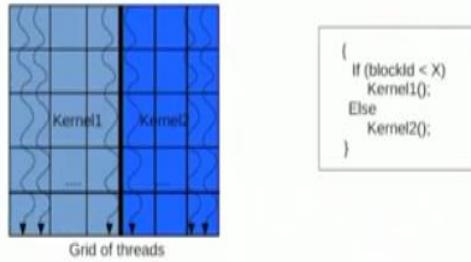
Now, coming to some of the important points like, as we have discussed earlier that total number of threads per block for fused kernel should not exceed the upper bound. We have to remember this, because the threads per block upper boundary architecture dependent and so if you are trying to write a parameterized code, like whether you should really have, I mean, suppose you are trying to write a code, where you are trying to generate a fusion of the kernels based on the GPU.

So you may first query the CUDA device property, figure out the allowed threads per block and then you can do the sum of the setting of threads per block for k1 and k2 and then decide in the runtime whether to execute a fused version of the kernel or not. So that is something which can be done, like, I mean you can actually decide on what kind of threading should you use, whether to fuse or not and also, this was the other important point we were discussing.

Like, CUDA does not support synchronization for partial threads in a block. So if we have synch or __syncthread statements in the reduction kernels, one of the components, then this kind of fusion is not a good idea.

(Refer Slide Time: 18:59)

Inter block Fusion



So moving aside into the other situation, which is inter block fusion, so we will need to first understand what is this idea of inter block fusion. So the first thing we did was, I am just recalling back. We increased part thread activity, so that was enough thread fusion. Then, we increased the number of threads per block and decided whether the thread will do the job of kernel 1 or do the job of kernel 2. That was inner block fusion. So I am fusing blocks.

Now we are saying that okay, let us not do that. Let us not fuse threads or let us not fuse blocks, but let us just fuse the entire arrangements of blocks together and launch a kernel. So that just means that you do not disturb the internal structure of the data space of kernel 1 or kernel 2. So you just define a kernel with number of blocks being the sum of the original number of blocks of k1 and the number of blocks for k2 and then, you let some of the blocks do the job for kernel 1 and some of the blocks to do the job for kernel 2.

So that is a much more coarse grain fusion. All you are doing is, you are just fusing at a coarser grain. So essentially we are executing kernel 1 and kernel 2, but concurrently. Instead of not dispatching kernel 1 followed by kernel 2, you are dispatching both of them together as a fused kernel, but nothing changes in terms of their internal coding structure, you just delegate some of the blocks to do the job for kernel 1 and then you delegate the rest of the blocks to do the job for kernel 2.

So this is how it goes on. So you have the code for kernel 1, you have the code for kernel 2. You just take the block ID, you check whether it is a valid block ID for kernel 1, it is inside the number of blocks boundary for kernel 1, otherwise you execute kernel 2. So for kernel 1, you have progress of threads like this. For kernel 2, you have another alternate progress of threads like this and you have a boundary here and this is the entire grid of threads.

So this denotes, maybe a theoretical way for you to think, that this is x. So if you are on this side, you are executing kernel 1, for example this side, you are executing kernel 2. Off course, this is a 2D picture. Overall, this would be the summary of inter block fusion. You distribute the computation of two different kernels among different blocks, for independent kernels with similar computation time, this is a good idea. Now this is important.

Why do you say, first of all, we understand the kernels have to be independent, otherwise as we have discussed even in the case of inner thread fusion, if we are trying to fuse dependent kernels, you have to bring in synch thread statements. So that, you exactly know when the threads for one kernel finish, other kernel finish, and then you do some computation, so that is quite tricky thing to do and it may not actually increase your parallelism that you can extract from the architecture.

So overall, we are always keeping ourselves restricted to the domain of independent kernels and not going to think of using dependent kernels. Then also, we figured out that okay, the kernels should be fusing, there should be suitable amount of load balancing, the data size or the data arrangement should not be too much different. Again the similar thing will also hold here that we will be fusing independent kernels, but the kernels, it should not have that they have this similarity in terms of computation time.

Let us look at the situation why is it so. So then, while some of threads we are executing this, let us kernel 1 is a heavy kernel and kernel 2 is a light weight kernel. So when you launch the threads, the threads executing kernel 1 finish this much faster, but these threads are still waiting. So that does not make sense then, because fusion of these threads are not making sense, because you have got these threads, which are finished.

But waiting at the end of the execution, while the threads have not ended their execution, so entire kernel is still live in the GPU, while some of the threads are still waiting. So then you are really not exploring the parallelism in the nice way, because otherwise you could have done a fusion of these workloads with some other workloads and that would have really led to a bit more balanced execution, because overall what is your target?

Your target is to increase the occupancy of the GPU and keep on using all the architectural elements, the compute units, the memory elements, the memory bandwidth in parallel as much with as much high throughput as possible. So if the independent kernels have similar computation time, then this is a good idea, otherwise not. So for our case, let us understand what should be our options for fusion.

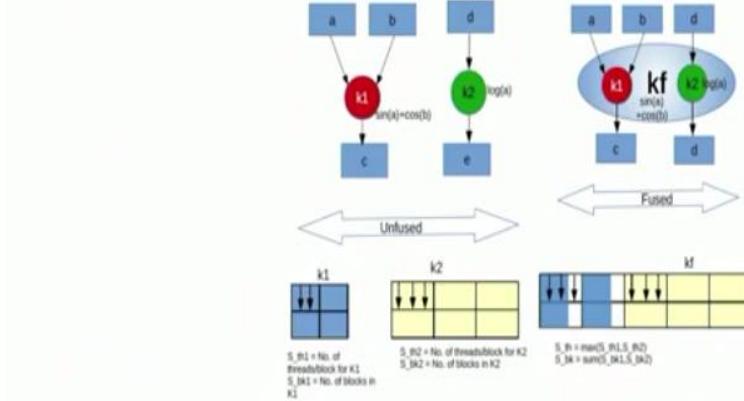
So let $S_{th,i}$ represent the number of threads in a thread block and $S_{bk,i}$ represent the number of blocks in a kernel. So when I am fusing them, so S_{th} would be the maximum of $S_{th,1}$ and $S_{th,2}$. So when I am fusing kernels, S_{th} would be the maximum of $S_{th,1}$ and $S_{th,2}$, because I am not going to increase part thread activity. All I am doing is, I am increasing the number of blocks. So the number of threads per block remains the same.

So it should just be the max of the originals and the number of blocks will be the sum of the number of blocks in each of the constituent kernels. So again, just to remember that this is not a good idea, if workload for different thread blocks differ a lot. That means, the kernels differ a lot in terms of their computation ahead it. So there are nice interesting points to take. For example, consider that kernel 1 is of compute intensive kernel, kernel 2 is a memory intensive kernel, is it a good idea to fuse them?

These are foods for thought, you can actually look into, you can try simulating the example of two such kernels. Consider induction kernel, consider some other kernel together, maybe a convolution kernel and think whether fusing them makes sense or not.

(Refer Slide Time: 25:34)

Inter Block Fusion Example



Now if we come to the inter block fusion example, we are again restricting ourselves to the original kernels. So we have the unfused versions on the left and the fused versions on the right. So as you can see, again we are considering the general case, where we have unbalance in terms of the number of threads per block and also unbalance in the terms of number of blocks, but now things are quite easy, all we do is, we increase the number of blocks and we delegate some of the blocks exclusively for activities of kernel 1.

And we delegate some of the blocks exclusively for the activities of kernel 2, so that would mean, we will now have, since in the original kernel there were two threads per block k1 and there were two threads per block for k2. In this fused kernel, we are having three sets per block in general. So when we are fusing them, first of all you see that, this is essentially nothing but k1's working area. So all the threads belong to this area are going to do the job for k1.

The threads, which are belonging to this area, that means or correct me, the blocks which belong to this area, they are resident threads. We will be doing exactly the computation for k2. So just to recall that, all we are doing is we are figuring out the block ID. If the block IDs belong to k1's block IDs, then they are going to do the activities for k1. For example, if this is the block ID 00, this is 01, like that, or this is 10, this is 11, those are the block IDs for which I will have k1 working.

For the other block IDs, I will have k2 working. So whatever was 00 here, if the block ID is greater than k1's block ID, just immediately greater than k1's block ID, for that it should be this block of k2 and so on, so forth. These are essentially nothing but k2's blocks and these are essentially nothing but k1's block, but we discussed the point of loading balance in terms of the computational natures of the kernels, whether k1 is a heavy kernel or k1 is a light kernel and k2 is a heavy kernel, but also observe the other point, we mentioned here.

So first was whether the kernels have similar computation time, that was the loading balance issue, also observe that here when we are looking into this kernel k1, the number of threads per block is now max, max of k1 and k2. So we have three threads for k1 as well as three threads for block for k2. But since k1 is originally utilizing two threads, so this third when it is belonging to the block IDs for k1 does not have much to do.

So these are just threads which are getting launched without any activity. So this is also a wastage of thread, which is running in parallel or it has got no activity. So if we look into the program example here, the fusion of independent kernels we are considering here. The fusion style is inter block fusion, but we are considering for different data space size. So these are my original unfused kernels.

So, of course, first we figure out what is the S_{th} and S_{bk} . So S_{th} is max and S_{bk} is sum. So we also have similar comments for the previous kernels. They are kind of repeated here. So I am just talking about them here, where it is a bit more complicated. Assume that $S_{th,2}$ is greater than $S_{th,1}$ and $S_{bk,2}$ is greater than $S_{bk,1}$, of course, you have to write suitable programs for that.

(Refer Slide Time: 29:32)

Inter Block Fusion Example

```
kf(a, b, c, d, e, n1, n2):
    b = blockId
    t = threadId
    i = global threadId

    if(b < S_bk1)
        if(t < S_th1)
            c[t] = sin(a[t]) + cos(b[t])
    else if(b < S_bk)
        if(t < S_th2)
            e[t] = log(d[t])
```

Now when I have this fused kernel, first thing I will do is, I will figure out what is the block ID. I will figure out what is the thread ID and I will also figure out the global thread ID and then we will use the block ID first to figure out whether this thread belongs to a block, which is for kernel 1 or kernel 2. So coming back here, again they will just repeat. So for the fused kernel, we have figured out what is S_{th} and S_{bk} and we are just considering.

The code is written in such a way that we are assuming that $S_{th,2}$ is greater than $S_{th,1}$. So of course, you can understand if for your second kernel, we are assuming that the number of threads per block is larger. If it is not so, just you have to switch the variant of the code. I hope that is clear. Here, the code is written assuming between k1 and k2, I have this property, otherwise your code has to be changed. I thought you should be able to do that.

So first you check, whether the block ID is less than $S_{bk,1}$. If so, then you get inside and check that whether inside this block, you have some activity for the kernel 1 or not, because if so, you just check whether the local thread ID is less than $S_{th,1}$, then you execute this code. Do I have an example where this will not execute? Yes, for example if you come here, consider a thread this one. For this thread, the block ID is less than $S_{bk,1}$. So it will execute kernel 1's code.

But now, the thread ID is not less than $S_{th,1}$. So it will fail the if case and it will not do any activity. So for the threads with the previous IDs, they will get inside this part and do the kernel

1's activities. Those which will fail this, they are the idle threads. They do not have anything to do. Otherwise, we will just check whether the block ID is beyond $S_{bk,1}$, but inside S_{bk} , then I know that this thread belongs to kernel 2.

Again, we will just have the check whether it is less than $S_{th,2}$, the thread ID. In that case, we will delegate the activity for the sets to kernel 2's activity. Now this inter block fusion also has limitation. So what are the limitations? So workload of different thread blocks, consider that, I mean, they may differ a lot. We are just considering the situation that suppose the workload of different thread blocks will differ a lot, then this may not be a good idea.

So that was what we are speaking about, like if the kernels have different computation time and also internally, if the threads per block differ a lot. If the threads per block differ a lot, then we will have a lot of idle threads like this.

(Refer Slide Time: 32:33)

Inter Block Fusion Limitation

Workload of different thread blocks differs a lot. For example, below two kernels are not suitable for this type of fusion.

```
k1(a, b, c, n1):
    i = global threadId
    c[i]=sqrt(sin(a[i])+cos(b[i]))
k2(d, e, f, n2) :
    i = global threadId
    f[i]=d[i]+e[i]
```



And if the activity of the block, the amount of all thread to differ a lot, then some of the thread blocks will finish, but some of the thread blocks will wait and that will again create a load imbalance. So if you just check here, for k1 and k2, for k1 I have this is a heavy kernel, because all it is doing is, it is doing a square root of sin a + cos b in a part thread manner. So it is going to actually use the special function units in the SM.

Hope you remember the special function units, which we are actually going to do the transfer data functions, the operations there. So they will actually compute the sin, the trigonometric series, the mathematical expressions, the sin trigonometric series approximated for cos and finally with the sum, it will implement Sqrd operation. So these are highly costly mathematical operations, which are going on, whereas here I have a simple single cycle addition.

So these kernels are highly imbalanced with respect to their load. Due to this absence of workload balancing, they are not good candidate for fusion, because fusing them I do not gain in terms of concurrent execution of the kernels. Because one kernel finishes much faster, the blocks from that kernel, the other kernel's blocks are still executing. So there is not much of concurrence.

(Refer Slide Time: 34:03)

GPU Optimization techniques

Some examples of GPU Optimization techniques-

- ▶ Reduction
- ▶ Fusion
- ▶ Coarsening

So with this, we will end this lecture and in the next lecture, we will start with another optimization technique, which is thread coarsening. Thank you for your attention.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture 38
Kernel Fusion, Thread and Block Coarsening (Contd.)

Hi, welcome back to the lectures on GPU Architectures and Programming. So in the previous lectures, we have been discussing certain GPU optimizations.

(Refer Slide Time: 00:32)

GPU Optimization techniques

Some examples of GPU Optimization techniques-

- ▶ Reduction
- ▶ Fusion
- ▶ Coarsening

GPU based parallel program optimizations, like reduction, like fusion, and we will now start with our last discussion on program optimizations in GPU more specifically CUDA based program optimizations and this would be the idea of thread coarsening.

(Refer Slide Time: 00:48)

Thread Coarsening

- ▶ Parallel execution sometimes requires doing -
 - ▶ Redundant memory accesses
 - ▶ Redundant calculations
- ▶ Merging multiple threads into one
- ▶ To allow re-use of result, avoiding redundant work
- ▶ Similar to loop unrolling, but applied across parallel threads rather than across serial loop iterations.



So speaking in layman's terms, so when we are doing parallel execution of threads, they sometime require to do redundant memory accesses and we also require to do redundant calculations and also we often launch multiple threads and the threads are pretty much light weight and launching so many threads in such cases do not make sense, because launching a thread also means creating a threads context in the hardware and keeping track of that thread.

And if for that thread, I do not really have too much of activity, then simply the launch of the thread does not make sense. So the idea of thread coarsening is that you do some kind of load balancing, that means you launch significant number of threads, but also make every thread bear some cost. That means make every thread do some meaningful activity and also the threads should make re-use of results and avoid redundant work.

What we mean here is, when we are doing a GPU based computation, every thread has got its own par thread private local memory. Every thread gets its part of share in the register file. So these are things which we have discussed earlier also, like let us say we are using an video GPU, so while you can actually set that statically that what should be, whether you are going to allow register speeding or such things to happen or not.

By default, they are prevented. So when you are going to launch your kernel, the system is automatically assigning a set of registers for par thread activity and that would also mean when

the thread is doing its computation, the results are available in the registers, which are fixed for that thread, but those are also the registers, which are concurrently running thread cannot see. Maybe it is doing some activity, for which it is not available.

But it may be the case that this adjoining threads, they have some locality in terms of computation. They do some activity, which could have been re-used. So if I make one thread to do activity in this case, for both threads together, then maybe some computations which the thread 1 has made can be used again by thread 1, if it is doing thread 2's activity without original thread 2 duplicating the same activity inside its own set of registers or the set of register that may allow that way.

We will make these things clear that how this re-use of results can happen and all that and how I can make threads have a redundant work. So the concept is quite similar to loop unrolling, but here instead of applying it on sequential execution of a thread and avoiding multiple executions of the loop, we are actually applying it across parallel threads rather than on the serial loop iterations of a single thread.

(Refer Slide Time: 04:18)

Effects of Thread Coarsening

Thread Coarsening results in reduction in parallelism.

- ▶ Beneficial effects:
 - ▶ Coarsening requires only less number of threads to be launched
 - ▶ Barrier execution is reduced
 - ▶ Increased number of instructions increases scope for exploiting hardware instruction-level parallelism

Now, off course, one important point we have to understand is that thread coarsening would result in a reduction in parallelism, but there are good effects of that also. First of all, why does it result in reduction in parallelism, because if I let one thread do the job of multiple threads, then

my thread block size can decrease or I can keep the thread block size same, but the oral activity can be done by a lesser number of blocks.

So if the total number of threads I launch is less than the number of threads that the GPU can execute concurrently, then there would be a reduction also in parallelism, but otherwise, what can happen is, if the original threads were very light weight, now each of the threads have got some significant activity. So anyway, if there is available parallelism, then the kernels would be launched and all the parallel execution units will be engaged.

So the beneficial effects, if we can summarize is that, the coarsening would require only less number of threads to be launched. So due to that, as we said that lot of redundant execution can be minimized. For example, if there are barriers for thread, now since I have less number of threads in total, the number of barrier execution is reduced, so that would also mean an overall gain in the end-to-end execution time of the kernel.

Since now I have more number of instructions to execute per thread, but that would also mean, that now I can exploit hardware level instruction level parallelizing, which maybe earlier for each of the threads being very light weight, there may not be too much parallelizing inside the threads.

(Refer Slide Time: 06:06)

Effects of Thread Coarsening

Thread Coarsening results in reduction in parallelism.

► Detrimental effects:

- Raises a kernel's resource consumption like registers, eventually resulting in reduced occupancy
(Occupancy is the ratio of the number of active threads per SM and maximum number of active threads allowed per SM)
- Also increase the pressure on the cache in some kernels

Now, this may also at certain points be detrimental. Of course, here we are just giving some placeholder definitions like we have said that okay, since the threads have own local share of register file, there may be redundant computation going on, if threads are light weight. So if we merge activities of threads to a single, then whatever was the computation by thread 1 and earlier it was done, maybe sometimes the thread 2 was unable to re-use it.

But since now one thread is doing the job of two parallel threads, it can make a better use of the register file. Those were the arguments that we are trying to make in terms of thread coarsening advantages. At the same time, if we try too much of that, it can also be disadvantageous. Why, because suppose I am now coarsening the threads making each thread do lot of activity, so then observe, this is the most important point here.

That when the program is getting compiled, the compiler is figuring out the par thread activity and it is allocating a part of the runtime system. The runtime system is going to allocate significant amount of part of the register file, now to par thread. So that is a finite resource. Register file, in total it is a finite resource, shared memory in totality is a finite resource. So when the runtime system is going to allocate it, it is figuring out that, okay, now the par thread register file usage has increased or the par block share memory usage has increased.

So since the resource increase par thread and par blocking effect has increased, that total number of thread that the system or more specifically parts streaming multi processor can execute will be reduced. Now this is a notion that is known as occupancy. So occupancy is the ratio of number of active threads per SM and maximum number of active threads allowed per SM. So let us say we have the number of maximum threads that are allowed per SM is 2048.

Let us say that is the maximum thread context that I can have and then it is not required that the occupancy that the SM will support in the runtime is same as that maximum limit. Why, because as you have figured out, that whenever I am launching the threads, the threads will be allocated local resources in terms of registered files. The blocks will be allocated local resources in terms of shared memory.

If I coarsen the threads, there will be point that these values will be significantly high, that is the par thread resource requirement is high. The hardware in that case will not really launch that number of blocks into the SM as is allowed for the SM, simply because that many number of thread blocks cannot execute parallel, because the lesser number or subset of them is good enough to completely engage the local shared memory and the register file resources.

So that would eventually result in a reduced occupancy value. This is a finer point, but I hope it is clear. We need to understand the SM can run as many threads as there are SPs and Sf is there and as many thread context that SM can remember, but at the same time, when it is running the threads, the threads will be occupying its own share of local resources. So once I increase that share of local resources, the hardware will not really run that many active threads, which it can and this ratio is the occupancy factor.

If I coarsen the threads too much, the occupancy factor is going to decrease. So that is the effect that the kernels increased resource consumption will create in terms of occupancy and that may be detrimental with respect to the overall kernels execution and the second thing is that if coarsening the threads or increasing the par thread activity, as we have discussed earlier, that if the par thread activity increase, I may be removing some redundancy in terms of thread computation.

Because earlier with two threads, we were not able to share their computations in terms of registers, I mean computed values to the registers, but now that can. At the same time, I am increasing cache activity. This would increase the cache pressure on the kernel and that is also a resource. Now at this point, I would like to also talk about something, like with respect to cache that is present in the kernel.

Earlier we have discussed that for in the GPUs, we have L1 and L2 cache. L2 is unified and L1 is present inside the SM. As a shared memory plus L1 setting, which can be configured. Now while that is true up to the Kepler case of architecture from Maxwell case of architecture, there are instances of GPUs with separate share on L1 caches.

So when the threads are going to do memory reads and writes, they are also consuming the resources that is cache and with par thread memory reads and writes increasing, the cache pressure also increases and that also plays a role in deciding what is the total speed up gain by thread coarsening and with the too much of coarsening, there may be increased cache pressure, there may be increased latency of the instruction execution, due to this cache pressure and that would be a detrimental effect.

It may finally lead to a loss of speed up. So the point we are trying to make here is for every kernel, it is not the cache that you keep on coarsening the threads, reducing the number of threads as much as you want and because if I think it on nice way, if I give more activity per thread, well then I would have less number of, I am doing more amount of work per thread, so I am really reducing the total grid size for the kernel, but that may not always be the case.

As we are finding, that there is a sweet spot with respect to what is a good coarsening factor. The optimum coarsening factor should be such that it will make maximal use of the GPU parallelism. We have to understand that the parallelism is in length and breadth, because I have a set of resources. I want to consume these resources that is the execution resources and the memory resources as much as possible in parallel, but also in length.

Because I want to use them as much as possible in parallel for as much time as possible. So there is this sweet spot in parallelism. So if we go too much with respect to coarsening, then finally it will lead to sequential execution and we will lose parallelism. So this sweet spot is something that as a programmer, you have to understand by looking at the activities and the algorithm.

(Refer Slide Time: 13:25)

Simple Example

Without coarsening:	With coarsening:
<pre>__global__ void square(int *g_idata, int * g_odata, unsigned int n) { unsigned int gid = threadIdx.x + blockDim.x*blockIdx.x; if(gid<n) g_odata[gid] = g_idata[gid] *g_idata[gid]; }</pre>	<pre>__global__ void square(int *g_idata, int * g_odata, unsigned int n) { unsigned int gid = threadIdx.x + blockDim.x*blockIdx.x; if(gid<n) { int tid0 = 2 * gid + 0; int tid1 = 2 * gid + 1; g_odata[tid0] = g_idata[tid0]*g_idata[tid0]; g_odata[tid1] = g_idata[tid1]*g_idata[tid1]; } }</pre>

So let us start with some simple examples of coarsening and we will get back to this issue of speed up that one can achieve with coarsening later one. First, let us start without coarsening code. So on the left hand side, we have the code which is without coarsening. As you can see, it is a simple program, which is just revealing what is the global thread ID and then for the content in the memory location for the thread ID, we are just executing a square operation.

All we do with coarsened version of the kernel is that make every thread do that operation for two consecutive locations. So the initial case is, you launch thread and they all perform squaring of the data and then in the modified version, this is the without coarsening case. So this is how, with half the threads, we are doing the activity and as you can see that first we find the global ID and we use this global ID, we just multiply it to create the offset at which it will work.

Because we know that now for all the previous thread IDs, the work from two data points. So we have to then bring in this multiplication factor by 2 and the next element, multiply by 2 and plus 1. So these are two consecutive locations. So for anywhere in between, you take the thread ID, compute the global ID and then go to the next consecutive location. So these are the two consecutive locations, for which we are going to do the work.

(Refer Slide Time: 16:01)

Outline of Technique

- ▶ Merge multiple threads so each resulting thread calculates multiple output elements
 - ▶ Perform the redundant work once
 - ▶ Save result into registers
 - ▶ Use register result to calculate multiple output elements

So if we just make a small outline of that technique, what we are doing is, we are merging multiple threads, so that each resulting thread calculates multiple output elements. So they perform the redundant work once. So why shall we really say that? Essentially you can see that each thread earlier, every thread would have some launch over it and they will do this computation.

Now we have reduced the number of thread context by 2, so that removes the product and we have given significant activity for each of the threads. For example, we can just say that when this has been fetched, the other data is also being used, is being computed in another register by the same thread. So the work of loading the data is being reduced and the results are getting saved into the register and use the register results to calculate multiple output elements.

While that is not exemplified here, but it is easy to understand that when we have the multiple threads merged, then suppose each thread was having some initial activity, now that I can do the initial activity together. So let us say the initial each of the threads had some common multiplication or some common constant evaluation to do. Now that is not specific to a thread ID or corresponding data space. If that is the case, then that is the definition of the redundant work.

Now that it would be done in a smaller number of times. That is point 1. Second point is intermediate results can be saved to the registers and they can be used for doing some

computation, which were originally done by the other threads and then the register results can be used to calculate multiple output elements, maybe let us consider the situation, that earlier there were some par thread activity and finally this par thread activities output, which are computed by which of the threads, were going to go through some transformation.

Now since two adjoining elements are already computed by a single thread, if that transformation was associative, then some part of it can be done here. I hope that I am making sense here. Suppose in the original kernel, whatever the outputs they are going to be computed, so let us say this is the final output of all the threads here, they are going to go through some transformation function f .

So let us say these values are a_1, a_2, \dots, a_n , so I am going to apply f on a_1 and then a_2 and then like that up to a_n . Now that many operations are required, but now inside this function only, I can compute $f(a_1, a_2)$ and similarly when these threads get done, then this job is some of it is already done here. So final label, I have to work with half of the values like considering this operation, this is fine here.

So just because each of the threads are now working on more data points, I am again making a lot of savings here with respect to achieving some more parallelism at the thread level, before the synchronization point. So we can understand that fundamentally what is happening, since I am increasing the par thread activity. I can reduce the redundant work, which each thread was doing as a common activity.

I can make use of the intermittent more number of registers to store intermediate values, which can be used and which can actually help me to calculate a final output value, which can actually help me to reduce a final step, which the threads are suppose to do, once each of them compute their output value and then synchronize and then perform. Now some of the value, if there is an associative operations, some of that can be performed, now when the threads are executing on different data points.

(Refer Slide Time: 20:31)

Outline of Technique

- ▶ Merged kernel code will use more registers
 - ▶ May reduce the number of threads allowed on single SM
 - ▶ Increased efficiency may outweigh reduced parallelism for a given hardware

So the point we are trying to make here is also that the marched kernel code will use more registers. Now what is the issue here, that this may reduce the number of threads allowed on single SM, like we were speaking earlier. This may reduce the occupancy. Now increased efficiency that we are getting may outwardly reduce parallelizing the given hardware. So this is an important point, reducing occupancy is not the bad thing.

So I may have reduced number of threads on the single SM, but since I am also having fine grain parallelizing inside each of the threads, I am having a reduction in the local computations of the threads due to lack of redundant activity. I am gaining by doing some parallel computation, on the outputs computed by the original threads now, due to the coarsened activity. Initially, the reduction in occupancy of the SM may not hurt, because it may be too more efficient.

Again I will repeat this part. Each thread is consuming more local resources. Due to that, up to some point, it does not hurt, because although they are consuming more resources, the SMs are full, where they are going to operate with the maximum number of threads they are allowed. After that point, if I coarsen more, then the SMs will execute a lesser number of threads that they are allowed to execute, but still this reduced parallelism may not be bad enough.

Because in spite of the reduced parallelism, inside each thread I am now exploiting some instruction level parallelism and I am doing some redundant work, less amount of time, but after

some point of time, it will start hurting the global execution view. So up to some coarsening factor, I may have increased efficiency even with reduced occupancy, that beyond a coarsening factor, the occupancy will go so much down that the reduced parallelism disadvantage will outweigh the effect of increased efficiency due to thread coarsening.

So again I will repeat up to some point, the efficiency factor in terms of coarsening that we gain in terms of efficient execution outweighs the reduced parallelism, but after that point it starts hurting. If we lost too much parallelism, then it will start hurting. So again to summarize, that up to some point, if you coarsen the threads, you may not lose occupancy, because still the SM is able to execute threads up to its maximal limit.

After some point, it will have reduced occupancy, reduced parallelism, but still the instruction level parallelism and increased efficiency will outweigh the reduction in parallelism, but again after some point the occupancy reduction is so high, that coarsening will be detrimental.

(Refer Slide Time: 23:37)

Register Tiling

With thread coarsening, computation from merged threads can now share registers.

Properties of registers:

- ▶ extremely fast (short latency)
- ▶ do not require memory access instructions (high throughput)
- ▶ private to each thread
- ▶ threads cannot share computation results or loaded memory data through registers



Something important here, that with thread coarsening, the computation of the merged threads starts sharing registers. This is something we have discussed earlier also, multiple times. So why is this a good thing. First of all, we all know that register file is extremely fast. So if I increase par thread activity without decreasing the occupancy significantly, then still it is a good thing, because if the thread brings more data and loads into its par thread available register file.

Then does not the threads while doing their local computation, they reduce execution of memory access instructions and in that way the gain in high throughput and this is a good thing, because earlier when the threads were not coarsened, each of the register file portions that were available to each of the threads at the lower ground level, they are private and suppose the thread 1 gets S1 of registers available to it, thread 2 gets S2 of registers available to it, so they cannot share the computation results, because they are private to their threads.

But now, when I coarsen the threads, the set of available registers may be $S1 + S2$. So the threads get more registers to work with and now they can share the intermediate results. They can decrease the number of memory access instructions in between. Why? Suppose earlier when some register was storing some local result and then for par thread activity, maybe the register size was not good enough, that is register spilling, by default it would be deactivated in the GPU.

So then, there will be some access to the local memory, sorry access to the global memory. Memory access instructions will be executed, but now if I have more number of registers available, it may happen, then due to this sharing of context registers, I have less number of accesses required. I can avoid the register spilling a bit and since the threads are able to share the computation results, they will make a more efficient utilization of the larger set of registers that would be available to them.

That will actually lead to the efficient computation, that we have been discussing in the previous slide.

(Refer Slide Time: 26:33)

Coarsening Factor

Coarsening Factor -

- ▶ is the number of times the body of the thread is replicated,
- ▶ gives the best performance depending on the program and on the hardware.

Coarsening Factor for any problem in a hardware can be decided -

- ▶ Through profiling
- ▶ Using static analysis



So the other thing, when we coarsen the threads, there has to be a choice of coarsening factor. So if we look into the example we give in terms of the program, so this is simply a coarsening factor of two. We are employing a coarsening factor of two. You are increasing the par thread activity by two. What is a good coarsening factor is also an important issue. The definition of coarsening factor is it is the number of times the body of the thread is replicated.

In that example, the replication is twice. Which coarsening factor gives the better performance, the best performance will precisely depends on the program and the hardware. Coarsening factor for any problem in the hardware can be decided in two ways. I mean, we can get an estimate of the coarsening factor by performing a static analysis of the program and the dynamic way would be to profile the program, you run the program with different possible coarsening factors.

Also for each run, you try different possible inputs of the program and you get the execution statistics and figure out, okay for what is the good coarsening deficient for this kernel. So just to summarize here, let us understand that for each state, which we increase the par thread activity, par thread memory requirement, thread starts sharing registers. In that case, they can store intermediate results inside the available registers.

This may reduce memory access for the global memory, because if it can make a good use of the larger set of registers, which are now available to it for sharing purpose and that would actually

ensure shorter latency for the memory accesses, because you have more data in the registers and you execute, if you do not require memory access instructions, then you can attain high throughput in terms of the overall kernels execution.

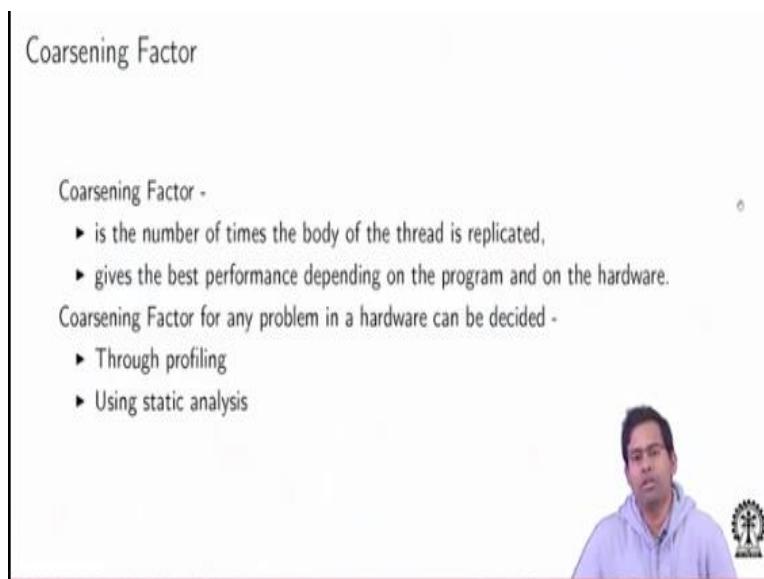
So this is the advantage. We are not going too deep here. In the next lecture, we will take some more examples, when we really take, we will see that how the coarsening factor affects the kernel's execution. With this, we will end this lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology - Kharagpur

Lecture – 39
Kernel Fusion, Thread and Block Coarsening (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming. So, if you remember in the last lecture we were discussing thread coarsening as a possible optimization for GPU programs.

(Refer Slide Time: 00:37)



Coarsening Factor

Coarsening Factor -

- ▶ is the number of times the body of the thread is replicated,
- ▶ gives the best performance depending on the program and on the hardware.

Coarsening Factor for any problem in a hardware can be decided -

- ▶ Through profiling
- ▶ Using static analysis

And in a short summary, we just discussed that what are the good and bad things of coarsening and we just tried to motivate why beyond a point coarsening threads too much may not be of help and in that regard, we define; we are trying to define rather that what is the notion of coarsening factor; essentially, by factor I mean that how much work I am going to allocate per thread over and above a base line implementation.

So, in that way a coarsening factor is the number of times I am going to replicate the body of a thread that means, the number of times I am increasing the per thread activity with respect to a base line implementation and what is interesting; is to figure out, what is the best performance, what is the coarsening factor that gives the best performing, performance depending on the program and on the hardware.

So, it is not something constant, it very much is a function of what program is under consideration and what is the target GPU and off course, I mean it is easy to understand why that would happen because as we have discussed earlier that whether coarsening is going to help beyond a point or not depends on the amount of architectural resources that are available in the hardware.

And up to some point, doing coarsening is fine that you are increasing thread activity without decreasing the occupancy of the hardware at some point, you are going to hurt the occupancy of the hardware but still coarsening may be good because you are utilizing the hardware resources more efficiently because each coarsen thread is making more efficient use of the hardware.

But beyond that point it may happen that the occupancy reduces so much that you lose out on the effective parallelism over the lifetime of the program and whatever is the best coarsening factor is I mean, deciding that statically is; I mean is one way that you can do a static program analysis to figure out a possible coarsening factor, it may not be the best but you can figure out a possible coarsening factor.

And also the other way would be that you try different coarsening factors and it is really an intricate problem because the architecture will have lot of parameters, the program can have lot of dependences, so figuring out the perfect coarsening factor, you can try doing that using a static analysis and you can without any guarantee that it is basically, the best coarsening factor.

The alternative can be that you try out different possible coarsening factors, profile each of the implementations, different coarsen implementations in the architecture, profile them, profile each of the coarsening factors for multiple possible input runs and create some speed-up analysis and figure out what is working best for you. Now, the static analysis based methods which can give you a good coarsening factor would be sound ones.

That means, they would give you, they will tell that well this coarsening is feasible to implement without hurting the functional equivalence of the program but it may not be able to say that whether that indeed is a best possible coarsening or not.

(Refer Slide Time: 03:49)

Types of Coarsening

- ▶ Thread-level coarsening: Increase granularity within a single block of threads
- ▶ Block-level coarsening: Increase granularity across multiple blocks



So, the different types of coarsening just like we discussed different types of techniques of fusion like what are the different possible ways in which you can implement fusion among GPU kernels, similar to that there are different possible ways in which you can coarsen a GPU kernel and basically, you have to increase the par thread activity now, that can be done at the thread level or at the block level.

So, when we say that you are doing a thread level coarsening that means, increased granularity within a single block of thread, so essentially your coarsening threads by giving them more activity per thread but those are all activities inside a single thread block, the other could be that you do block level coarsening that is you increase the granularity of coarsening across multiple blocks.

That means, when you thicken or coarsen a thread, you give it more activity not from the original block of thread but more activity from other blocks of thread. I hope this is clear let me just reiterate, so when I do thread level coarsening, I will coarsen a thread by giving it more activity but that activity was originally inside this specific thread block, inside who; which I am talking about the threads.

When I say block level coarsening that means, I am coarsening each of the threads inside the thread block by giving them extra activity which is not the activity of the original thread block but from some other thread block.

(Refer Slide Time: 05:19)

Thread-level Coarsening

- ▶ Applies coarsening at the level of individual threads
- ▶ Combine two or more threads from the same block
- ▶ Each thread block performs the same amount of work but with fewer threads
- ▶ But each SM has limitations in terms of registers, shared memory, and concurrently runnable thread blocks
- ▶ These hardware constraints bound how much to coarsen.



So, let us first discuss thread level coarsening in detail, so when I am applying the coarsening, I am applying it at the level of individual threads and so essentially, I am combining two or more threads from the same block, activities of two or more threads I am delegating it to one thread inside the block. So, each thread block now performs the same amount of work.

Because I have not delegated work from other thread blocks to threads inside this thread block, since I have not done that so now, each thread block performs the same amount of work but it is able to achieve that with coarsen threads which are fewer in number so, I am now decreasing the threads per block while doing the same functionality of the thread block.

But when we do this, each of these trimming multiprocessors have their limitations as we have discussed earlier in terms of the registers, the total register file size inside the SM that amount of shared memory inside this and concurrently, runnable thread blocks like how many concurrent thread contexts that SM can hold; so, these are the 2 limiting factors and that actually limits the total amount of; so total number of effective threads that will really be launched in the SM after I thicken or coarsen the threads.

Just to make the point clear so, every SM has an upper bound on the total number of concurrently runnable thread blocks, now it also has an upper bound on the size of the register in the shared memory. So, when I coarsen the threads, I increase the par thread demand of registers and share memory, so due to that the SM maybe also getting further limited in terms of the actual thread blocks that it can run concurrently.

And this hardware constraints will actually I mean, this is something we have already discussed, these hardware constraints will actually decide on how many threads I can really run that will affect the occupancy which in turn will give the bound on how much really I should apply the amount of thread level coarsening.

(Refer Slide Time: 07:27)

Stride Length

- ▶ Acts as an offset between the IDs of threads that are to be combined
- ▶ Max stride length \leq (Number of thread per block in the dimension where coarsening is applied)/ Coarsening Factor
- ▶ Minimum stride length \geq Warp size to ensure memory coalescing *transaction*

So, the next important thing is what is the stride length across which I should do the coarsening, so this acts as an offset between the IDs of threads that are to be combined so, when I am combining 2 threads what should be the offset between them. Now, the maximum stride when I allow, so first of all we are still talking about thread level coarsening but when I am coarsening threads, it is not necessary that I just coarsen tid with the amount of activities for the original tid and the tid plus 1.

But rather I am trying to coarsen the thread with tid, the thread ID tid with activities of original threads with a thread IDs being tid and tid plus s, where s is a stride length. So, this amount or this stride length can have some limits, first of all what should be the maximum stride length; it should be less than the number of threads per block in the dimension where the coarsening is applied divided by the coarsening factor.

So, let us try and understand what it means so, let us consider that this is the total thread arrangement and I am using a coarsening factor C equal to 2, so effectively I am going to use these many threads, I mean half of this number of threads okay, so maybe yeah, so let us say I

am going to use half of this number of threads and I am doing the coarsening in this dimension.

And so, when I am choosing the stride length so, the number of threads per block in this dimension, let it be some X , so divide by 2, so that would be the maximum number of threads; thread IDs in these dimension in the coarsen kernel. So, the maximum stride length for the threads has to be less than this; that is the limiting factor why because; of course, if it goes beyond these, then I have a problem.

Because the thread seating at the boundary of this coarsening boundary, they those thread IDs plus the stride value would shoot beyond the original thread block boundary. So, I if you just consider the number of threads per block in any specific dimension, in which we are applying the coarsening, so let us say this is the dimension we are applying the coarsening, you divided by the coarsening ding factor.

So, that would give you the total arrangement of threads that would be there in the coarsen kernel. Now, when I am talking about that stride length, I hope this is clear the stride length has to be limited by the original dimension in that original threads per block number in that dimension divided by the coarsening factor because if I consider a stride length, which is greater than this, then what happens?

If the thread which has the tid is sitting in this boundary, those plus the stride length would go beyond the thread dimension in this, in the original arrangement of the data. So, the maximum stride length will be limited by this equation but at the same time, what should be the minimum stride length? Now, just to make sure, that we do not want to disturb the original memory behaviour of the program.

So, let us say originally, I had a few tid's which are doing accents of some data sequentially, so that when these thread IDs they get packed inside a warp perform coalesced access of the data from this memory, I do not want to disturb this behaviour, so let us understand that if I make one thread, each of the threads in this warp, if I am trying to coarsen them by making this thread ID to the job of accessing this thread ID followed by the next let us say.

Again, the other thread ID followed by the next and so on so forth, then I may possibly lose out on the memory coalescing. So, when I am trying to coarsen each of the threads, I would like to have the minimum stride length to be greater than the warp size, so that whatever is the behaviour of the threads inside the warp, they do not lose out on their coalesced memory accesses.

But rather when the thread does its coarsened extra activity, it belongs to another separate coalesced global memory transaction, so those would belong to another separate global memory transaction like this. So, this is the original transaction of the threads, this other set of transactions were supposed to be let us say, this is one transaction, this is other transaction. I do not want to disturb this nice behaviour.

So, if I just make the strides greater than the warp size, then essentially I am not disturbing the original memory coalescing, whereas if I do something like I am accessing for each thread ID, I am accessing consecutive locations, that does not make real sense, because then in many cases it may happen that the original memory coalescing behaviour which was nicely distributed across the threads that may get into a problem.

So, I hope with some examples you can actually work this out that why we would like to keep the stride length greater than the warp size.

(Refer Slide Time: 14:30)

Thread-level Coarsening Example

```
__global__ void
thread_coarsened_reduce3(float *g_idata, float *g_odata, unsigned int n)
{
    //Apply thread coarsening factor 2 and stride 32
    // The coarsening factor dictates how many replicas of the local thread id and
    // global thread id will be in the program
    // Since we have coarsening 2, we will have two instances of tid (local thread
    // id) and i (global thread id)

    unsigned int tid0 = (threadIdx.x/32)*32*2 + threadIdx.x%32;
    unsigned int tid1 = tid0+32;
    unsigned int i0 = blockIdx.x*2*blockDim.x + tid0;
    unsigned int i1 = blockIdx.x*2*blockDim.x + tid1;
```



So, with this as a motivation let us try to look at a coarsened version of our reduction kernel in fact, if you remember that in a reduction kernel whenever we are trying to do access, we

are trying to ensure that the global memory transactions are not nicely coalesced and even the shared memory reads are nicely coalesced, without any kind of shared memory bank conflict.

So, we let us first go through this example, here we are trying to use a thread coarsening factor of 2 with the stride as 32 which is equal to the warp size now, this coarsening factor is dictating how many replicas of the local thread ID and global thread ID will be in the program. So, I hope this is clear so, we have to replicate this local and global thread IDs in the program.

So, since we have chosen a coarsening factor of 2, we need 2 instances of local thread IDs to access 2 consecutive data, 2 possible data values for a given global thread ID. So, let this tid be the local thread IDs and i is the global thread IDs, so we will need 2 instances of both of them here so, we compute this tid 0 and tid 1. So, that would actually give us this different local thread IDs as you can see that the local; the tid 0 is a standard local thread ID inside the block.

And we know this because we are just calculating the offset, so we are just assuming that it is all in the; were coarsening in the X dimension here, since were coarsening in the X dimension with this calculation we are finding out the offset in the local thread ID and then we are doing; since, we are choosing a stride of 32, so that would give me the second location from where I will try to do that thread level activity.

And then I compute i_0 and i_1 which tell me what is the global thread ID corresponding to this local thread IDs, so, it is just that I know already what is the block, so, with that block ID and all that I just will add tid 0 and tid 1, observe this multiplication factor of 2 in terms of the block dimension because now, the thing is I am going to access the corresponding data points and I am launching half of the threads.

So, that means that I am going to have half; half the value of effective block dimension while defining the thread blocks, so in order to get the suitable access patterns for the corresponding locations in the data, I will have this multiplication factor by 2 here in the blocks.

(Refer Slide Time: 17:33)

Thread-level Coarsening Example

```
// load shared mem
__shared__ float sdata[BLOCK_SIZE]
sdata[tid0] = (i0 < n) ? g_idata[i0] : 0;
sdata[tid1] = (i1 < n) ? g_idata[i1] : 0;
__syncthreads();

// do reduction in shared mem
for (unsigned int s=2*blockDim.x/2; s>0; s>>=1) //Note every instance of
    blockDim.x gets multiplied by the coarsening factor
{
    if (tid0 < s)
        sdata[tid0] += sdata[tid0 + s];
    if (tid1 < s)
        sdata[tid1] += sdata[tid1 + s];
    __syncthreads();
}
// Note in the for loop, sdata is updated by both tid0 and tid1
```



So, with this we are able to compute the i0 and i1 the positions for which I am going to do the global memory accesses for the respective data points. Now, so this is basically the reduction kernel with thread coarsening, so what we are trying to do is that first each of these thread ID; these global memory locations i0 and i1, we are just checking whether their valid locations for this kernel and then we are bringing them into a shared memory.

And once we bring them into the shared memory, we are doing the usual reduction step, now so as you can see this is the loop for the usual reduction step like we have discussed earlier. Now, just observe one simple thing like just like we have done it here since, we have a coarsening factor of 2, so we will have half of the original number of threads in the X dimension, so that is why wherever I have block dimension is getting multiplied by 2.

Similarly, here it is getting multiplied by 2, this divided by 2 is the original code semantics for the reduction kernel, and then inside we have the original code that you just do an addition in shared memory with a stride of s but now you are doing it for, so this is basically this strides of which of the for loop in reduction whereas, our choice of stride for coarsening is 32 which is already hard coded here, just to avoid the confusion.

So, this s is the effective strides we divided getting divided by 2 in each iteration of the standard reduction kernel, inside the for loop instead of having one if statement of a standard reduction kernel we have, these are coarsen kernel so, we have 2 if statements for 2 different locations that is why we have this common. That in the, for loop, the s data is updated by both the tid's, so by both tid 0 and tid 1.

(Refer Slide Time: 19:33)

Thread-level Coarsening Example

```
// only one write - the condition of the second will never be true
if (tid0 == 0)
    g_odata[blockIdx.x] = sdata[0];
if (tid1 == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```



And when we do the, only one of these can be 0, so the condition for the second the other will not never be true, so whichever is 0 for that we will be doing the output data calculation, I mean, loading back to the global data. So, with this we have an example of coarsening by 2 for the standard reduction kernel.

(Refer Slide Time: 20:03)

Block-level Coarsening

- ▶ Combines the work of several thread blocks into one block
- ▶ Number of threads per block remains unchanged so the number of executed thread blocks is reduced
- ▶ Each block has to handle an increased workload
- ▶ Resource requirements per block, in terms of register and shared memory usage, will typically increase



Now, let us just have a discussion on block level coarsening, so as we saw in thread level coarsening, we were doing the coarsening inside the same block and while for coarsening, we are making a choice of the stride, the stride should be that inside the coarsen kernel, the threads do not access locations beyond the thread block size; reduce thread block size so, it was the original block dimension in that coarsening dimension divided by the coarsening factor that was the maximum possible stride.

And the minimum stride that was definitely greater than the warp size, so that any original memory access coalescing pattern that was a good part of the original kernel should never get disturbed that is why we were not; we are always choosing a stride which was greater than the stride length now, coming to the other part which is the block level coarsening.

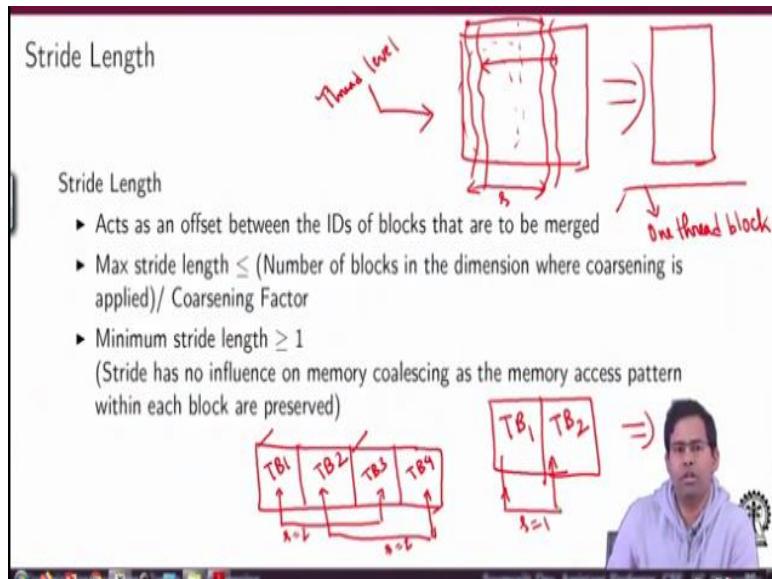
Now, we are I mean, what we are really going to do is; that we will essentially combine multiple thread blocks to one block that means, when I coarsen a thread, the thread should be doing its original activity additionally, it will be doing activity from some other thread from another block. So, effectively in the original thread level coarsening, what was happening is the thread number of threads per block was getting reduced due to that coarsened threads.

But here what will happen is the number of threads per block will remain unchanged but we will have the number of; effective number of blocks getting reduced. So, each block still has to handle and increase workload as you can see in the original case in the thread level coarsening, the power block activity remains same number of threads per block reduced threads per block got coarsened with original thread original thread blocks activities.

Here, the thread block sizes remains same, the number of; total number of blocks reduces by the coarsening factor since, the thread block sizes remain same and the threads gets coarsened so, each block handles the increased workload. Coming to resource requirements per block in terms of register and shared memory usage, such resource requirements will typically increase.

So, when we talk about resource requirements per block, then in terms of registers and shared memory usage, these requirements typically increase.

(Refer Slide Time: 22:38)



So, what is the choice of stride length, when we are talking about block level coarsening? So, now we are going to discuss the stride in terms of blocks, this acts as an offset between the IDs of blocks that are to be merged, as you can understand now, we are talking about bringing activities from different blocks together. So, if I am trying to draw a picture originally, when I was doing thread level coarsening, so if this was the original block, I was kind of coarsening threads inside the block by let us say this were the original warps for the threads.

So, now when I coarsen the thread, I am delegating this thread its original activity along with the activity here at a stride length, so by delegating activities at a stride length together to the original thread, we were coarsening and creating smaller thread blocks, but now when we are doing block level coarsening, so let us say I have a thread block let us say, this is 1 thread block; thread block 1, this is thread block 2, like that.

So, when I coarsen, I coarsen one thread here with activities from threads in the other block, now here when I do this, my stride length s is 1, now this is not general. So suppose, so here I am not going to threads but I am drawing thread blocks, so this is one thread block, this is a thread level coarsening. So, suppose these are 4 thread blocks and I pick up a thread from here and I give it activity of its original activity and some activity from the thread here.

And similarly, for other threads here getting the activities from this that thread block 3, since they are sitting at a distance of 2, so here the definition of stride length is the stride across blocks. So, with this we can just say that the maximum stride length can be discussed in a

similar way but now at the grid level. So, if I am coarsening in a specific dimension, so then the maximum stride length would be limited by the number of blocks in the dimension in which coarsening is applied divided by the coarsening factor.

So, just to observe the difference between thread level and block level coarsening, so when we are doing thread level coarsening, the maximum stride length was limited by number of threads in the dimension of the thread block, where coarsening was applied divided by the coarsening factor of the thread block. In this case, the maximum stride length is less than equal to the number of thread blocks in the dimension, number of thread blocks inside the grid of the kernel in that dimension where coarsening is applied divided by the coarsening factor.

So, if I say that in this dimension I am applying and the coarsening factor is 2 so, I will be merging this, and so I am; so, let us say there are 4 thread blocks here and the coarsening factor is 2, so maximum stride length is less than equal to 2 and when I operate with s equal to 2, I am operating at the maximum stride line possible, and then we have the idea of minimum stride length.

Now, of course that would be 1 because by definition of block level coarsening, I have to pick up threads, thread activity from another block, the nearest would be the next block, so the stride is greater than or equal to 1. Now, since the blocks we are now using the thread blocks for different thread blocks for selecting threads, so try doing have no influence on memory coalescing as the memory access pattern within the blocks.

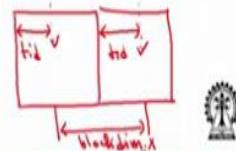
Or whatever is original memory access pattern inside the block that is always preserved, I mean it is quite easy to understand because when you are forming warps, you are forming warps with threads from inside a block, and since you are coarsening across blocks, the original warps they are packing their memory access patterns, their memory coalescing whatever was there in as a warp level activity, as a warp level coalescing due to global memory transactions, they do not really change.

(Refer Slide Time: 28:29)

Block-level Coarsening Example

```
__global__ void
block_coarsened_reduce3(float *g_idata, float *g_odata, unsigned int n)
{
    //Local Thread Id remains unchanged
    //The coarsening factor dictates how many replicas of global thread id will be
    //there in the program

    unsigned int tid = threadIdx.x;
    unsigned int i0 = 2*blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int i1 = (2*blockIdx.x+1)*blockDim.x + threadIdx.x;
    //Apply block coarsening factor 2 and stride 1
```



So, let us take an example of block level coarsening, so here your local thread IDs remain unchanged, the coarsening factor dictates how many replicas of the global thread ID will be there in the program, so observe the difference; earlier you replicated the global, local thread IDs and then you actually, replicated the global thread IDs but now, your local thread ID is same, you are using.

The reason the local thread ID is same is that the local thread ID gives you a specific offset inside the block, earlier, you were trying to access two different thread level activity inside a single block, so that would mean different possible offsets inside the single block, so you had computed 2 different local thread IDs and of course, their corresponding global thread IDs.

But now, your offset will remain same but for the same offset, you will be accessing different blocks so, for the same offset you are computing two different global thread IDs using two different blocks. Considering here that the stride is 1 and the coarsening factor is 2, since the coarsening factor is 2 in the X dimension, so you are multiplying blockIdx, blockDim x; the block dimension x by 2.

And then you are going to take another thread at the same offset from the adjoining block, so the original block dimension by 2 times blockIdx that same thing here, you are applying to the next block, so twice of blockIdx in that dimension plus 1, the adjoining block, in that block you go to the thread ID with the same offset. So, since we are operating at one stride length, so the thread should be separated by one block dimension dot x value.

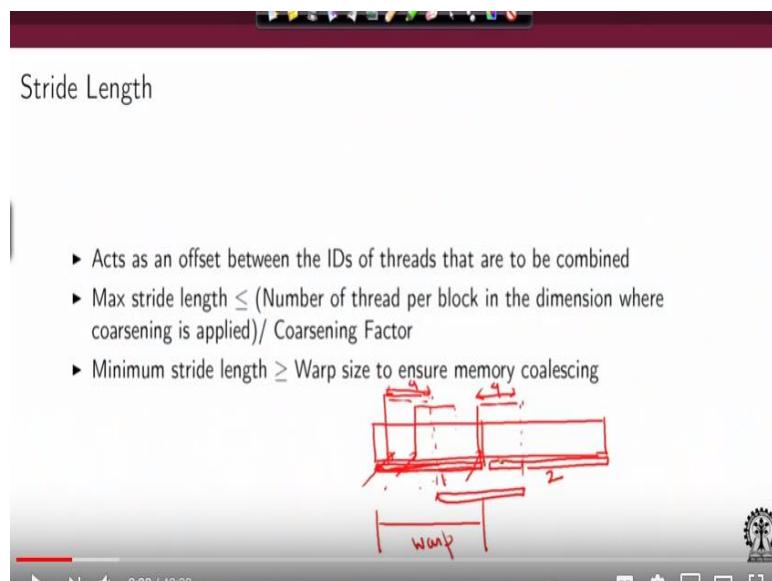
So, maybe with this introduction to the block level coarsening example, we will end this lecture and in the next lecture, we will go into further details, thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture – 42
Kernel Fusion, Thread and Block Coarsening(Contd.)

Hi welcome back to the lecture series on GPU architectures and programming. So if you remember in the last lecture, we explained in detail the idea of thread level coarsening and we are looking into some examples of block level coarsening.

(Refer Slide Time: 00:39)



So just a small point we like to make before getting into further with the block level coarsening example. So if you remember in the thread level coarsening we talked about what is the minimum stride length and we said that it should be greater than the warp size. So just to clarify a bit that I mean, off course, we understand that the basic idea is that it should ensure whatever was the original global memory transactions and coalescing behaviours they should warp fine.

Just to clarify why we will, I mean, clarify further on this consider this example that you have a warp of threads. So this is one warp and it was originally accessing a consecutive data elements now you have coarsened them let us say by some factor of 2 and the stride length is not greater than the warp size. The stride length is, let us say, 4 so if it was greater than the warp size then it the thread should access the original consecutive data elements followed by.

So this is the original consecutive 32 data elements in one global memory transaction and then since if the stride is 32 then it will access the next 32 consecutive data elements in a next transaction that is perfect coalescing activity. Now if the warp size, the stride is still 4 than in the first transaction it is bringing the data like this the original one global transaction. But what more does it want each thread now once the data sitting at a offset of 4.

So in the next transaction is going to bring the same data starting with an offset of 4. So in the next global memory transaction is going to bring, sorry, starting it should the glitch memory transaction. So in the original behaviour this was the first transaction, this was the second transaction.

But now this would be the first transaction and the second transaction; let us say the offset is this, offset is this, the second transaction would be something like this. Now see there is lot of commonality in the data points. So we are actually losing out on the coalescing behaviour in the global scale looking at all the threads in the kernel because there is lot of reuse happening we are not able to cover more number of global memory elements through a perfect coalescing behaviour that was achieved in the original.

Now this can be easily alleviated if we increase the stride from 4 to 32 which is the warp size because then there will be no extra redundant global memory transaction or bringing data and perfect coalescing would be achieved. So this is one point we wanted to clarify about thread level coarsening and then let us move to the example on block level coarsening, we are talking about.

(Refer Slide Time: 03:41)

Block-level Coarsening Example

```

// Shared memory requirements increase for block coarsening. Here since
// coarsening factor is 2, it is doubled

__shared__ float sdata0[BLOCK_SIZE];
__shared__ float sdata1[BLOCK_SIZE];
sdata0[tid] = (i0 < n) ? g_idata[i0] : 0;
sdata1[tid] = (i1 < n) ? g_idata[i1] : 0;
__syncthreads();

// do reduction in shared mem
for (unsigned int s=blockDim.x/2; s>0; s>>=1)
{
    if (tid < s)
    {
        sdata0[tid] += sdata0[tid + s];
        sdata1[tid] += sdata1[tid + s];
    }
    __syncthreads();
}

```

So the first idea about block level coarsening was that since the threads will be picking up activities from blocks sitting at a specific stride length considering coarsening factor of 2, we are going to compute two global threat ids in this case its i0 and i1 and they are both corresponding to thread Ids which are sitting at the same offset value of tidx.

So we compute only one local thread id and we use that single local thread id to get multiple global thread ids from different blocks since we are using a coarsening factor of 2. So you multiply the block idx by 2 wherever it occurs and since we are using a stride of 1 you should have this 2 block idx+1 for taking the thread from adjacent block right threads from adjacent blocks and threads with the exact same local offset.

So now once I have picked up the threads, the rest of the thing is quite same you are going to bring the data from the global memory but then there is something important earlier in the previous example of thread level coarsening, you are coarsening the block, you are coarsening the threads by using activity inside the block.

So effectively the requirement of the each block was remaining same, the compute requirement of each block was remaining same, the threads inside the block we are getting thickened and the number of threads inside the block were decreasing by the coarsening factor. But now you are

creating a block with more amount of activity with respect to the original block and you are not reducing the number of threads.

So when we talk of a reduction kernel or for that matter any kernel, the resource requirements that the block level is increasing. So again I will just like to highlight in the original thread level coarsening, the resource requirement at the block level was kept constant the thread level requirements were increasing. So the SM could accommodate more number of thread blocks in case the par thread level resource requirements were not too high but here what is happening is the thread level activity is increasing but the threads per block is not decreasing its constant.

So effectively the block level activity is increasing for example since I have a coarsening factor of 2 for a reduction kernel. Now each thread is shared inside the block is bringing in effectively double the amount of data and we will need two share memory locations. So we are just allocating to save memories is rate as 0 as data 1 each of size equal to their new block size and the block size, off course, is the same because we do not decrease the number of threads per block.

So now for thread id, the first thread id, I am just using the global threat id i_0 to load data in s data 0. I am using the global threat id i_1 to load data in the other shared memory which is s data 1. Observe one thing when I am using i_0 or i_1 , I am loading data from adjacent blocks; I am loading the data into two shared memories.

Since I am loading the data from adjacent blocks with the same offset, I am loading the data in that different shared memories in the same offset. So just to highlight consider these adjacent blocks so these are adjacent blocks, thread block ,one thread block 2 let us say, I am loading data from here, the offset is tid; I am loading data from here, here also the offset is same.

Now this data will go to shared memory s data 0, this data will go to shared memory s data 1 and the location in which that it, I would go, will also have the identical offset. Because this shared memory sizes are same as the thread block sizes. So with one thread, I am loading data from two

adjacent thread blocks into two shared memories, two different shared memory segments at identical locations.

Now once this is done, you perform the reduction in each of this shared memories in parallel. So all the threads would parallel load data into the shared memories then all the threads would go into doing the reduction. So it will do all the threads will do reduced step here as well as here then again in another reduced step parallelly in both the shared memories and like that. So earlier we were reducing in a power block basis and when we did thread level coarsening each thread was reducing more number of locations.

Now we are reducing across blocks and each thread is doing 2 reduction steps in adjacent blocks by bringing the data in the corresponding shared memory locations and once all the reduced steps, off course, keep on synchronizing for each stride value where the stride I mean the stride of reduction in the original for loop.

(Refer Slide Time: 10:15)

The screenshot shows a presentation slide with a title 'Recap Reduction'. Below the title is a table comparing seven different reduction kernels based on their optimization strategies. The table has two columns: 'Kernel' and 'Optimization'. The kernels listed are Reduce1 through Reduce7. The optimizations described include interleaved addressing, sequential addressing, unrolled loops, template parameters, and multiple elements per thread. A video player interface at the bottom indicates the slide is from a video titled 'Optimization Examples: Thread Coarsening' and is currently at 13:17 / 42:30. A small video thumbnail of the speaker is visible on the right.

Kernel	Optimization
Reduce1	Interleaved addressing (using modulo arithmetic) with divergent branching
Reduce2	Interleaved addressing (using contiguous threads) with bank conflicts
Reduce3	Sequential addressing, no divergence or bank conflicts
Reduce4	Uses $n/2$ threads, performs first level during global load
Reduce5	Unrolled loop for last warp, intra-warp synchronisation barriers removed
Reduce6	Completely unrolled, using template parameter to assert whether the number of threads is a power of two
Reduce7	Multiple elements per thread, small constant number of thread blocks launched. Requires very few synchronisation barriers

So if you just do a small recap of reduction. So these were the different optimizations that we are going one write for reduction 1, your optimization was that there were a lot of problems in that basic reduction kernel. So there was interleaved addressing but you are not really using consecutive threads so that I mean.

So first of all, that was the problem because the warps will contain threads which are not really active. So the warps will also have lot of underutilized occupancy of threads that was first problem and then there was the modulo arithmetic which was also overhead and then there was divergent branches.

Because some of the warps were not that some threads in the warps would actually have no warps and then you in the reduced reduction 2 step, you used interleaved addressing but now you have to actually warp to it continue contiguous threads and that led to balanced warps because the threads will not diverge but still we had the problem of bank conflicts and then in reduction 3.

So this is again I am doing a small recap of the reduction steps because this is relevant here; so in the reduction three step, what you did was, you actually perform sequential addressing and you actually remove the shared memory bank conflicts then in the fourth step of reduction, you half the number of threads and delegated some activity to the threads in terms of global loads. So each thread was practically coarsened to do global loads.

So this is the step from where you start increasing the par thread activity. So this is like some example of coarsening inside the reduction kernels itself. So you use actually $n/2$ number of threads then in reduction 5 instruction 5 you unrolled the last warp and you were able to remove intra warp synchronization barriers for the last warp and all that.

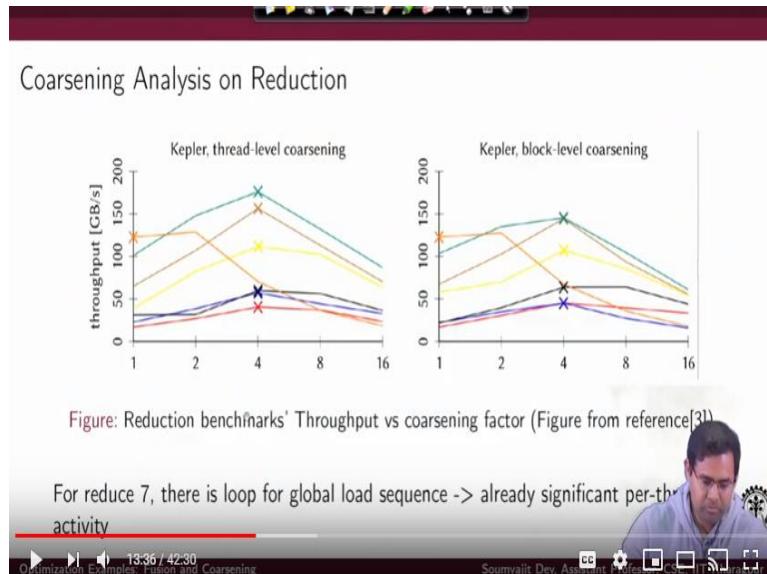
And if you remember in reduction 6 you performed a template parameter based complete unrolling of the kernel based on whatever is the block size you will be generating code where there was effectively no loop and it was a completely unrolled version of the kernel. So all the loop overheads were removed and then in reduction 7 you had multiple elements per thread and you launched very small constant number of thread blocks.

So that was also somewhere you have increased the par shared activity significantly and it was requiring very few synchronization barriers. So the point we are trying to make here first of all these are the reduction steps we are referencing some of these we have discussed up to some

level of detail and you can easily find the further detail of all these reduction steps in the Nvidia example of reduction kernel optimization which is easily available online and which has been referenced in our earlier lectures.

So the point we are trying to make here is with each higher step of production we are increasing the per thread activity significantly and actually removing lot of redundancy and unnecessary barriers and unnecessary loop overheads and things like that.

(Refer Slide Time: 13:36)



So the point here is that if we are actually applying thread coarsening into this different variants of these reductions, we see that how things perform. So here we do an analysis of how coarsening effects the execution time for each of these reduction kernels. So in this plot we provide the examples of this reduction kernels.

So here what we have is this different reduction kernels and their overheads in terms their peak throughputs and how the throughputs keep on changing with the different coarsening factors. So this is the plot for reduction 7 and the other plots are for the reduction kernel this is for the reduction kernel 1 then the 2, 3, 4 ,5 ,6 and this is the 7th kernel so they are in this order sorry the legend in the figure is missing here.

So I hope I mean you can see that with more amount of optimization the initial speed up as you can see in terms of throughput there is it keeps on increasing. So this reduction 1 this reduction 2 3 4 5 6 and 7. Now observe firstly that and so here we are using a Kepler architecture based system tastes like a 40 curve in our system and we are applying a thread level coarse we are actually using 2 different kinds of optimizations.

One optimization is the I mean coarsening optimization we are applying thread level coarsening and we are applying block level coarsening. So the first important characteristics we see here is that the optimal coarsening factor for all of the first 6 reduction kernels appears to be the coarsening factor of 4 both for thread level and block level coarsening.

This is for our system and for this specific example of reduction and for reduction kernel 7 we see that coarsening does not really help it first remains constant or increases a bit in terms of the peak throughput. So that by the way that is the index its telling you that how many how I mean that what is the throughput of the kernel and with coarsening is going to increase.

But however for reduction 7 we see that its gradually decreasing by the way, we like to say that this is a plot where reproduce, I mean, we are actually reproducing from this reference which is actually highlighted here at the end. So this is from this paper and these results actually are given from this paper, we also have this kernel versions but these specific plots that we have given they have been directly taken from this paper I mean you can also try them we have tried these programs and the I mean it is very easy to reproduce this behaviour in your own system.

Off course, the coarsening factors may be it may be different but this is an example and the plots which you have directly getting from this reference here. So what they are trying to show is how far the reduction kernels the activities keep on changing and as I was saying that for reduction 7 you see that there is no real effect of coarsening but for all the others it seems that coarsening by a factor of 4 has 5 is actually providing you the peak throughput.

So this is the observation first brought out by the authors of this paper which appeared recently now if we try to analyse, we thought that, this would be a very nice example for this specific part

course material on thread coarsening because it really brings out the effect that coarsening has in terms of the occupancy of the architecture. Okay first of all, let us understand why for reduction 7, there is no significant speed up.

Now if you remember what was happening in reduction 7 in reduction 6 so first of all we started the coarsening of threads in terms of increasing not really coarsening of threads but actually increasing par thread activity from reduction 4. So I would not say we coarsened threads by we just simply delegated some activities for the threads it was not in the semantics of coarsening and so we will not say that for.

If you see from reduce 4 we delegated some activity to the threads, it was in terms of performing more number of global loads. But then it was only normal par thread activities so we should not say it is not coarsening but just increasing the par thread activity but that also increases the amount of resources the threads are going to consume in terms of memory traffic here in terms of storage in the registers here.

Then it reduce 4 it was simple you just unrolled the last warp for reduction in reduce 6 we completely unrolled so these are more or less 5 and 6 or optimizations where you remove the loop overheads and you remove the synchronization over it specifically introduced 5, reduced experiment the loop overheads but what happens in reduced 7. In reduced 7 the idea of making each thread warps some more is being generalized.

So instead of making each thread bring some only 2 data points we can make each thread bring multiple more number of data points and make them do the local additions on global memory data. So each thread is given more amount of sequential activity in general. So let us say they are adding up they are bringing in data elements from the global memory adding them up and then starts the usual reduction behaviour.

So reduce 7 actually is a good optimization where you can select the amount of sequential addition each of the threads are going to do such that overall we have the perfect occupancy in terms of the reduction kernel. So as you can see with each reduction step we are doing an

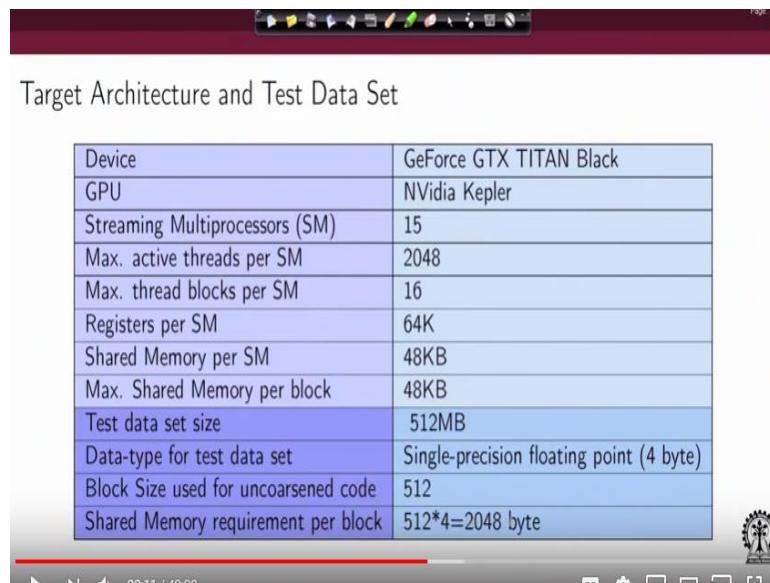
optimization at some point we have reached all the possible optimizations. Now what you are trying to do is you are trying to increase the par thread activity so that you get the perfect amount of occupancy.

So the perfect amount of occupancy can be achieved if whatever is the total number of threads you are able to schedule in your GPUs SM you make all the threads there is this peak number of threads used all the SM resources maximally. So then that is your perfect notion of occupancy and which it seems that with a suitable amount of with suitable choice of par thread activity in terms of number of sequential loads.

I can tune this value and make reducing 7 achieve that perfect occupancy. Now once we do that I hope it is clear to us that once we are able to tune that perfect occupancy just by increasing the sequential activity per thread with respect to reduced 7 further coarsening is not really going to help because whatever is the peak parallelism to be explored is already done with reduced 7.

So that that explains this figure which has been reported by the this publication reference and but let us now try and analyse in our terms that why really these authors were able to get a peak tribute in the Kepler architecture for the other reduction kernels at a coarsening factor of 4.

(Refer Slide Time: 21:26)



The screenshot shows a presentation slide with a title 'Target Architecture and Test Data Set'. Below the title is a table with the following data:

Device	GeForce GTX TITAN Black
GPU	NVidia Kepler
Streaming Multiprocessors (SM)	15
Max. active threads per SM	2048
Max. thread blocks per SM	16
Registers per SM	64K
Shared Memory per SM	48KB
Max. Shared Memory per block	48KB
Test data set size	512MB
Data-type for test data set	Single-precision floating point (4 byte)
Block Size used for uncoarsened code	512
Shared Memory requirement per block	$512 \times 4 = 2048$ byte

So if you look into the target architecture and the test data set for this warp. So this is some analysis we are doing and maybe this is a very good example where we are really talking about optimization in terms of not only algorithmic optimization but also the optimization with respect to the architectural parameters that are there so here the device under test is a Nvidia GeForce GTX TITAN Black GPU for this specific warps by the authors and their GPU is a Kepler architecture number of SMs is 15.

The SMs allow maximum active threads per SM as 2048 maximum thread blocks per SM as 16. So observed these are very important parameters the maximum number of active threads allowed is 2048 maximum thread blocks allowed is 16 the number of registers in the register file inside this SM is 64 K that is the size file size and then shared memory is 48 KB and the maximum shared memory provide allowable per block is also the same.

The test data size, the data for which the reduction has been performed is a 512 megabytes. The data type is single precision floating point in this for the reduction kernel so that is 4 byte. So if we consider the block size for the original uncoarsened code, it has been taken as 512 and so since the block, I have 512 for data points and each of them are floating point type.

So the shared memory requirement for each block is also to 2048 bytes. So this is important in the uncoarsened version, the shared memory requirement power block is 2048 total amount of shared memory available in the SM is 48 kilobytes.

(Refer Slide Time: 23:20)

Coarsening Factor Calculation

Thread-level Coarsening:

Let x be the coarsening factor.

$$\text{Block size} = 512/x$$

$$\text{Active blocks per SM} = 2048/(512/x) = 4*x$$

For thread-level coarsening, shared memory requirement per block is same.

$$\text{Shared memory requirement per block} = 512*4 = 2048 \text{ byte}$$

$$\text{Shared memory requirement per SM} = (4*x) * 2048 \text{ byte}$$

Max allowable shared memory per SM = 48KB.

$$(4*x) * 2048 \text{ byte} = 48 * 1024 \text{ byte}$$

$$\therefore x = 6 \simeq 4 \text{ (since } 4 = 2^2 \leq 6 \leq 2^3\text{)}$$

So let us figure out what is a good coarsening factor here let x be the coarsening factor, so what is the block size we are saying that effectively the block size if I am doing thread level coarsening so I am going to reduce the threads per block so that would be reduced by original block size of 512.

So this was the original block size 512 that would get reduced by x , $512/x$ is the reduced block size with this what would be the number of active blocks per SM. So as you can see that you have maximum active threads per SM is 2048. Now there is a maximum allowed limit on active threads per SM. So now it is going to be active blocks per SM would be original total 2048 threads and threads per block is $512/x$. So that would give me 2048 divided by this.

So that gives me $4x$, off course, the other limiting factor is the maximum thread blocks per SM which is 16 we are just hoping that we will be reaching the threshold by this limiting factor otherwise you have to consider the other limiting factor also. So for the thread level coarsening the shared memory requirement per block remains same as we know because we are coarsening the threads and reducing the threads per block simultaneously.

So per block remain is same so what is the requirement so shared memory requirement per block is 512 times 4 which is 2048 byte as we have required computed earlier already and in that case since we know that active blocks per SM would be $4x$. So what is the shared memory

requirement overall part is same, so that is 4x times 2048 byte and off course, the maximum allowable shared memory in the SM is 48 KB.

So here we have an equation so we have 48 times 1024 and that should be equal to this 4x times 2048 byte. So 48 KB that is multiplied by 1024. So already you get $x=6$ now if we put the nearest value in terms of I mean here if you just get the lower value in terms of that powers of 2 so you get it as 4 since you have it the coarse you can use so since its 6 so you have to go to the next lower power of 2 will be coarsening always in powers of 2 like that and that would give you the 4 here.

So with that, we see that if we choose 4, we get the part of coarsening factor as 4 here out of this calculation that with this value of 4 we are making a use of shared memory such that we have the maximum possible occupancy. If we go beyond this then what would happen is that with further coarsened threads the number of blocks that you can read the number of threads that you can really schedule they will go to be less than the number of active threads per SM.

Because they will keep on requiring more amount of shared memory. So this is a possible calculation that you can do for thread level coarsening but off course, you can here in this calculation we are just using the limiting factor which is the number of active blocks per SM where the limiting factor is taken as 2048 you can just check whether the other limiting factor which is threads per block per SM is also limiting this value of x is just a take home you can just check it and see whether with that if I consider that constant.

I hope you are understanding that here we are considering the constant of the maximum number of active threads and computing using that; okay in that case, what can be the number of active blocks, but we also know that with this the maximum number of thread blocks is 16. So here with $x=4$ I get this as 16. So that anyway would mean that I cannot go beyond this because with a higher factor what would happen is it may happen that you can get you can also hit this limiting factor.

So in general you have to take care of both you can do a calculation and figure out whether if you just start with this as the limiting factor what value do you get and whether it hits the other barrier in terms of the other limiting factor which is this, they are kind of related.

(Refer Slide Time: 27:44)

Coarsening Factor Calculation

Block-level Coarsening:
Let x be the coarsening factor.
Block size = 512
 $\text{Active blocks per SM} = 2048/512=4$
 $\text{Shared memory requirement per block} = 2048*x \text{ byte}$
 $\text{Shared memory requirement per SM} = 4 * (2048*x) \text{ byte}$
 $\text{Max allowable shared memory per SM} = 48\text{KB}$.
 $4 * (2048*x) \text{ byte} = 48 * 1024 \text{ byte}$
 $\therefore x = 6 \simeq 4 (2^2 \leq 6 \leq 2^3)$

27:44 / 42:30

Soumyajit Dey, Assistant Professor, CSE, IIT Madras

So you can just replicate a similar analysis for a block level coarsening. So choose x as a coarsening factor block size is known for the original uncoarsened version of the kernel. Active blocks for SM would be in that case for like us like the original and so shared memory requirement per block you can just.

So here will be what it will be it will increase by the x value, shared memory requirement per block whatever was original it multiplies by the x value and then shared memory requirement per SM would be multiplied by the requirement per block multiplied by the number of blocks. So you multiply by 4 so then you can just have a equation where you have this shared memory required per SM equal to the maximum allowed shared memory which is 48 KB.

So 48 multiplied by 1024 and you get the similar calculations again. So these are the possible ways in which you can also empirically derive limit on the coarsening factor. But off course, there are other issues in the architecture in terms of nonlinear relationships between the different factors like interference and other issues we are not getting to that detail. We are just trying to

provide a handle here like how you can analyse that why possibly you are reaching this kind of a peak performance with this coarsening factor here.

(Refer Slide Time: 29:13)

The screenshot shows a presentation slide with a dark header bar containing various icons. The main title is "Detimental Effects of Thread Coarsening". Below the title is a list of bullet points:

- ▶ Raises a kernel's resource consumption like registers, eventually resulting in reduced occupancy
- ▶ Also increase the pressure on the cache in some kernels

At the bottom of the slide, there is a video player interface. It shows a thumbnail of a person speaking, a progress bar indicating "29:13 / 42:30", and several control buttons for navigating through the video.

So there are several other effects of detrimental effects of thread coarsening which also gets exemplified with this picture. As you can see that the peak throughput we are getting with the coarsening factor 4 for both for thread and block level but then the peak throughput reduces for increased coarsening factors of 8 and 16.

Now we have discussed this that with higher amount of coarsening I can have reduced occupancy and the reason being the kernels resource consumption I mean increases parts read in per thread resource consumption in terms of registers and share memory increases and that is resulting in the reduced occupancy. The other important thing is this also creates an increase in the pressure on the kernel.

(Refer Slide Time: 30:07)

Cache Pressure

- ▶ Overworking memory cache with too many or wasteful memory access
- ▶ Cache line re-use happens when data from same cache line are frequently accessed
- ▶ Cache line re-use (on conventional serial architectures improves performance) is a potential problem for thread coarsening for a GPU.
- ▶ In GPU, cache pressure is, however, not an issue when data is accessed in a streaming manner, where there is no data re-use or cache line re-use.

33:38 / 42:30

So this is an important parameter which is cache pressure, so of a few words on cache pressure. So if you are overworking the memory cache with too many or wasteful memory accesses then it is a bad thing and that is the effect on the cache is used I mean modelled by this term cache pressure.

Now it may happen that you are reusing a cached line time and again by a single thread. So which is which is not a good idea for a GPU so and that can create issues. So when I am doing cache line reuse where for data that means 1 thread is accessing the same cache line multiple times its considered bad for the GPU.

Because in the GPU I would like to have consecutive thread ideas inside a warps accessing sequential data elements and that would make that would mean a single warp makes a single cache line access and brings in the data rather than 1 thread bringing data multiple times in different iterations of a loop or similar kind of activities.

So whenever there is cache line reuse there is a bad thing in a GPU unlike a conventional serial architecture where its considered a good thing. So if you have a GPU kernel where a thread is accessing a cache line multiple times. Let us say you have a for loop using which the thread is accessing consecutive data elements then that is a bad thing ideally you should not be writing the code like that.

But if you have if the programs nature is such that the thread is accessing consecutive locations then in terms of warps is going to access the cache line multiple times by the same thread which is good in the sequential or serial architecture but not in the GPU. Now consider that you are cautioning this kind of a program.

So then this thread will be performing its own execution of a for loop where its making multiple accesses of a cache line and it will also be doing the activity of another originals or another thread in the original kernel and for that also it will be sequentially accessing the cache that the elements inside the same cache line.

So if the original kernel has got lot of instances of par thread cache line reuse in the coarsened kernel there will be further iterations. So just if I am trying to take a picture here that suppose you have a for loop through which 1 thread a for a you have an active a behaviour says that 1 thread id is accessing the same cache line for consecutive memory locations.

So in so there is one load instruction here and it is increasing with the index i and the for loop is iterating over i and 1 thread is looking into the same cache line time and again and it is making a cache line reuse for this kind of a code when I do the coarsening. So then the thread will do this behaviour for multiple cache lines because it will also replicate some other threads activity and again do cache line reuse well it was originally doing cache line reuse only in one place.

So this is bad thing in this for a first of all this is a bad programming behaviour for a GPU ideally this sequential access should have been delegated to sequential threads but if that the nature of the algorithm as I am saying. So then you are increasing cache line reuse by a coarsening factor which is significantly increasing the cache pressure and due to this the overall kernels throughput will decrease.

So there is the impact of cache pressure on a program and the coarsening will hamper it further.

(Refer Slide Time: 34:10)

Cache Pressure Due To Coarsening

- ▶ With coarsening, in the worst case, the same number of active threads per SM will be executing as in the original kernel, but with each thread doing more work.
- ▶ In this worst case, the number of cache lines being accessed at any time will thus scale with the coarsening factor increasing the pressure on the cache.
- ▶ Higher coarsening factors usually yield lower occupancy and result in fewer active threads, the effect is not always linear.
- ▶ However, in the presence of cache line re-use, the cache pressure frequently increases with the coarsening factor. This typically outweighs the benefit that coarsening might otherwise bring.

So in general I can say that if I do a coarsening in the worst case the same number of active threads per SM will be executing as in the original kernel. But now each thread will, off course, be doing more warps and due to its each thread doing more work. If the work is to load the cache line multiple times then the number of cache lines accessed will also scale with the coarsening factor just as we are saying and with higher amount of coarsening factors will have more higher cache line reuse.

And that will effectively mean walk down the SM and decrease the occupancy in that way and this will significantly outweigh the benefit of performing any kind of coarsening whatsoever.

(Refer Slide Time: 34:57)

Simple Matrix Transpose

- ▶ Each thread reads a single element from source array and writes to target array at its transposed index
- ▶ Cache lines are read in coalesced fashion by threads of single warp
- ▶ But written to different cache lines in an uncoalesced way from multiple warps
- ▶ Data cached for read will not be accessed again
- ▶ Cache line for write ideally should be present till written by each warp (cache line re-use)

not evicted quickly

Figure: Coalesced Read

Figure: Uncoalesced Write

(Figures from reference[3])

Now let us take another example of the matrix transpose kernel which we know as a its a quite difficult kernel and there are lot of optimizations which we have already discussed. Now let us have a relook into this from the point of view of coarsening. So in the single matrix transpose kernel each thread will read a single element from the source array and write to the target array let us consider the situation where we are doing coalesced reads whereas we follow un coalesced write.

SO the cache lines are read in coalesced fashion by threads in the single warp because we are doing coalesced read. So there is no much of cache line reuse in terms of read but when the writes are going to happen they are uncoalesced. So writing to the different cache lines is happening in an uncoalesced way in multiple different warps.

So data cached for a read first of all is not going to be accessed again. So there is the thing that when you are reading from here you are reading in one shot, so you are reading in one shot after the data has been read this cache line is not going to be accessed. But that does not mean immediately after the read has happened the cache line will be evicted. Why? because the write is the read is performed quickly.

But then the write is going to happen column wise in a uncoalesced way from multiple warps. So that the write is going to happen in different in multiple cycles. So for the same warp so let us say this is the warps which has performed the read this warp is going to load the data here in multiple cache transactions, so there will be multiple cache transactions happening.

So at any point of time we should be saying that well there are lot of write instructions in flight since there are a lot of write instructions in flight and the average latency of the cache flight is very high and that effectively increases the amount of time for which I am unable to evict the data from a cache line.

I hope this is clear as long as I am not able to read as well as write well the data column wise I cannot evict data from the cache. Now since the writes are getting serialized in this case I am

unable to perform you know I am unable to actually evict the data. So that creates significant delay and decreases at throughput.

(Refer Slide Time: 37:52)

The slide title is "Simple Matrix Transpose". On the left, there is a bulleted list of three points:

- ▶ As writes come from different warps, cache lines will not be evicted until the data is written by all the warps responsible for writing.
- ▶ For a coarsening factor of 2, twice as many instructions (in worst case) may be in flight at any time,
- ▶ Average time for completing writes to cache lines will increase.

On the right, there are two diagrams labeled "Figure: Coalesced Read" and "Figure: Uncoalesced Write".

Figure: Coalesced Read: This diagram shows a vertical stack of four horizontal rows of colored squares. Each row contains four squares: the first two are green, the third is yellow, and the fourth is red. Ellipses indicate that this pattern repeats. The label "Figure: Coalesced Read" is positioned to the left of the diagram.

Figure: Uncoalesced Write: This diagram shows a vertical stack of four horizontal rows of colored squares. Each row contains four squares: the first is green, the second is yellow, the third is red, and the fourth is green. Ellipses indicate that this pattern repeats. The label "Figure: Uncoalesced Write" is positioned to the left of the diagram.

(Figures from reference[2])

So as there again as the writes are coming from the different warps the cache line will not be evicted until the data is written by all the warps responsible for the writing. So the read is coalesced write is un coalesced write takes multiple cache cycles. So I cannot evict the data from the cache even immediately after the read happens. So for a coarsening factor of 2 I will have twice as many instructions which may be in flight at any time.

So I hope you are understanding that if we consider this transpose operation already there was this difficulty with the write and there was this issue with the cache, now if I increase the coarsening factor earlier whatever was the number of instructions which were in flight at any time that would increase by the factor itself.

So the average time for completing the writes to the cache are going to increase it was already high the average time was already high the average time for the cache line eviction was high and since the write time will increase further due to the coarsening. The coarsening increases the parallel thread activity the coarsening will increase the amount of data to be written to the column.

So we will just go back again if we coarsen what happens now we will be reading more amount of data in the coalesced read. So again we will be writing more amount of data in the coalesced write by inside the single thread I am not having the advantage of having multiple threads to do the writes. So the amount of sequential column wise access here for the data will increase.

So that would further increase the cache line reuse here I mean that would further increase the amount of time for which I cannot evict the data from the cache.

(Refer Slide Time: 39:31)

Effects of Coarsening on Matrix Transpose Kernel

- ▶ Primarily depends on how threads are scheduled on SM and memory system
- ▶ If we coarsen by a factor 2, there will be two read-write statements, so twice as many read and write instructions are required
- ▶ So as we increase the coarsening factor, the number of cache line access will also increase resulting in cache pressure
- ▶ So, the average time taken to complete write to a given cache line increases leading to decreased performance

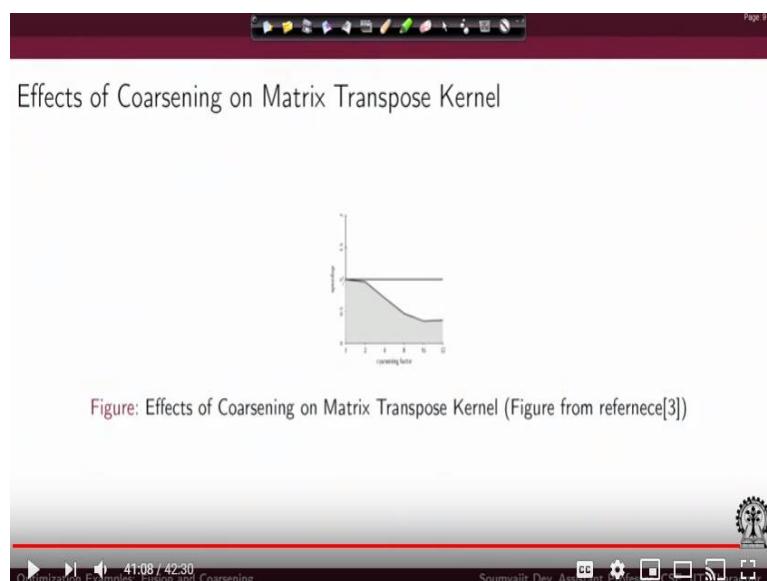
So this also will create a problem here and so just as an example this was our uncoarsened version of the transpose kernel we have seen earlier. So you just read you compute the index in row wise input and you write column wise write and for this kind of a non-optimized matrix transpose kernel the effect of coarsening is severely bad.

So it primarily depends on how threads are scheduled on the SM and memory system. Therefore there is a dependency and if we coarsen by a factor of 2 as we are saying there will be 2 read write statements so twice as many read and write instructions are required here twice as many of course. Because I am increasing the per thread activity by 2 and as we increasing the coarsening factor the number of cache line access will also increase.

Because now each thread will access more and now that would effectively increase the cache pressure and therefore as time taken to complete the write to the cache line increases as we have discussed since that time taken to complete the write increases it also increases the overall latency and it decreases the performance in terms of as we have seen from the picture that after I mean for the reduction kernel there was a decrease.

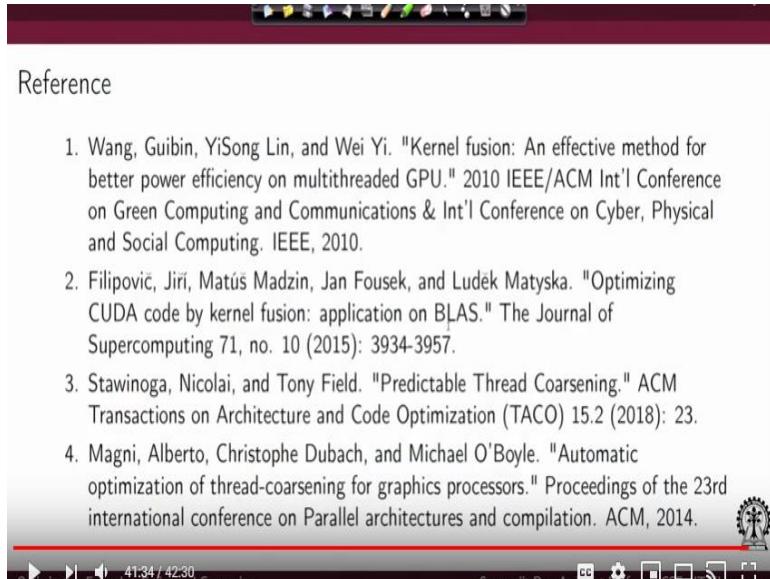
But for transpose doing any kind of coarsening is going to be difficult specially for this transpose kernel where we have this kind of a column wise write and row wise reads.

(Refer Slide Time: 41:01)



So this is the simple example here this is a picture again taken from the reference paper we are talking about that for this kind of a matrix transpose by the way this is the optimized matrix transpose that we discussed earlier in our lectures on matrix transpose as you can see any kind of coarsening is actually creating the cache coarsening and cache reissue and create increasing the latency of the writes and due to that any kind of coarsening is linked to a hamper in the leading to complete hampering of the speed-up.

(Refer Slide Time: 41:34)



So with this we like to end our discussions on the different on thread coarsening is good and bad ways I hope we have performed a good architectural level analysis specifically for these examples. So you can really think more in these terms for other examples and the other example algorithms and their course learnings and here are some of the references important references which we used for this lecture material.

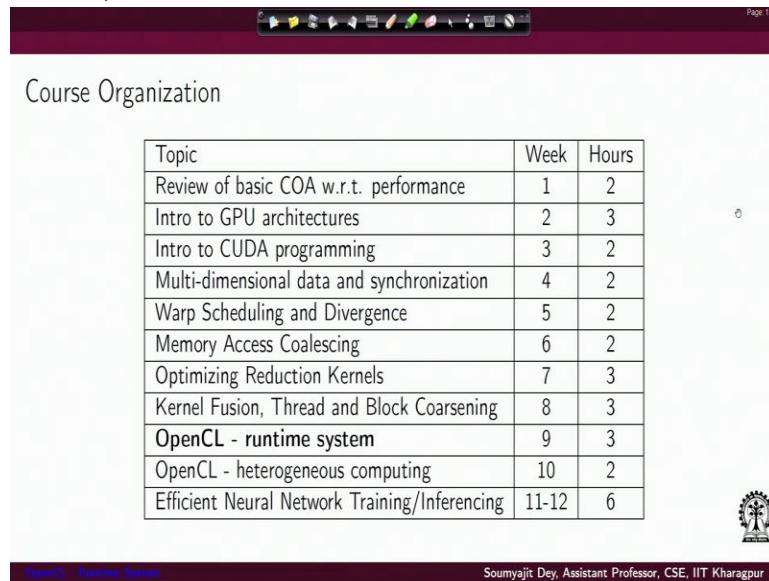
These are reference on kernel fusion which we extensively used then the second reference on kernel fusion. These are all some of them are quite recent papers actually and the thread coarsening part was almost entirely taken from this predictable thread coarsening paper which appeared very recently and off course, this is another paper on optimizations with respect to thread coarsening. So with this we will be ending this long lecture and thank you for your attention.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 41
OpenCL-Runtime System

Hi, welcome back to the lecture series on GPU architectures and programming. So from this lecture onwards.

(Refer Slide Time: 00:31)



The screenshot shows a presentation slide titled "Course Organization". At the top, there is a toolbar with various icons. Below the title, the slide content starts with a table:

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

In week 9, we will be moving out of CUDA programming of GPUs. And we will be introducing an alternate programming paradigm which is also fairly useful in the context of GPU programming and this is what we call as OpenCL programming language. So, the way we will be making the coverage is that we will be introducing OpenCL as a programming language. But also, we need to understand that it has an underlying runtime.

So we will look into how the runtime system operates. And the next important thing would be looking into OpenCL. Unlike CUDA, which is useful only in the context of Nvidia GPUs, we will like to see that how OpenCL became a more kind of open, and vendor independent offering. How it is applicable for a wide variety of GPU or GPU like programming approach and computer architectures.

And that is why we will see that how OpenCL is specifically suitable for what we know call as heterogeneous computing. That means one can write an OpenCL program, which can be partitioned and scheduled on a complex system comprising different computing blocks, starting from CPUs, GPUs, as well as FPGAs.

(Refer Slide Time: 01:50)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Recap'. Below it is a bulleted list:

- ▶ GPU architecture
- ▶ GPU programming using CUDA
- ▶ GPU optimization techniques

At the bottom, there is a footer bar with the text 'OpenCL - Runtime System' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' along with the IIT Kharagpur logo.

(Refer Slide Time: 01:50)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'GPGPU Timeline'. Below it is a bulleted list:

- ▶ 1999-2000 computer scientists from various fields started using GPUs to accelerate a range of scientific applications. GPU programming required the use of graphics APIs such as OpenGL and Cg (C for Graphics).
- ▶ 2002 James Fung (University of Toronto) developed OpenVIDIA.
- ▶ In November 2006 Nvidia launched CUDA, an API that allows to code algorithms for execution on GeForce GPUs using C programming language.
- ▶ Khronos Group defined OpenCL in 2008 supported on AMD, Nvidia and ARM platforms.

At the bottom, there is a footer bar with the text 'OpenCL - Runtime System' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' along with the IIT Kharagpur logo.

So with this background if we just look back into how the idea of OpenCL came in. So in the time around 1999 to 2000, computer scientists from various fields have already started using GPUs to accelerate a range of scientific applications. And, also, we know that if I am looking specifically

into the case of graphics workloads, then also GPUs were particularly developed targeting those kinds of workloads.

And if you look into this idea of who was the primary vendor for GPUs that would be Nvidia around that time. And then people also start to think of alternate programming languages. Around 2002, James Fung from University of Toronto developed OpenVIDIA. And in the way CUDA came into the picture was in 2006, November. Nvidia launched the CUDA API, which actually allowed programmers to write programs that would execute on Geforce GPU.

And finally around 2008 this group, essentially, a conglomeration of people started coming from different companies, the Khronos group, they came up with the specification of OpenCL. And it was supported by some of the leading semiconductor vendors like AMD, Nvidia, and ARM.

(Refer Slide Time: 03:28)

The slide has a dark header bar with various icons. The main title 'OpenCL - Open Computing Language' is centered above a list of bullet points. The list describes OpenCL as an open, royalty-free standard for portable, parallel programming across CPUs, GPUs, and other processors like DSP. It is a framework for parallel programming including a language, API, libraries, and a runtime system, supporting C, C++, Python, and Java. At the bottom right is a video player showing a person from the chest up, wearing a dark jacket, against a blue background. The video player has a progress bar at the bottom labeled 'OpenCL Runtime System' and a name 'Soumyajit Dey, Assistant Professor, CSE, IIT Roorkee'.

- ▶ Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing across CPUs, GPUs, and other processors like DSP
- ▶ A framework for parallel programming includes a language, API, libraries and a runtime system.
- ▶ Language support for C, C++, Python, Java

So what opens here, first of all, the full form would be Open Compute Language or Open Computing Language. So essentially, it is an open royalty free standard for portable parallel programming or heterogeneous parallel computing or I mean portable parallel programming for a wide range of systems like CPUs, GPUs, and also other kinds of processors are, for example, DSP processors. Now the idea is that we have seen how CUDA works.

So essentially you write data parallel kernels and you execute them in GPUs, but just like GPUs have got a way to execute assembly code. And we know that CPUs also have that facility, although

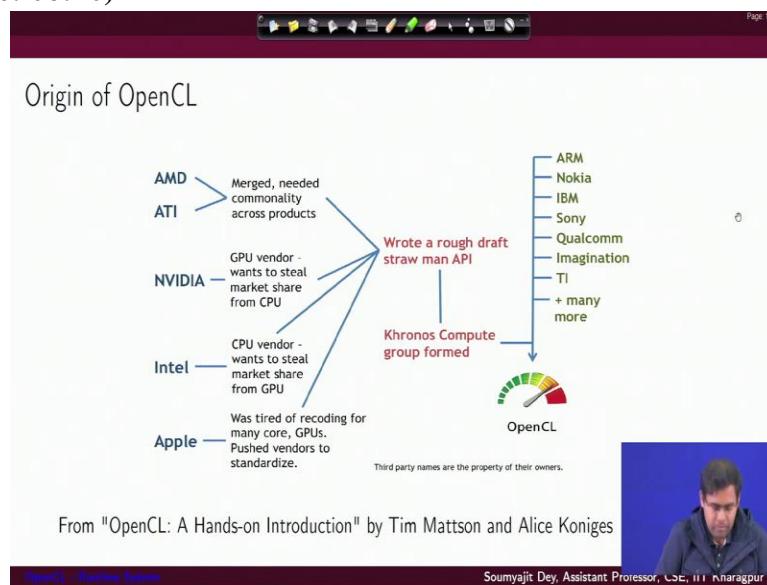
at a limited level of parallelism. Similarly, there are also other processors like DSPs, who also provide facilities like parallel data parallel program execution, of course at a level that is not much as parallelized as GPUs.

But the idea was there, since this kind of data parallel computation is supported by other architectures also, why not we come up with a specification such that any semiconductor vendor if they develop drivers, which satisfy their specification, then the program should run on those platforms. So, OpenCL in totality, comprises the definition of OpenCL Language, its application, programming interface, the API, the associated libraries, and a runtime system.

The runtime system will manage the dispatching of the code and interfacing with the lower level drivers and all that. Also, OpenCL provides language support for C, C++, Python and Java. That means that the underlying thing is essentially all defined in C. But for providing somebody it will be able to give some high level specification like from a language, which is much easier to understand or easier to in terms of specification.

And there are OpenCL C++ as well as Python binding like PyOpenCL, which are also useful for writing OpenCL code. So that one need not write the code and add the much at the level of detail that is required if you are writing an OpenCL program in the original semantics.

(Refer Slide Time: 06:10)



So, how did really OpenCL come up with first of all these comments and things that are written here and not my personal opinions, but they are more like so, I put in the references from where I have taken up this idea that how the evolution of OpenCL happened. So, for example, there were a series of events which took place for example, the company that is AMD got merged with ATI. Now, ATI some famous for creating low cost graphics card and AMD brought ATI and created his own line of GPU architectures on the in parallel.

Nvidia was also working and it has come up with Nvidia GPUs which are already in the market and have drawn a significant amount of market share from CPUs because people soon found that these GPUs were useful for running some parallel compute, like they were giving much higher throughput with respect to a comparable CPU. But that would also mean that Intel CPU vendor like Intel will like to get people back into their fold by providing or by supporting a language.

So that if you write GPU oriented kernel in that language, it will also run seamlessly in the CPU maybe with a different performance factor. And some other company like apple at that point of time was also advocating usage of OpenCL. So all Apple products came with OpenCL library already built in so that if you are writing an OpenCL code that would be that can just run like off the shelf on any Apple system.

So with support from all these meters may some vendors the khronos group was formed who kind of develops the specification of the OpenCL language and they formulated the basic draft of OpenCL. And also it received support from a significant number of other companies like embedded suppliers like ARM, Nokia, IBM, Sony. Qualcomm and TI were also among the supporters. There are many others who support OpenCL.

So, when I say there are companies who support OpenCL that would mean that whenever they come up with a silicon, their own architecture process the architecture associated with that they will also provide a library, a low level library and the required drivers called the ICD files using which, if somebody is writing an OpenCL program, satisfying the original khronos specification of OpenCL programs, then that library would support the execution of such a program on the respective platforms.

(Refer Slide Time: 09:09)

The slide has a dark header bar with various icons. The main title is 'OpenCL Working Group'. Below it is a bulleted list:

- ▶ Diverse industry
 - ▶ Processor vendors, system OEM, middleware vendors, application developers
- ▶ OpenCL became an important standard upon release by virtue of the market coverage of the companies behind it.

Below the list is a grid of company logos under the heading 'PROMOTER MEMBERS'. The grid includes logos for AMD, Apple, ARM, Google, Intel, NVIDIA, Qualcomm, Samsung, Sony, Valve, and many others. At the bottom of the slide is a video feed of a man speaking, identified as 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

So, this is also a figure which has been borrowed from the existing literature, which kind of shows that OpenCL is supported by a diverse set of silicon vendors system OEMs, middleware vendors, application developers, and it is and due to this huge popularity, because of the set of companies that are actually in its support. Now, of course, one natural question would be that, I believe there will be 2 important questions like, we always hear about Nvidia GPUs and CUDA.

But we are not that much familiar with OpenCL as a programming language. First of all, the reason is, I would say, sensitive stuff fundamentally C base programming language and also the specification is a bit more complex, in my personal opinion writing an OpenCL program using the lowest level CPI would require a bit more effort than writing a CUDA program.

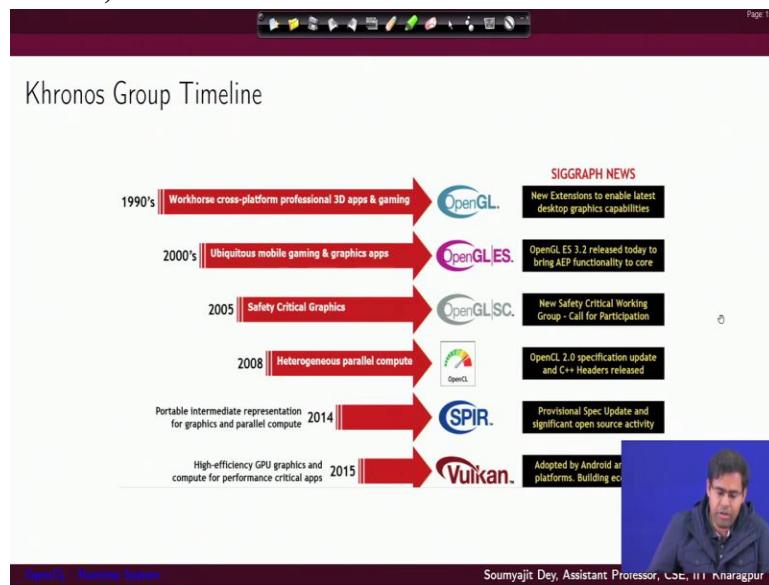
But then also there are benefits of that and we will soon see that why the apparent difficulty is not significant, because in most cases, it is a one time job. So, somehow, it is not significantly popular to knife programmers. But if you discuss this with people, let us say working on embedded processing, and there is significant penetration from OpenCL programs. And also there are other vendors who are slowly realizing why OpenCL is important from the perspective of computing with heterogeneous systems.

Second important thing I would say if you notice that there are also FPGA based companies like Xilinx, who provide a lot of support for OpenCL. The idea is that is I mean, the way FPGA

development works is, you will write a program in a hardware description language like Verilog or VHDL. And there would be a synthesis tool, maybe a high level synthesis tool or a normal standard synthesis tool which will actually compile your structural or behavioral specification written in that high harder description language and create the corresponding LMT map for the FPGA.

But the thing is, when they say that, they are also supporting an OpenCL that would mean they provide alternate high levels in those tools through which if you are writing an OpenCL kernel, it is also compilable and runnable on FPGAs coming from Xilinx, so there is also an internet workload so as we can see that we have one common language through which if I write an assembly kernel, it can execute on any vendor supported CPU on any vendor supported GPU as well as a FPGA like device.

(Refer Slide Time: 12:13)



So this would be an alternative way to look at it. So in the 1990s, the workers cross platform professional 3d apps and gaming. That was OpenGL. And it came up with new extensions to enable desktop graphics capabilities. Coming to 2000s. We had ubiquitous mobile gaming and graphics apps being supported. So that actually meant OpenGL had an extension. OpenGL ES 3.2. It was released to bring this AP functionality to the court.

So they supported all these embedded domains like mobile gaming and graphics apps. And then we in around 2005, there was another extension of OpenGL into safety critical graphics, so that

was the extension OpenGL SC. So it was a separate working group which is a safety critical working group. And around 2008, there was this advent of OpenCL as a language for heterogeneous parallel compute.

And pretty soon there was this OpenCL 2.0 specification, and associated pro header files and libraries were released. I mean, the headers were decided, and the libraries, of course, had to be released with vendor backing. And around 2014, we have this portable intermediate representation from for graphics and parallel compute. And finally, something important happened around 2015, which was the creation of the Vulkan API.

So people found that, in the mobile world, for writing a program or writing a data parallel kernel is possible to write an OpenCL based heterogeneous kernel and run it. But maybe it is still not that popular from the mobile developer's point of view, and alternative was the Vulkan API, which is a very much OpenCL like language. And it is supported directly in the Android platform. So this would not mean you can have OpenCL code executing inside a mobile system.

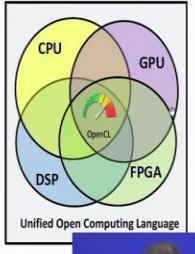
For which you have to have OpenCL support on the mobile system, which is fairly common if you have an ARM Mali GPU or similarly other vendor supported GPUs who also provide the corresponding OpenCL ICD files. However, an alternative way, which is a bit easier would be to write the code using the Vulkan API, which is also a very much like OpenCL, or the original OpenCL like specification for writing such data parallel kernels, and they get direct support from the Android ecosystem. And that is also heavily used nowadays for writing performance critical graphical apps.

(Refer Slide Time: 15:08)

Page 1 / 1

OpenCL: Portable Heterogeneous Computing

- ▶ Open-source, cross-platform, cross-vendor standard that target a wide range of systems like supercomputers, embedded systems, mobile devices
- ▶ Enables programming of diverse compute devices like CPU, GPU, DSP, FPGA
- ▶ Same code can be executed on all these devices
- ▶ Can dynamically interrogate system load and balance across available heterogeneous processors



Unified Open Computing Language



OpenCL: Heterogeneous Systems

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, this is what we have been discussing, in a nutshell, it is a portable, heterogeneous computing language. It is open source, it has cross platform, cross vendor standard. It can target a wide range of systems starting from supercomputers to embedded platforms, mobile devices, etc . Using it, you can program diverse compute devices like CPUs, GPUs, DSPs, and FPGAs. And the same code can be executed on all these devices.

That is another very important point as we are trying to make that you write an OpenCL kernel. You can execute it in any of those devices provided the vendor of the device and gives you the support libraries. And it can also dynamically interrogate the system load and balance its execution across heterogeneous processors. Now this is something important we need to understand that if we are considering a heterogeneous system at any point of time there may be each of the constituent processing elements may be partially loaded.

So, from an idea load balancing point of view, when I submit an OpenCL kernel, it should be able to get an idea what is the amount of load in each of the compute elements and accordingly balances execution across the different platforms. And that is also a facility which is provided inside the OpenCL runtime system.

(Refer Slide Time: 16:27)

The slide has a dark header bar with various icons. The main title 'OpenCL Language Specification' is centered at the top. Below the title is a bulleted list of features:

- ▶ Subset of ISO C99 with language extensions
- ▶ Well defined numerical accuracy (IEEE 754 rounding with specified max error)
- ▶ Rich set of built-in functions: cross, dot, sin, cos, pow, log ...
- ▶ Can be compiled JIT/online or offline
- ▶ Execute compute kernels across multiple devices

At the bottom left is a small video thumbnail of a speaker, and at the bottom right is the name 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

So, what about the language essentially, it is a subset of C99 with certain extensions, it has well defined numerical accuracy and of course, rich set of special transfer special math functions. And the usual ways that you can either decompile it or you can regenerate it have generate a binary in offline mode. So, ideally, I can have a runtime compilation and execution or I can generate offline binary code. Now, the last point which we were discussing earlier also that I can create a compute kernel and I can distribute it's threads across different devices.

(Refer Slide Time: 17:12)

The slide has a dark header bar with various icons. The main title 'CUDA and OpenCL Correspondence' is centered at the top. Below the title is a table comparing CUDA terms to OpenCL terms:

CUDA	OpenCL
GPU	device
multiprocessor	compute unit
scalar core	processing element
thread	work-item
thread-block	work-group
grid	NDRange
global memory	global memory
shared memory	local memory
local memory	private memory

At the bottom left is a small video thumbnail of a speaker, and at the bottom right is the name 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

Now, here is something important since we are coming from a CUDA background. So, we will first need to understand how all the concepts of OpenCL are very much linked up with all the concepts of CUDA maybe some of the names get a bit changed. So, the moment, we are able to

identify what is the correspondence, we will soon understand why they are so similar. The fundamental reason is both OpenCL and CUDA are built with the assumption of execution model as well as a memory model or I would say a platform model.

So if you remember how a GPU architecture was genetically define, we said that, it comprises a set of streaming multi processors or SMs. SM was containing shared memory L1 cache, and there were a set of special function units and ofcourse the Scalar Processor or SPs. And these SPs were actually used to execute the CUDA threads in with a basic scheduling unit of warps.

So we had a hierarchical arrangement of processors, the basic processors who are grouped together to create SMs and all the SMs together created the entire GPU, that is about the processing part. Regarding the memory organization part, we thought that, inside the GSM, we would have shared memory and L1 cache. And of course, we also discussed that in modern GPUs, we also have the shared memory and L1 separate units.

But the other important thing was there are a collection of such SM together with unified cache and then there was the global memory. And of course, for each thread we can have its own definition of local memory. Now, when we say local memory there would 2 options, finally, you have you will have local variables for thread which are private to each of the threads. They will be making use of the subset of register file which is allocated to each thread and our identity.

So, essentially from each threads point of view, it will get binding with an SP and a set of registers. SP means scalar processor that this thread executes on this scalar process, it has access to this subset of registers from the entire register file for storing its local variables. And if these histories are not enough, then it can further make use of local memory, which is essentially a small part of the original global memory.

So that was a memory model and execution model, OpenCL also considers a similar memory model and execution model. So, again the point would be that whenever any vendor is trying to create some silicon, which will be OpenCL compliant, here is to ensure that whatever process

architecture it defines it should have a correspondence with this memory model and architectural execution model otherwise, it will be difficult to implement of corresponding libraries.

Now, coming back to this name correspondence, whatever in CUDA, we only had GPUs from Nvidia, whereas in OpenCL, it is going to be a heterogeneous compute language supporting a set of devices we will call everything a device, every OpenCL device, a set of OpenCL devices together create a platform for computation. Inside the GPU, we had SM (streaming multi processors) in OpenCL will call them as compute units.

So we have OpenCL device and OpenCL device will contain compute units. Inside in each of the streaming multi processors of GPU, we had this collection of scalar core or scalar process or SPs. And here we will just call them SPs or processing elements. So just to recollect, a GPU is replaced by the term device because everything is an OpenCL device inside OpenCL device like GPUs, where we had streaming multiprocessors, we have compute units in OpenCL semantics.

Inside streaming multi processors, we have scalar processors, that will now have processing elements. Inside the GPU, we had threads executing. A collection of threads will be called work items. A bunch of threads together was defining a thread block. A set of work items or set of thread-block work items will be defining what we call as a work-group. So the block becomes work-group and the threads become work-items.

And if you remember, we had a set of blocks, and set of thread blocks, which together defined the entire grid of CUDA threads, the grid is known as ND range here. So grid-block-thread that was the hierarchy in CUDA. Here it is ND range, work group and work item in OpenCL. The name global memory remains same, what was called shared memory in CUDA becomes local memory. That means, inside that each compute unit, we have a shared memory, which in OpenCL terminology would be called local memory.

So, it is local to that compute unit, but it is not local to the processing elements. So, it is local to the compute unit. And the original definition of local memory would be called as private memory in OpenCL.

(Refer Slide Time: 22:45)

The screenshot shows a presentation slide with the title "CUDA and OpenCL Correspondence". Below the title is a table comparing CUDA and OpenCL keywords:

CUDA	OpenCL
<code>__global__ function</code>	<code>__kernel function</code>
<code>__device__ function</code>	no qualification needed
<code>__constant__ variable</code>	<code>__constant variable</code>
<code>__device__ variable</code>	<code>__global variable</code>
<code>__shared__ variable</code>	<code>__local variable</code>

At the bottom of the slide, there is a small video player window showing a person speaking, and the text "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

Now, in terms of functions, if you remember for functions, we have the following keywords in CUDA. So we had `global` device. So whatever was a `global` function becomes the name of kernel function. So a `global` was a function which was scalable from the host but executable on the device in CUDA, here we are calling it a kernel. So it is OpenCL kernel we execute on a device and a `device` function means it is scalable as well as executable only in the device. In case of OpenCL such a kernel does not require any qualification.

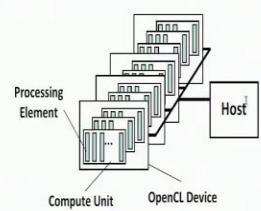
Now, coming to variable types, we had `constant` variable types that name remains same in OpenCL. So that would mean a variable which would be only defined once it would encourage exclusive storage space will be the `constant` memory segment in the GPU. Now that terminology and idea would remain same. Then in CUDA, we had this keyword called `device` for variables, which would be defined in the GPU DRAM.

In case of OpenCL will have the `device` on memory where it will be defined as a `device` on `global` memory where this will be defined with that keyword `global`? Again since the `shared` memory, the name `shared` memory is being used with as a `local` memory. So, the `shared` keyword for this CUDA variable gets replaced with the `local` keyword. So, `shared` is `local` and `device` becomes `global` in terms of variable names.

(Refer Slide Time: 24:17)

OpenCL Platform Model

- ▶ A host is connected to one or more OpenCL devices
- ▶ OpenCL device is collection of one or more compute units
- ▶ A compute unit is composed of one or more processing elements
- ▶ Processing elements execute code as SIMD or SPMD



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, here we have a pictorial view of the OpenCL platform model. So, this is how the OpenCL platform model is conceived. And as I was saying, any vendor who is trying to create an OpenCL compliant processor data parallel processor has to conform to this theoretical model. So we identify that there will be a host device, which will connect to 1 or more OpenCL devices and as an OpenCL devices as we have discussed earlier, it is a collection of 1 or more compute units.

Each compute unit will have one or more processing elements. And these processing elements will execute code in a data parallel way. So, or SIMD or SPMD, i.e. single program multiple data.

(Refer Slide Time: 25:09)

OpenCL Execution Model

- ▶ The two main execution units in OpenCL are the kernels and the host program.
- ▶ When a kernel is submitted for execution by the host, an index space is defined
- ▶ An instance of the kernel called a work-item executes for each point in this index space.
- ▶ Work-items are organized into work-groups
- ▶ An NDRange is an N-dimensional index space, where N is one, two or three

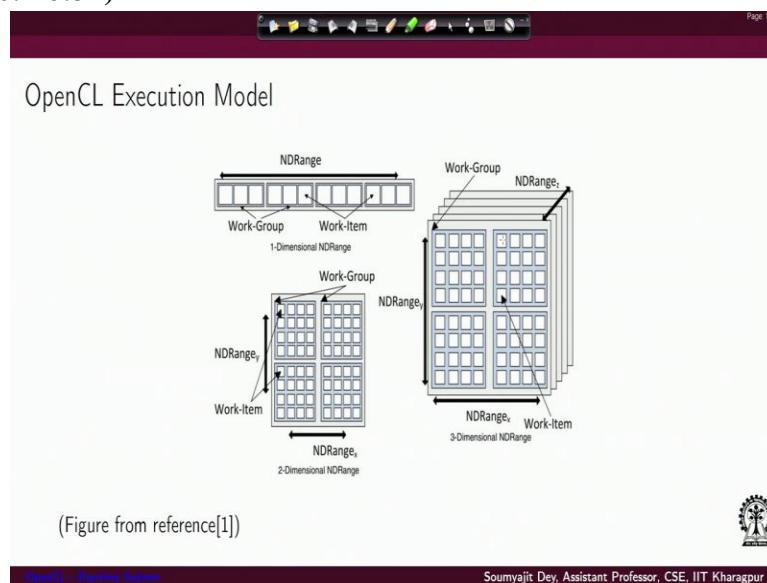
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, similar to CUDA programming, OpenCL also have 2 main execution units, they are the kernels and the host programs. Essentially the host program will be occupying this host device and it will be orchestrating the execution of the OpenCL kernels in the different other OpenCL devices. So, when you submit OpenCL kernel using a host program to one of the OpenCL devices, it defines an index space. An instance of the kernel is a work item and it executes on each point in this index space.

So, if you remember in CUDA, we had launched set of thread blocks. So, if you see here that is now the index range. And inside this we are instead of launching threads, we are launching this work items, each work item will execute and try to compute some value for one of the points in this index space. Now, the work items are now organized into work groups as we are discussing earlier.

And by ND range what we mean is that in the N-dimensional index space, where the value of N can be 1, 2 or 3, so, in the highest possible dimension, we can have a 3 dimensional arrangement of this OpenCL work items.

(Refer Slide Time: 26:37)



So, this picture is kind of reproducing us the idea of parallel execution of threads and the different OpenCL keyword concepts. If you remember our initial introduction to the CUDA programming language, we showed the arrangement of CUDA threads. Where the threads are bunched into thread blocks, the blocks are bunched together into a grid. Here we have a similar thing, we are

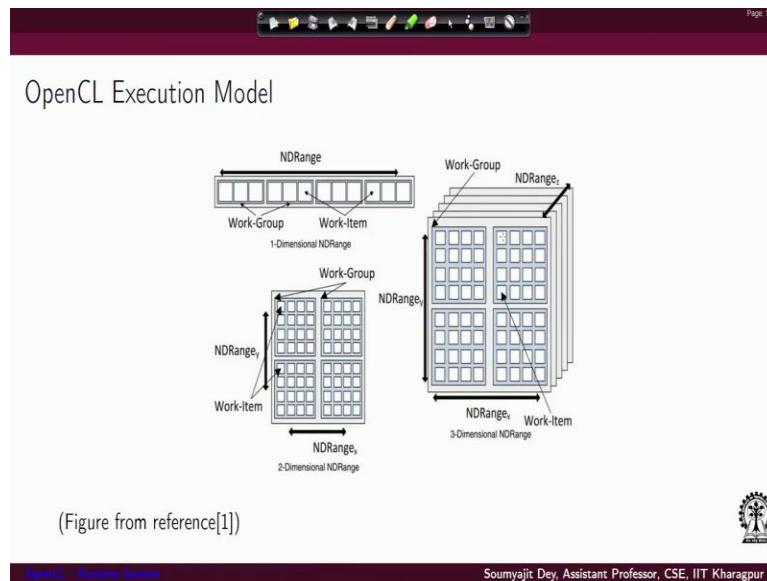
trying to show different possible cases of one dimensional 2 dimensional as well as 3 dimensional element of the index space in the ND inch.

So, we have a 1 dimensional ND range where we have multiple work groups, each of the work groups are containing work items. Similarly, you can have can have a 2 dimensional ND range, which will contain 2 dimensional work groups and the work groups again will contain the work items arranged in 2 dimensions. And similarly, I can have also a 2 dimensional ND range where the work groups are also packed into dimensions. So with this basic introduction to the OpenCL execution model, will end this lecture. And in the next lecture, we will be continuing the discussions further. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

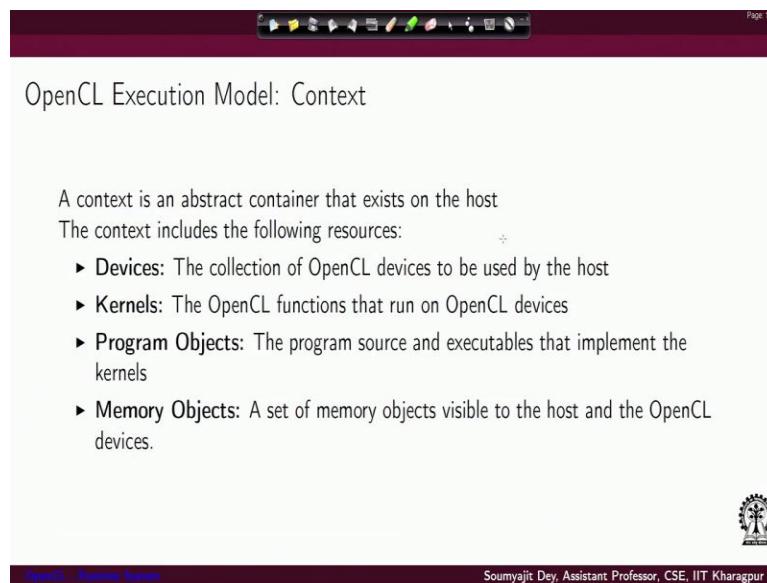
Lecture # 42
OpenCL - Runtime System (Contd.)

(Refer Slide Time: 00:33)



Hi, welcome back to the lectures on GPU architectures and programming. So, in the last lecture, we have briefly introduced these concepts of OpenCL programs and we were discussing the OpenCL execution model.

(Refer Slide Time: 00:40)



So, continuing that, we will slowly introduce the different OpenCL concepts one by one. So, what will come first is the OpenCL context. So, we are starting with the execution model of programs. So, what is the context? So, essentially our context is nothing but an abstract container that exists in the host side and it includes the following resource definitions. So it is like an abstract object and its context containing information of the following resources.

That is first of all the devices that it will contain the information about what are the OpenCL devices which had to be used by the host and how to access them, it will contain the different OpenCL functions that are to be run on the respective OpenCL devices and it will contain this program objects. So essentially what is the program source and executables that implement the kernel, so you write a kernel.

Now, from that kernel, the OpenCL runtime system will build a program object and then it will also contain the memory objects. So the set of memory objects which are visible to the host and OpenCL devices. So in a nutshell, so we are trying to define how OpenCL programs execution is set up. It is all based on this definition of a context and the context captures what am I going to execute, which are the kernels, which are translated into a program objects.

And they are going to use data from again the memory model, which is known to the host in the form of memory objects and the devices like which are the actual assembly architectures where these kernels will be executing.

(Refer Slide Time: 02:34)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "OpenCL Execution Model: Context". Below the title is a bulleted list of points about the context. At the bottom of the slide, there is a footer bar with the text "OpenCL - Runtime System" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur". To the right of the footer, there is a small portrait of a man.

- ▶ The context is created and manipulated by the host using functions from the OpenCL API
- ▶ It coordinates the mechanisms for host-device interaction
- ▶ It manages the memory objects that are available to the devices
- ▶ It keeps track of the programs and kernels that are created for each device
- ▶ Limiting the context to a given platform helps to fully utilize a system comprising resources from a mixture of vendors (ICD: "Installable Client Driver")

So, a context is something that gets created by the OpenCL API , as essentially, you have to write a suitable host program, and you have to call suitable functions from the open underlying OpenCL API through which have to come we have to actually create the context. Now, it is useful primarily for executing the kernels getting their results back from the devices, using those devices for executing more kernels in other devices, and all that.

So, essentially the context inside , whatever coordination one will do, , we understand from our basic knowledge and GPU programming now, that the host device is actually responsible for orchestrating the execution of kernels in different GPU devices. So, coming to OpenCL, the similar concepts are actually going to hold. So, the host problem is the host site code is actually orchestrating the execution of this kernel objects on different devices.

But all these things are happening inside the definition of this context is also managing the memory objects it is deciding which memory object is accessible to which device and all that so, essentially the context is an abstraction which binds devices with programs and memory objects. Now, limiting the context to a given platform helps to fully utilize a system comprising resources from a mixture of vendors this is a useful thing.

Like why do we need the definition of a context, the primary reason is because of the OpenCL platform model. So, it may always have been that you have built a large compute system for

multiple different kinds of devices coming from different vendors, let us say AMD, or say somebody else, they are all connected to the PCI Express Bus. Now let us understand that each of them may be OpenCL vendors.

So what they will do is each of them will provide the underlying ICD Installable Client Driver conforming to the OpenCL specification, but they are all specific for each vendor. So a collection of devices from a specific vendor for them. If I define a context, then that context will use that vendor applied ICD for performing your execution. For some other context I will define for some other diverse set of devices from some other vendor and I can have the host program, coordinate executions in different devices, where each of the devices are mapped to different context.

So, inside the context, I have the memory definition that device definition pro kernel definition and program execution going on I can have another context where another set of kernels are executing and top level program you can actually manage multiple contexts executing on different vendor server supplied platforms. And it can help these different contexts communicate data among each other for performance benefits and getting more optimization.

So the basic idea would be that you want to have an abstraction through which you can actually link up things from different vendors in a coherent way, and still proceed with building something big.

(Refer Slide Time: 06:06)

OpenCL Execution Model: Command Queue

- ▶ Host creates a data structure called command-queue to co-ordinate execution of the kernels on the devices.
- ▶ Host places commands into the command-queue which are then scheduled onto the devices within the context.
 - ▶ Kernel execution commands
 - ▶ Memory commands
 - ▶ Synchronization commands
- ▶ These execute asynchronously between the host and the device
 - ▶ In-order Execution (default)
 - ▶ Out-of-order Execution

OpenCL - Parallel Systems

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

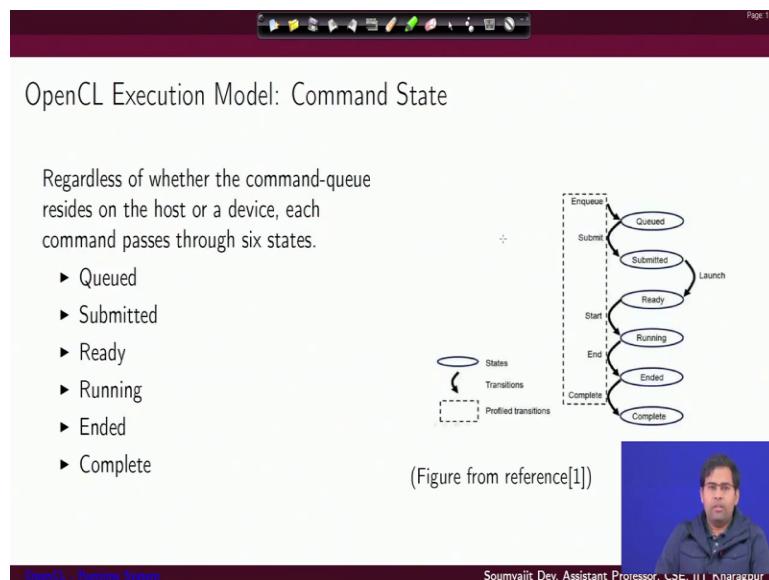
The next important concept here will be, what we call as the command queue. So when we are talking about context, as I said, it is basically an abstraction through which I am trying to define how an OpenCL program, we look at the memory, how opens CL program, you look at a device and all that but there has to be a specific formalization that the devices are going to execute this kind of commands in this specific order.

And how will that sequence of commands coordinate among each other? So for that, the top level host let us understand that difficulty. With the host trying to manage a set of devices, as I said in the worst case that devices can be of different vendors. That problem is solved by bunching out devices together into a set of context. So the host has to manage a set of devices sitting in different contexts. Well, that is fine. But when the host is trying to dispatch some action for each of the devices, the host needs some mechanism for doing that.

So what the host does is it creates our data and other data structure called command queue through which it can submit instructions that to our device that you execute a kernel, you get this data back, you use this data as an input. So these are all elements that decide the actions like which kernel will perform, on which device will it perform, and in what order it will perform. These are orchestrated by the host through a command queue. So all that the hosts does is that, it place commands to the command queue, which are then scheduled into the devices within the context.

A device will have its own command queue and the kinds of commands can be different. They can be kernel execution commands, they can be memory transfer related commands for the device and they can also be synchronization commands. Now, these queues can execute both in order as well as out of order. The host can also specify that for some device, whether the commands that are queued up in the command queue of the device whether they will execute in the order in which they are queued up or they can execute out of order.

(Refer Slide Time: 08:22)



Now, we have understood what is the command so, essentially it is one of these following things like kernel execution and memory transfer instruction or a synchronization command. So, these are the different possible commands that our hosts can issue to a specific kind of device. Now, once this command is issued, it can go through the following well defined states that the OpenCL runtime system understands.

So, a command will be queued. The command will be submitted from the device, it will be dequeued and submitted to the device. It is ready to execute. It is actually running the command and the command gets completed. So these are all the different possible status messages for a command under execute. The lifetime varies between when it starts sitting in a queue up to the point when it gets completed in its execution.

(Refer Slide Time: 09:18)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "OpenCL Execution Model: Synchronization". Below the title, there is a text block and a bulleted list. At the bottom right, there is a small video player showing a person speaking. The footer contains the text "OpenCL: Parallel Systems" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

Synchronization refers to mechanisms that constrain the order of execution between two or more units of execution.

- ▶ **Work-group synchronization:** Constraints on the order of execution for work-items in a single work-group
- ▶ **Command synchronization:** Constraints on the order of commands launched for execution

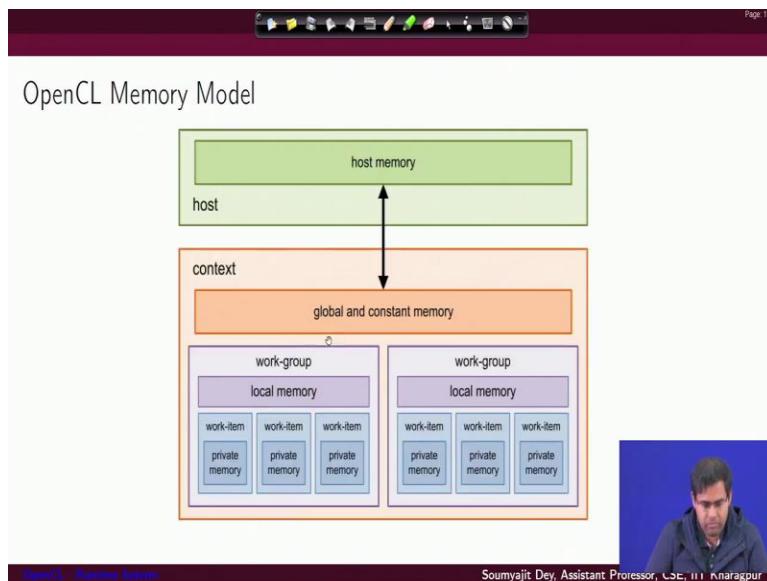
Now in OpenCL synchronization execution model, the other important thing from our perspective would be synchronization. So, what is synchronization? So, we understand that in our data program, we often will require threads to synchronize at certain specific points, which are the barriers. For example, in CUDA we had the synchronized constraints, similarly, in OpenCL, we have work groups containing set of work items and work items will often like to synchronize among each other.

So, the synchronization would refer in this case to mechanisms that constraint the order of execution between two or more units of execution. So it is possible to do while group synchronization. So this essentially means constraints on the order of execution for work items in a single work. So this is kind of similar to idea of sync threads in CUDA programs. So a sync thread would synchronize all the threads inside a thread block.

But not across thread blocks. Here in this case, I can perform in similar way I can synchronize work items sitting inside single work group. The other notion of synchronization is command synchronization. So I can put sensors synchronization primitives inside the command queue. As you can see that this is one of the possible commands that can be submitted by a host program in a command queue. So I can give a kernel execution command, or memory transport command and I can also give a synchronization command.

So when we talk about a synchronization command that would mean a constraints on the order of commands launched first in execution. So, that would mean a suitable synchronization command will be enqueue by the host on the command queue. And it will restrict the order in which commands are going to be launched for execution.

(Refer Slide Time: 11:15)



Now, as we have been discussing that in OpenCL, we define that execution model and also a memory model. And so, this is a picture of the memory model. So, you will have host memory and so, that is a host side memory and host side compute environment, which is we have here from the host, you can copy data to the global memory of a specific context. Now, this is where things become a bit more complex and different with respect to CUDA.

We need not have a single device we need not have a single GPU, I can have a rather small collection of devices. Among the collection of devices I can have devices coming from different vendors accordingly. What I will do as a memory object, I will define a context as an abstraction inside this current context, I will have a collection of devices, there are global and constant memory. And I will have this notion of work groups executing on the processing elements.

So, work group will be constituting work items, and this work items will be have access to their private memory, the shared local memory and in the worst case, they will all have access to the global memory in the context. So, I can have multiple devices in the context. So I can have a set

of four groups executing together in one device, another set of four groups in other device like that. So, to be more specific, what we have more here is the host can orchestrate execution across multiple devices and they will set up multiple devices that can be grouped into this abstraction of contexts.

(Refer Slide Time: 13:04)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Memory Objects'. Below the title, there is a text block and a bulleted list. At the bottom right, there is a small video frame showing a person speaking, and at the bottom left, there is some navigation text.

The contents of global memory are memory objects. Use the OpenCL type `cl_mem`.

- ▶ **Buffer:** Stored as a block of contiguous memory and used as a general purpose object to hold built in types (such as `int`, `float`), vector types, or user-defined data used in an OpenCL program. Can be manipulated through pointers much as one would with any block of memory in C.
- ▶ **Image:** Holds one, two or three dimensional images. Formats are based on the standard image formats used in graphics applications. Must be managed by functions defined in the OpenCL API.

OpenCL - Memory System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, coming to the memory object definitions, so, the contents of global memory are essentially open memory objects which used OpenCL type `CL_` name. Now, memory objects can be of the type buffered, which are stored at block of contiguous memory and used as general purpose object to hold the built in types like integers or floats, other vector types or any user defined data type in an OpenCL program. And they can be manipulated through pointers similar to any kind of memory block access that can you can do in normal C program.

The other thing is image so in OpenCL, you have a separate memory type defined for images, which is useful for holding multi-dimensional images up to 3 dimension. And the formats are based on standard image formats. So like you have support for different standard image format. So that is specifically provided targeting graphics applications, which was the original workload domain that was targeted. And they need to be managed by functions which are defined in the OpenCL API.

Which would mean whatever vendor provided ICD file you get as part of the OpenCL distribution on your browser. It should contain implementations of the required functions for reading and storing such image objects.

(Refer Slide Time: 14:28)

The screenshot shows a presentation slide with the title "OpenCL Memory Model". Below the title is a table comparing memory types across Host and Kernel perspectives. The table has five columns: Global, Constant, Local, and Private. The rows are categorized by perspective: Host and Kernel. The table details the allocation and access rights for each memory type.

	Global	Constant	Local	Private
Host	Dynamic Allocation	Dynamic Allocation	Dynamic Allocation	No Allocation
Kernel	Read/Write access to buffers and images	Read/Write access	No access	No access
	Static Allocation for program scope variables	Static Allocation	Static Allocation	Static Allocation
	Read/Write access	Read-only access	Read/Write access	Read/Write access

Source: OpenCL Working Session
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So if we do a summary here of the OpenCL memory model, so from the global perspective, in the host, you have support for dynamic allocation. You have read/write access to buffers and images. From the kernels perspective, you have static allocation for programs called variables and read/write access. If you look at the constant memory segment, again, from the host side, you can do a dynamic allocation. You have read/write access from the kernel side, and you would have static allocation and only read only access from the constant.

But, so whatever has to be defined from the host side as constant. Now, from the local memory part again for host you have a dynamic allocation. But from the kernel side you again have static allocation, but every thread can of course, read as well as write in its local memory of each of the compute units and have similar behavior is also expected from the private memory for each thread.

So, that is what you also have read/write access in the private memories, of course, for the local and the private memory which are in the device, because this is in the inside each of the compute units and this is inside each of the processing elements that are sitting inside the compute units. So

they are all in that device. So, the host really does not have any access to this memory segments. This is what is supported by the OpenCL kernels that are submitted to the respective devices.

(Refer Slide Time: 15:57)

OpenCL Programming Model

- ▶ Data Parallel Model: Defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object
- ▶ Task Parallel Model: Defines a model in which a single instance of a kernel is executed independent of any index space.
- ▶ Hybrids of these two models

OpenCL - Runtime System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if we come to the programming model, you have a data parallel model, of kernel execution where you define the computation in terms of a sequence of instructions that are, of course applied to multiple elements of a memory object. So that is how we have been writing usual C code and using usual CUDA code. And the alternative is task parallel model. Where do you define a model in which single instance of a kernel is executed independent of any index space. So this is also something we will look into and of course, you have support for a hybrid combination of both these models.

(Refer Slide Time: 16:36)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "OpenCL Data Parallel Programming Model". Below the title is a bulleted list of points about OpenCL's hierarchical data parallel programming model. At the bottom right is a small video frame showing a man speaking, identified as Soumyajit Dey. The footer includes the text "OpenCL Runtime System" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

- ▶ OpenCL provides a hierarchical data parallel programming model
- ▶ In explicit model programmer defines the total number of work-items to execute in parallel and how the work-items are divided among work-groups.
- ▶ In implicit model programmer specifies only the total number of work-items to execute in parallel and the division into work-groups is managed by the OpenCL implementation.



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, in OpenCL, you have support for a hierarchical data parallel programming model, which would mean in the explicit model, the programmer defines a total number of work items to execute in parallel, and how the work items are divided among the work groups. And you can also have an implicit model where the program will only specify the total number of work items to be executed in parallel.

And the actual way in which the work items would be divided into the work groups will be managed by the OpenCL implementation. Now, this is important, this essentially means that as a programmer you can choose to explicitly specify the hierarchy of work items and work groups or you can actually let the runtime system take care of that, so, that would be the implicit model.

(Refer Slide Time: 17:29)

OpenCL Task Parallel Programming Model

- ▶ Logically equivalent to executing a kernel on a compute unit with a work-group containing a single work-item
- ▶ Users express parallelism by -
 - ▶ Using vector data types implemented by the device
 - ▶ Enqueuing multiple tasks

OpenCL - Function System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And this is logically equivalent to executing a kernel on a compute unit with a work group containing a single work item. So, this is more about the task parallel programming model. So, logically it seems as like executing a sequential task and the parallelism is expressed by vector data type **implemented by the device** and you can just also use multiple tasks.

(Refer Slide Time: 17:59)

OpenCL Program Structure

- ▶ Platform Layer API
 - ▶ Abstraction layer for diverse computational resources
 - ▶ Query, select and initialize compute devices
 - ▶ Create compute contexts and work-queues
- ▶ Runtime API
 - ▶ Launch compute kernel
 - ▶ Set kernel execution configuration
 - ▶ Manage scheduling, compute, and memory resources
 - ▶ Synchronization

```

graph TD
    PL[Platform Layer API] --> HP[Host program]
    PL --> R[Runtime API]
    HP --> H[Host program]
    H --> P[Platform Layer]
    H --> R[Runtime]
    K[Kernels] --> L[Language]
  
```

OpenCL - Function System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, the introducing the name only and will of course focus on this later on. Now, coming to this OpenCL program structures, like how really OpenCL program is ordered into different parts. But for that, we will first see that there are different supports in a typical OpenCL program. The first part will define what we call as the platform API. And the second part will deal with what we

call the runtime API. So first of all, let us understand what is the platform API? Because then we will see that how that is required to query the platform in the first day of the OpenCL program.

So in OpenCL runtime system, you have a platform layer API, which is an abstraction layer for diverse computational resources and as a programmer you have to use the query functions defined in this platform lead API for querying selecting and initializing different compute devices. And using this API, you can also create contexts. So that is the high level API through which you have to actually create the different objects and abstractions like context, the memory objects, the program objects, and all that.

And also, once all these data structures are up and running, then you will be using functions from the runtime API to launch the kernels and manage the execution of the kernels, scheduled executions throughout the threads across the different devices. And utilize the memory resources and achieve synchronization as and when required across the threads and all that. If we just try and understand at a broad level, inside OpenCL program, the 1st priority would be to set up the data structures.

That are there, which will be managed by the platform API and then to do the actual workflow scheduling across the different devices which is managed by runtime API now, as we have been discussing in a OpenCL program, you will have initial part which is the host program part and also the kernel part. And the host program part we have a set up different steps to execute. And this is kind of a much more I would say, a bit more complex.

Then, normal CUDA was program where all you need to do is just define the basic variables and then define the setup the memory space for the GPUs and launch the kernels. The reasons are also something that we will discuss, but first let us understand what are the different steps that OpenCL host program has to perform. The first thing it will do is, it will query the different compute devices through the platform API and try to figure out the way the devices are arranged they are from which vendors.

And the query phase of course if you know your executives and all that, then you can write an OpenCL code, which will not require to do this query and all that. But let us understand that we are trying to line we are trying to understand what is the generic way to write OpenCL programs, which can execute on any open source supported device. So for that, initially, the program that you write should be able to query the platform.

And get suitable values about its architecture information, which it will use for setting up the different memory objects and which it will also use to schedule the work items and all that. So using this query phase, you are first learning about the parameters of the underlying platform. And whatever you learn, let us say the number of devices from different vendors and all that that are useful for creating the context of execution as we have discussed earlier.

Inside the context, you will next be creating the memory objects in the context and then the next step will be to compile and create the kernel program objects. Now as you can see, this is also part of the runtime system. This is also supported by the runtime API. But these are all activities that are to be written in the host program itself. So now this seems a bit weird that we are saying, we have a program and we have to first set up the context.

We have created the devices and set up the context and now we are seeing that, we will have further commands in the host program to create memory objects associated with context that is also fine. But the next thing is we will have compilation commands in the host program for combining the kernel code that is also going to come after those programs. Now, that is a weird thing because till now, we have been looking at things like we will write the code and then we will use a compiler to compile it but let us understand here.

The compilation process has to be managed by the host program and the way the program will be executed with will be managed by the runtime system. So the entire thing starting from , starting from basic compilation to setting up the memory objects and using these compiled kernels to issue commands to the command queues, everything is performed by the host program. So this makes an OpenCL a bit more complex.

But at the same time, the structure of most of the OpenCL host programs are very similar. So once we are acquainted with one, writing another OpenCL host program is quite easy, because most of it is going to be a copy paste job. But, we need to understand what are the different steps so the takeaway at this point would be that the first program is not only responsible for declaring memory objects declaring inside the content, binding them to context but also for doing a runtime compilation of the different kernel program objects, and then, including the kernel launch commands in a specific order into the command queues. So, that also means the host program is also responsible for defining the command queues in a per device manner. And in giving the different kinds of commands like data transfer commands, can launch commands synchronization commands all of them in the command queue.

And at the end of after that the next step will be launching the kernels through the runtime system. And once the kernels are finished their execution and the results have been properly computed to clean up the underlying resources, which were occupied. , these are the underlying resources that the open source objects were occupying. So this entire orchestration of computation activity, memory object definition, activity command, including activity command execution, activity, synchronization activities, and the final cleanup.

This entire thing is managed by the OpenCL runtime system based on the query of the platform and the creation of context which is supported by the platform layer. So these are the different phases through which a typical host program or an open source program would go through. And so then the question is what is happening with the kernel.

The kernel is a simple CL program with certain restrictions and extensions. And so, that would also be there about the host program, , the point I was also trying to make multiple times. I would say that the host program will be also responsible for runtime compilation of this kernel problems.

(Refer Slide Time: 25:31)

```

OpenCL

__kernel void vadd(__global float* a,
                   __global float* b, __global float*
                   c, const unsigned int n)
{
    int i = get_global_id(0);
    if(i < n)
        c[i] = a[i] + b[i];
}


```

```

CUDA

__global__ void vectorAdd(float* a,
                           float* b, float* c, unsigned int n)
{
    int i= threadIdx.x + blockDim.x *
           blockIdx.x;
    if(i < n)
        c[i] = a[i] + b[i];
}


```

OpenCL Working System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, today will end with just a loop on a loop, and a simple OpenCL kernel, which would be our most popular vector at kernel. So on the right hand side, I have vector kernel in CUDA, on the left hand side I have a vector kernel in OpenCL. So as you can see, like we discussed earlier, the global keyword in CUDA gets replaced by underscore whereas in 1 side only in the kernel keyword in OpenCL and here you had the input parameters and output buffer parameter.

And the size similar things, so, you have A, B and C, but they also come with this global keyword in case of the OpenCL, the OpenCL variables, if you go back to our earlier discussion.

(Refer Slide Time: 26:32)

CUDA	OpenCL
<code>__global__ function</code>	<code>__kernel function</code>
<code>__device__ function</code>	no qualification needed
<code>__constant__ variable</code>	<code>__constant variable</code>
<code>__device__ variable</code>	<code>__global variable</code>
<code>__shared__ variable</code>	<code>__local variable</code>

OpenCL Working System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, when we are talking about this, we had this global variable declaration here and that would mean that though they will be defined in the device global memory, and the output will be computed again in the device for each specific device's global memory. Now, coming back to the CUDA code you see that the first item was to compute a global thread id so this was all usual computation equal to `threadIdx.x + blockDim.x * blockIdx.x`.

And then based on the value of id being said the range, you are computing a normal C statement which is `c[i] = a[i] + b[i]` here in OpenCL code, the last 2 lines are exactly the same, because fundamentally it is a C program with its own definitions of OpenCL types, data structures and runtime primitives. But look at the way `i` is computed, you simply have a call to something called a `get_global_id` function providing you the global id global id fellow for this thread.

The corresponding OpenCL work item and the work item will execute this edition for this specific value of `i`. So that is one good aspect here, you need not write this line. And again, rather you just figure out what is the global id using this kind of function. So you have similar functions like `get_local_id` to get the id inside the thread inside the work group. And so the `global_id` will give you the actual position inside the global arrangement of threads that we have defined.

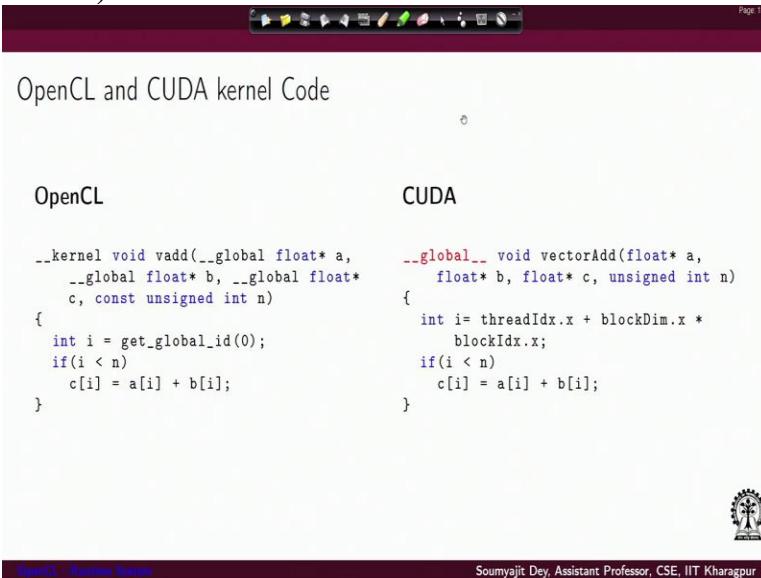
And the 0 will signify that this is just in the first dimension that we are talking about and this is not a multi-dimensional kernel. Now, if we write 1 or 2 things like that we will be actually talking about finding of the global id specifically in that dimension. So in case you are trying to linearize a multidimensional kernel, there are additional OpenCL global kind of commands, which we will be discussing later on. So with this minor introduction to writing OpenCL kernels will end this lecture. And then next lecture we will look at the skeletal structure of OpenCL host program. Thank you for your attention. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 43
OpenCL - Runtime System (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming. And so, in the last lecture we have just introduced the example on OpenCL and CUDA kernel if you remember.

(Refer Slide Time: 00:36)



The screenshot shows a presentation slide with a dark header bar containing icons for navigation and search. The main content area has a title "OpenCL and CUDA kernel Code" and two side-by-side code snippets. The left snippet is labeled "OpenCL" and the right is labeled "CUDA". Both snippets are C-like code for vector addition. The OpenCL code uses `__kernel void vadd(...)` and the CUDA code uses `__global__ void vectorAdd(...)`. The code is identical, showing a loop from 0 to `n` where each element `c[i]` is set to the sum of `a[i]` and `b[i]`. At the bottom of the slide, there is a footer bar with the text "OpenCL - Runtime System" on the left and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur" on the right, along with the IIT Kharagpur logo.

```
OpenCL
__kernel void vadd(__global float* a,
                   __global float* b, __global float*
                   c, const unsigned int n)
{
    int i = get_global_id(0);
    if(i < n)
        c[i] = a[i] + b[i];
}

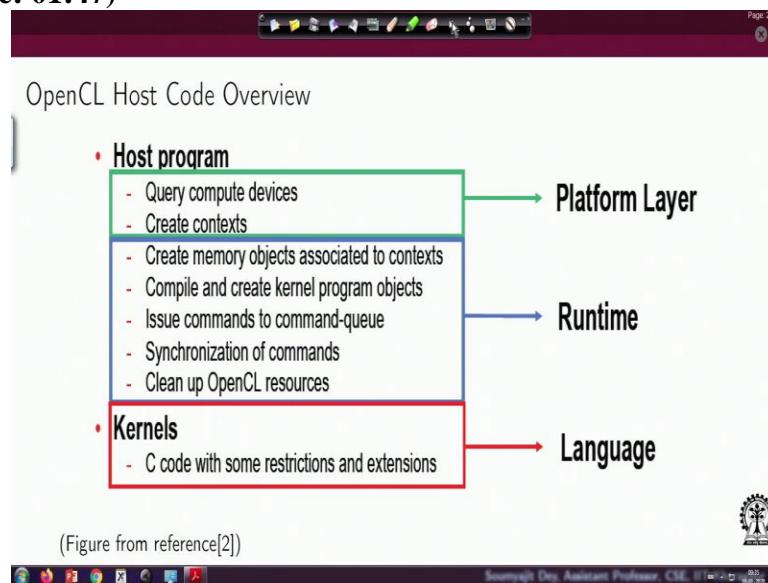
CUDA
__global__ void vectorAdd(float* a,
                         float* b, float* c, unsigned int n)
{
    int i= threadIdx.x + blockDim.x *
          blockIdx.x;
    if(i < n)
        c[i] = a[i] + b[i];
}
```

And we have just shown a simple Vector addition example, where we are side by side showing examples of OpenCL and CUDA kernel just to identify what would be the differences in the way things are to be written when you are creating an OpenCL kernel. So, from this 1 thing is for sure we understand that both they are kind of extensions of C and when we come to OpenCL, there are several variants at that at the lowest level you have the OpenCL since specification which is being implemented in the form of a C library that can execute in a parallel manner in the suitable devices.

And at the high level, you also have C++ based wrappers and PyOpenCL, which is a higher level wrapper, which makes the job of writing OpenCL kernel is far easier. At least writing the host programming specifications becomes easy for our purpose since we are living with a C background in this course. We are assuming on with that so, we are looking into everything at the lowest level

that how OpenCL C library and a specified what are the function calls which are used for different functionalities.

(Refer Slide Time: 01:47)



So coming back, if you just recall the way we discussed the different layers through which OpenCL code executes. So every OpenCL program comprises this host program that will start with some platform layer level queries. Because in the runtime system, you have the platform layer followed by the runtime. So, these are essentially what is going to the platform layer and to let know, what are the compute devices available in a system.

And again, I will say that this is the way we are going to write a genetic code. Of course, if you are going to write a simple, OpenCL code, which is specific to some device, your code can be much simpler. So initially, we will be querying the compute devices and based on this query will get the result how many compute devices are really there and what are the different vendors, what is organization of the compute device? which of them are sitting on the same die and all that.

And based on this information of computer devices, you can decide that you will be kind of accumulating a set of devices together to define what we call as a context essentially, that a context is an abstraction through which multiple devices can share their data. If you have segregated a set of 2 sets of devices into 2 different context, they would need to communicate among each other through the host program without having any direct scope of communication or synchronization among them.

Now, once you have queried the compute devices, understood their organization and created the context in the platform layer, you move on to the runtime where they are actually happening in the program. You have to do all these things followed by execution of the kernel. So we are just trying to specify the structure of the program here. So these are the initial structures and now they will be followed by the way different memory objects are to be created and they are to be associated to the context.

So, this is the part of code which should come immediately after you create the context. Once this is done, we also assume that you have the kernel to be used is written, and the typical way in which the host program will be managing the kernel, is where something really is a bit different. So, essentially, we will be capturing the kernel like a string, let us say in a character edit. So, once that is done, there will be OpenCL kernels, which will be used to create object.

A program object from that character string, which is containing the kernel and with this, you have the entire setup ready. So just to summarize, again, you have query devices, you have segregated devices belonging to different vendors in architecture specific way, so that you can create platforms in the platform layer.

After you query the devices, you create context. Of course, one context again, this is also important that in 1 context, typically, it would be using that vendor's OpenCL implementation. You may have 2 different contexts sitting in 2 different vendor provided devices at 2 different vendor. Different vendors provided platforms containing multiple devices. So this is very important, you have to understand, you can have multiple platforms coming from different vendors.

And each platform can contain multiple compute devices of CPU, GPU, maybe some other compute devices, like dsp and all that. So, that is why all this complexity at the lower level program can come in. But you have to also understand that this gives you a lot of power with respect to deciding how to really program such a heterogeneous device. And as we move on the importance of heterogeneous devices, will be clear, very soon.

So you query the devices, you understand the entire setup of the system that is you understand, which are the different vendor supplied devices, or platforms inside the platforms? How are the devices organized? Based on that you can have, let us say you have 4 devices in a platform you can create, I am just giving an example. Subsets of 2 devices to create 2 different context. Or you can bunching all the 4 devices together to create a single context and like that.

And then in the next part here, what we have is the things that you do after the contexts are created across different platforms,. If I am having a very simple architecture, I may be creating context inside a single platform. And then you create data objects associated to the different contexts. And then you create kernel object based on a kernel specification, which is encoded inside a character array and that is compiled to create the kernel objects after all this what you have is you have programs steady as objects.

Which will do the compute and you have data segments also setup as memory objects. So, that are containing the data on which the programs are going to the executables are going to compute. So, what is the logical next step? The next step would be that you have to set up a sequence of commands which are to be executed. What do I mean by commands comments? It means when and how to write, run which kernel on what memory object produce the result, it to which memory object followed by which kernels execution, all this stuff.

So, to give you an example, let us say I want to implement the following computation. I have kernel A and kernel B. They produced data and that is going to buffers. Once the data is ready, then I want some kernel C to fire and produce the data and moving into a buffer. And you have these input buffers. So I have a sequence of operations. So I need to write the data into these buffers. The most important you need to remember that initially, you do not have you just have these buffers there.

And you have these kernels available to you as program objects. These are available as memory objects, buffer objects. So you need command to write data here. You need command to execute these kernels you need command who will let you know that these kernels have executed and

produce data. And then you need another command to execute this kernel. And again, you have to quit the system to know that this kind of execution is done so the data is ready here.

So that means I need a sequence of commands somewhere, because if the commands are set up in a wrong way, then everything is going to go wrong. So, this leads me to the idea of command queue, through which all these comments will be sequenced. And that will provide me and specific order in which the operations have to be carried out. So that this required functionality is actually implemented. These are important.

So I am just trying to motivate why I need this comment queue, this idea of a comment queue. Because unless I have a queue like structure for storing comments, I do not have any way to force this kind of dependencies and sequences in the execution. Of course, I also need something else. Which is for example if I take this thing that this guy is going to execute only when the data is ready here and here.

So this would be naturally like a synchronization point in the execution. So how do I specify which are the points where the programs need to synchronize? That would require not only execution commands, or read or write commands, but also synchronize commands. So in this way, I am just trying to motivate what are the different kinds of commands that are going to be part of the command queue.

So once this is done, let us say you get the data ready here, you know that things are done, you need to clean up the resources that you have consumed in a system that would mean you need to clean up this well, first, you need to clean up this program. I have just done everything. So in what form is essentially a kernel present? As we have said that finally inside the host program, the kernel has to be available as a string of character type array.

In general, when we are specifying the kernel, it is just like a normal C code with some restrictions and extensions. There by there will be some OpenCL constructs which has been pushed in the form of the functions which are implemented by the runtime. And also you have to understand that

these are data parallel code. So essentially in the C code what you write is all portrayed via just like a CUDA function.

(Refer Slide Time: 11:24)

```
OpenCL Host Code
Initialisation and declaration

#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
#define LENGTH (1024) // length of vectors a, b, and c

int main(int argc, char** argv)
{
    size_t dataSize = sizeof(float) * LENGTH;
    float* h_a = (float *)malloc(dataSize);           // a vector
    float* h_b = (float *)malloc(dataSize);           // b vector
    float* h_c = (float *)malloc(dataSize);           // c vector (result)
    cl_int err; // error code returned from OpenCL call

    // Fill vectors a and b with random float values
    int i = 0;
    for(int i = 0; i < LENGTH; i++){
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
    }
}
```

So, with this will first take an example of our host code. So, this was our example of a kernel code. And now, we will take an example of a host code. So, essentially is just like a normal C program as you can see you include stdio.h, stdlib.h and then you have OpenCL specific declarations in cl.h. So, these are the declarations that are kind of specified by the working group. The joint working group called Khronos.

So you may visit this website. So this is essentially the Joint Working Groups website, who actually specifies all the standards and specifies all these functions for OpenCL. And what will be their behavior and all that. Now, whenever some vendor will confirm to this behavior, they will provide the library which will contain the implementation of the functions which realize this behavior.

So, let us just have a look into the program like what is really going on. So as you can see, the initial part is just like a normal C program. You are declaring 3 spaces using memory spaces, by using malloc you are allocating their memory for h a, h b and h c corresponding to this data size. The length of the vector that you did like to work on. And then let us say we are just using a code to fill up these vectors by randomized initialize values.

(Refer Slide Time: 13:10)

The screenshot shows a presentation slide with the title "OpenCL Host Code Cont." at the top. Below the title is the code for setting up a platform:

```
Set up platform:  
// Set up platform  
cl_uint numPlatforms;  
  
/*cl_int clGetPlatformIDs(cl_uint numEntries,cl_platform_id *platforms,  
cl_uint *numOfPlatforms)*/  
  
err = clGetPlatformIDs(0, NULL, &numPlatforms); // Find number of platforms  
if (numPlatforms == 0)  
{  
    printf("Found 0 platforms!\n");  
    return -1;  
}  
cl_platform_id Platform[numPlatforms];  
err = clGetPlatformIDs(numPlatforms, Platform, NULL); // Get all platforms
```

Once that is done, now comes the platform specific programs. So let us have a look at how this platform is created. It is very important. Please pay attention here. This is the point from where things become a bit different from our normal C program running in a desktop computer. So, first, you see the usage of OpenCL type here. So you have a `cl_uint` for OpenCL kind of unsigned integers.

So that would be storing the number of platforms. So you are initializing a variable `num_platforms` for storing the number of platforms that would be available in a system. So forgetting that you make a call to this function, `clGetplatformIDs`. So, this is the function which are called. Now this is definitely the kind of declaration we have put in the comments there.

So, as you can see in this commands, we are specifying that this is asking for the following number of entries and then you are hoping that it will actually give back the entries. And so that would be coming in another output, which is the `cl_uint` type pointer to number of platforms. And then also there is another argument we will see what why this is useful. So, first thing you do is you make a call to `clGetplatformIDs` with the number of entries the first parameter set to 0. And the second parameter which is the type platform ID.

So there is another OpenCL type for platforms. There is also set and you are just passing the address of this variable, which is a `cl_uint` or unsigned integer type, which is the `cl_platforms`

variable. All that is going to happen. If this call is set up in this way, with the first parameters being 0 and now, it will essentially write the total number of platforms into this variable. All that is all literally, suppose there are only 2 platforms available, it does very well we will get a valid 2, if it is only 1 platform, which will get a value 1 like that.

In case it is 0, we have written some handler code that then it will just return back from here. Because it is finding 0 platforms that means there are no platforms at all. Now, suppose you have figured out through this call that what is the number of platforms here? Now you are going to use exactly the same call again, to figure out what are the different platforms. Just knowing that i.e. how many platforms you have is not going to help you.

You need more information like how to access each of these platforms. For that, you again make a call to the same function, you can see the functions of `clGetplatformIDs`, but the arguments are different. Initially you give the first argument is 0 because you simply did not know. So, that is like an initial value you are giving. But now, you know, how many platforms are there through this style of call to the function. Now you are calling it in a different style, you are supplying it the value of number of platforms.

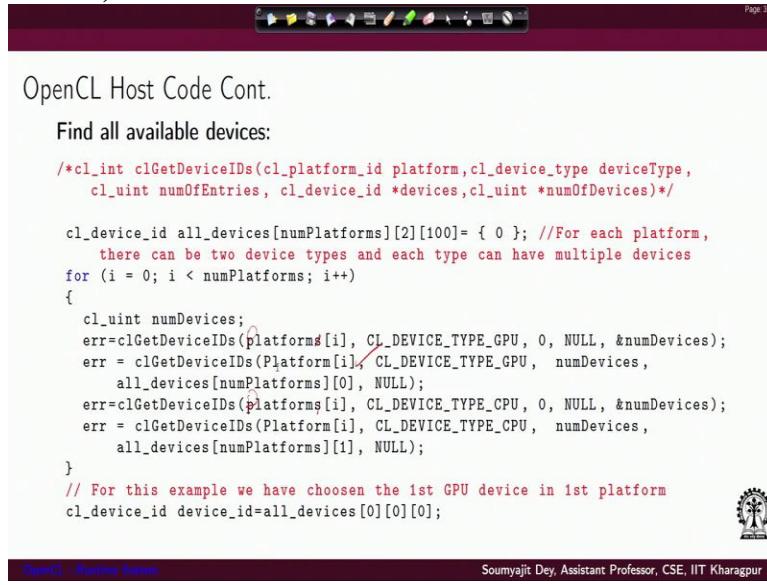
And you are, setting up something here which you are passing the variable that would be platform and its type is `cl_platform_id`. So of course that needs to be declared here, and that is something you have to do. So, suppose you have this. So, you have an earlier set it to `NULL`, suppose you have already declared this, that what is here platform id, and then you have given this variable name, this is essentially a pointer out here.

And you are now setting the last thing to `NULL`, because now you want this function to act in a different way, because you already know how many platforms are here. But you really want to copy back these platforms ids to some structure, which is of type `cl_platform_id`. We have been declaring each here. So you have `cl_platform_id` type for this variable `platform`, which is essentially an array and by knowing a proper initialize value of `num_platforms`, you are able to declare it.

So, this is like a dynamic way you are declaring, you are able to declare what should be the size of the platform at it. And you are so in a way, using that size, which is now known to you, you are simply declaring platform of size number of platforms. And it is of type platform id, because you expect this array to contain the number of platform ids. So, you have these many entries, each of the entries should be a valid platform id.

So when this call will be set up in this way? When it will return without any error, then this platform array will get filled up with all the device the all the ids of the platforms in each of these positions. So you can see that you are using `clGetplatformID` to first recover the number of platforms that are there. Accordingly you are declaring the dynamic array. And then you are populating that array in different ethe positions in the array with the platform IDs, which are being recovered using the second call.

(Refer Slide Time: 19:12)



OpenCL Host Code Cont.

Find all available devices:

```

/*cl_int clGetDeviceIDs(cl_platform_id platform, cl_device_type deviceType,
                      cl_uint numEntries, cl_device_id *devices, cl_uint *numOfDevices)*/

cl_device_id all_devices[numPlatforms][2][100] = { 0 }; //For each platform,
// there can be two device types and each type can have multiple devices
for (i = 0; i < numPlatforms; i++)
{
    cl_uint numDevices;
    err=clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_GPU, 0, NULL, &numDevices);
    err = clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_CPU, 0, NULL, &numDevices);
    all_devices[numPlatforms][0] = NULL;
    err=clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_CPU, 0, NULL, &numDevices);
    err = clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_GPU, 0, NULL, &numDevices);
    all_devices[numPlatforms][1] = NULL;
}
// For this example we have chosen the 1st GPU device in 1st platform
cl_device_id device_id=all_devices[0][0][0];

```

Page 3/8



OpenCL Host Code Cont. Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, let us inspect this code very minutely, this has been written in this way just to give you a general feel, how like how things occurred there. So, you know, now, what are the platforms using their IDs? But as we know that in OpenCL, we are assuming that the underlying system is very complex, like there are the things from multiple different vendors. And each of these platforms may contain multiple devices, like 1 CPU, 1 GPU, or multiple GPUs, multiple CPUs in 1 device in 1 platform and again in another platform multiple such devices and all that.

Till now all that you have got is the platform ids. So now, you would like to get the device IDs for each of the platform IDs for this you are going to use this function `clGetdeviceIDs`. We have provided here the prototype of this function. So, it will be taking the following parameters, you are going to ask the system, what is the device ID is available for some specific platform. So, the first parameter is of type `cl_platform_id`, where you are going to push the specific entry of this earlier platform array.

And then you are also going to provide it with the type of devices that you wanted to discover. Maybe you were interested in only a set of only a specific type of device like CPU or GPU, then this will give you those device ids. And then you also have number of entries. You provide it a pointer of type `device id`, which you wanted to figure out and provided the device id there and then you have another pointer for number of devices.

This is also a type `unsigned int`. Let us see how things get set up. See now that we have identified that what is the number of platforms. So, using that. So, this is the number of platforms, which you have now figured out from this call, and you have declared this and you have figured and filled up this platform with a different platform ids, you make the declaration of another 2d array called `all device id`.

So the use of types here is `platform id`. These are type `cl_device_id`, why is that the 2d added because for each platform, you can have technically multiple devices. So we are just making a declaration like this. I mean, these are static is a reason. So for each platform, we are expecting in architecture, the devices are of 2 types? So we are writing it 2 of course, this will this is not generic I am just repeating.

We are simply assuming in this case that the devices are only have type CPU and GPU. We make it so, and we are giving an upper bound on the size here, which is also statically defined. So for each platform, there can be 2 device types. And each type can have multiple devices, which would be stored in these positions? Now let us see how that thing works. We are going to browse through acquiring the system.

And we will for each of the platforms, we will query the system for device IDs and we will get the device IDs. There is a code that will go into run and this is how it works. So we have a follow through which will run from $i = 0$ up to the number of platforms types, each of the values of the iteration, what is going to do is you so suppose you are writing for 0.

So essentially you have got the first platform ID at platforms 0 position of the array. So, now you fire the `clGetdeviceIDs` call using this call you have you have also provided it with let us say platform 0. So, the first entry with the `platform_id` and decided for this platform you are trying to discover devices of type GPU and here you also pass it with the `num_devices` which has been declared here immediately before the call.

So in every iteration you will get a variable here called `num_devices`. So with `num_devices`, this will give you back the number of GPU devices that are available in that platform 0. I am speaking of things going on the $i = 0$ th iteration. So with this call, so as you consider, the structure is again similar, you have the same call, same kind of call made twice. First you kind of discover what is the number of GPU devices that you have in this platform and then you make the second call again with `num_devices` initialized. So it looks similar to platform.

And then so just pardon us, we are being inconsistent like this is platform and so this should be let me make the correction here. So the issue is, as you can see, I have we have here declared platform with P caps. So let us just follow that here. And I believe you should be able to resolve it. This is fine. So, with this first call, we get the number of GPU devices in platform in the first platform, then we provide that value here. And we are expecting this system earlier, as you can see that 2 of these arguments were 0.

And now, you are actually setting them up the first argument, because if I am initializing the number of device to 0, now I know the number of devices, so I put it here. And then I want those device IDs to be returned? So since my queries, what are the number of devices of type GPU in platform 0? Now I know that now I want to know that what should be and what are the device IDs for those GPU type devices in platform 0. So essentially here I am providing it the point to all

devices, num platforms. In place of 2 we will have 0, the let us say the first part is for storing the GPU IDs.

And then so that would be the pointer. And so I would expect with this call, this array would get populated with the different GPU device IDs for platform with ID 0. Now, in the next again, I will just repeat the same thing. So again, you are firing the get device IDs call? Using these get device IDs called you are figuring out what are the what are the more different I mean, how many GPU or CPU type devices are there in these platforms.

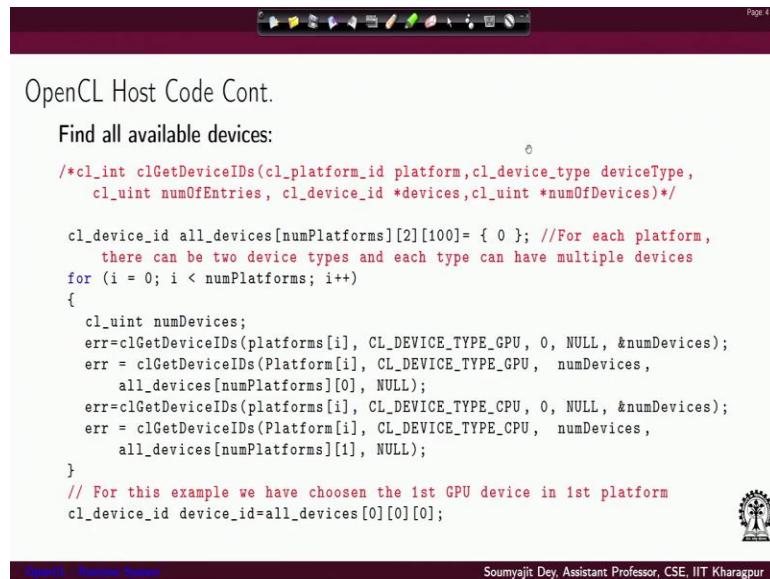
How many CPU type devices are there? In this platform, we were essentially figuring in platforms 0 and then for all those devices, you are storing back their device ids. Inside this all_devices, non platforms, for CPU it will have 1 here. And all the rest of the positions in the array will be filled up with the CPU device IDs? As you can see, this is a 2d array. So if I am associated 2d level essentially have a pointer.

Since we have passed it, the function will ride back inside this array sequentially in these positions like num platforms [1][0], num platforms[1][1] in all these positions, it is going to ride back the CPU device IDs for platform 0. So with this iteration of this loop, I have discovered as it iterate through all the platforms which have discovered all devices for each platform, device IDs for CPUs, device IDs for GPUs, and all that together. For this example we have chosen the first device to be a GPU device in the first platform. So, maybe with this part, we will be ending this lecture. See you soon. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 44
OpenCL - Runtime System (Contd.)

(Refer Slide Time: 00:33)



```
OpenCL Host Code Cont.

Find all available devices:

/*cl_int clGetDeviceIDs(cl_platform_id platform,cl_device_type deviceType,
cl_uint numEntries, cl_device_id *devices,cl_uint *numOfDevices)*/

cl_device_id all_devices[numPlatforms][2][100]={ 0 }; //For each platform,
// there can be two device types and each type can have multiple devices
for (i = 0; i < numPlatforms; i++)
{
    cl_uint numDevices;
    err=clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_GPU, 0, NULL, &numDevices);
    err = clGetDeviceIDs(Platform[i], CL_DEVICE_TYPE_GPU, numDevices,
    all_devices[numPlatforms][0], NULL);
    err=clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_CPU, 0, NULL, &numDevices);
    err = clGetDeviceIDs(Platform[i], CL_DEVICE_TYPE_CPU, numDevices,
    all_devices[numPlatforms][1], NULL);
}
// For this example we have chosen the 1st GPU device in 1st platform
cl_device_id device_id=all_devices[0][0][0];
```

Hi, welcome back to the lecture series on GPU architectures and programming. So, I believe, in the last lecture we have been discussing in detail about simple OpenCL host program, which is actually looking into the architecture and discovering the different devices. And let us try and figure out now how after discovering the devices things are going to work. So the way will be interesting. The presentation is that before going into any segment of the code will be again kind of doing a small recap of the concepts.

We have discussed earlier, but we will do it in a more formal way, the properties of those concepts and why they are useful. And then we will be showing the corresponding program segments there so we will structure. So if you recall from our last lecture, here, what we did was, we have discovered the different devices, we are storing the GPU type devices for each platform. In the first part, we are storing the GPU type device_ids.

And then the second part of the id we have the CPU type device ids. So for plot platform 0, the first device, in all devices, as you can see all_devices[0][0]. In this, the first device that will you have is going to be a GPU device. So let us say we get the device id here in our device type in a device id variable of type cl_device_id.

(Refer Slide Time: 02:01)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Context'. Below the title is a list of bullet points. At the bottom of the slide, there is footer information including the OpenCL logo, the text 'OpenCL - Runtime System', and the name 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' next to the IIT Kharagpur logo.

- ▶ The context is created and manipulated by the host using functions from the OpenCL API
- ▶ It coordinates the mechanisms for host-device interaction
- ▶ It manages the memory objects that are available to the devices
- ▶ It keeps track of the programs and kernels that are created for each device
- ▶ Limiting the context to a given platform helps to fully utilize a system comprising resources from a mixture of vendors (ICD: "Installable Client Driver")

Now, let us see how this can be used to do some real work. So, if you remember from our earlier lecture, so we discussed setting up a context and then building up a context using devices that you slowly keep on discovering in the system and all that so now we are going to really do that using OpenCL code. So what is really important is to remember a context is created and manipulated by the host using functions from OpenCL API.

So it is basically an OpenCL host program, which is defining the context using the platforms and devices. And then inside the context, the host program will perform interactions with the device inside the context. It is the memory objects that are also managed. And because you can manage you can share data among memory objects sitting inside one context because that is like an abstraction. So for all available devices inside the same context, the management of memory objects will be happening inside that context itself.

It also keeps track of the programs and kernels that are created for each device. So whatever devices you have created for which, what are the programs and kernels that are there, they are all

managed by that context. Now, typically, that rule is that you should limit a kernel to a given platform. Because then you can fully utilize the system comprising resources from a mixture of vendors, because each vendor will provide you an ICD or by installable client driver opens your implementation of that vendor.

So that is the lower level library, which will be running and managing the OpenCL code in that vendors platform. So if you need to limit the context, specific to vendor specific platforms, and inside the context, you can have multiple devices working together. Which are also part of that vendor distribution.

(Refer Slide Time: 04:08)

Page 5/5

Command Queues

- ▶ Creates a data structure called command-queue to co-ordinate execution of the kernels on the devices.
- ▶ The command-queue can be used to queue a set of operations (referred to as commands) in order.
- ▶ Before OpenCL 2.0, commands could be enqueued only from host. OpenCL 2.0 allows both host-side and device-side command-queue that allows a child kernel to be enqueued directly from another kernel executing on a device
- ▶ To create a host or device command-queue on a specific device, API used is `clCreateCommandQueueWithProperties`. (`clCreateCommandQueue`)
- ▶ We can set the property parameter of the Command-Queue in the `clCreateCommandQueue` command

Soumyajit Dey, Assistant

Now, if you remember earlier we discussed the idea of command queues like why do we require a queue here so, that you can give an ordering of the commands and all that so, that is also from a programmers point of view that is also a specific kind of data structure that would be needed to define for coordinating the execution of the kernels. Now, inside the command queue, you can actually specify what are the queuing or what are the queuing actions to be taken, which are the commands to be executed in which order and all that.

Now, before OpenCL 2.0 commands will be included only from the host that means only the host can specify which kind of read writer kernel execution operation is to be done, but from the OpenCL 2.0 specification both hosts side and device side command use these concepts have come

up like a code that is executing in a device like a device code like a kernel is executing in a OpenCL device from that itself, I can create child kernels, which can included directly.

So, will we like to summarize like the device side command queue, you can have a host side command queue, you can also have a device set command queue. And this allows that you can spawn a child kernel from an execute from a kernel which is already executing on a device and, and you can allow the child kernel to be enqueued directly from the currently executing kernel.

So this is essentially known as device side enqueueing, maybe we will touch upon it later on. Now let us go straight to the simple basics here. So in order to any of these commands, use either a host side command queue or a device command queue on some specific device, there is a specific API that is used called `clCreateCommandQueueWithProperties`. So, first of all let us understand you need to operate in a part you can have command queue to define that many levels, you can have command queue in a pod device manner.

You can have multiple command queues giving, commands to a device. So in what are the possibilities? We will see all those now. You can see, we can also set the property parameter of the command queue in the CL create command queue command like what kind of executions will be supported and all that.

(Refer Slide Time: 06:47)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Command Queues Properties". Below the title, a text block states: "The Command-Queue Properties that can be set are-". A bulleted list follows:

- ▶ `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` : Determines whether the commands queued in the command-queue are executed in-order or out-of-order ..
- ▶ `CL_QUEUE_PROFILING_ENABLE` : Enable or disable profiling of commands in the command-queue.
- ▶ `CL_QUEUE_ON_DEVICE` - Indicates that this is a device queue. Only out-of-order device queues are supported.
- ▶ `CL_QUEUE_ON_DEVICE_DEFAULT` - Indicates that this is the default device queue and used with `CL_QUEUE_ON_DEVICE`. There can only be one default device queue per context.
- ▶ If not specified, an in-order host command queue is created for the spec

At the bottom of the slide, there is a footer bar with the text "OpenCL™ Parallel System" and "Soumyajit Dey, Assistant".

So this is just like a basic introduction to the structure called command queues. Now, what are the properties that are supported? For example, you can have a property enable, which is CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, if you are setting this to be true that determines that the commands in the command you can execute out of order. So, it will be executing out of order. Also you will be interested in doing some runtime profiling like what amount of time exactly is taken by some read/write or execute kind of operation in the device.

So, you can enable the profiling by using this flag CL_QUEUE_PROFILING_ENABLE to indicate that some command queue is a device specific, you can have this flag enabled. So only in such device queues, there is something important that will feel for your facilitator. They are only out of order queues are possible in devices. Now, this other property CL queue on device default, which indicates that this is the default device queue and to be used with CL queue on device.

We will see some examples now. There can be only 1 default device queue per context. So inside a context for the device, you can have 1 default device queue? So these are all again, things specific to device queues like in a device, you can have a queue with out of order enable for our device, you can have only 1 default queue like that. Now, if it is not specified, and in order host command queue is created for the specific device, nothing else is specified, you have created a command queue for a specific device.

That is the host side command queue, which is going to execute commands in the execution in the order in which they have been queued not in out of order way. So there is a default that if you do not specify something else, for a specific device, you have a host site in order command queue.

(Refer Slide Time: 08:53)

OpenCL Host Code Cont.

```
Create context and command queue
/* Ideally for each platform (from different vendors), there will be one context
   and each device has one command-queue. Single device can have one or
   multiple command-queues but a command-queue can be associated with only
   one device (To be discussed later). For given example we have created only
   one context and command-queue for the chosen GPU device */

// cl_context clCreateContext(const cl_context_properties *properties, cl_uint
// num_devices, const cl_device_id *devices, void CL_CALLBACK *pfn_notify, void
// *user_data, cl_int *errcode_ret)
cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

// cl_command_queue clCreateCommandQueueWithProperties(cl_context context,
// cl_device_id device, cl_command_queue_properties properties, cl_int *
// errcode_ret)
cl_command_queue commands = clCreateCommandQueueWithProperties(context,
device_id, 0, &err);
```

So first of all, we will just see such a simple examples. So you create context and command you how does that work? So, ideally for each platform which comes from different vendors, there will be 1 context and each device will have 1 command queue, for each platform. There is important from different vendors from each platform you have 1 context and each device will have 1 command queue. So, using that command queue ,give commands like read/write execute kind of commands to the device, single device can have one or multiple command queue.

Now that is also important you can have multiple command queue for the same device. But what is not possible is a command queue can be associated with more than 1 device. I cannot have a command queue which is churning out command to execute in multiple devices . We will discuss this simulator. So for this example, we have created only 1 and will create 1 command queue for the single GPU device which we choose. Write here this 1 so, here as you can see, for the 0 platform.

We have chosen the fastest GPU device and afford that GPU device will be doing all these definitions of context queue and all that so that I can create a context pack and tag the GPU. Inside the context, create a queue, which will be enqueueing commands targeting that device and all that. So 1 thing one must appreciate, while we go through these concepts is how genetic this system is . I will just repeat that you should not get overwhelmed with such lot of definitions.

Let us understand that this has been kept like this to make it generic. You can execute so many different kinds of devices coming up, if they are you have multiple devices arrange together you can so this gives you a way to organize their architecture, scheduled programs on them and create dependencies and all that. Now, of course, the other important point would be while this code looks so difficult, most of these are kind of repetitive in the context of program execution.

So, those can just be copied if you have one implementation available with you. So, the first thing that would come is how to create a context? We have been talking about context. So this is a call for that `clCreateContext` which is going to return you an `cl_context` and it can go through several parameters, which are specified here. Now, initially, when you are just creating the context, we will just discuss that for we just specify what are the devices to be attached with. In this case we are just given one device.

So we just have a single pointer to that `device_id`. If I have a collection of devices, this would essentially be pointers for that so I have just created a context with 1 GPU device. In general, as I am repeating, I can create a context with multiple devices, multiple CPUs and GPUs, kinds of devices and all that by creating an app device id array and passing it here the next thing that comes is I have to create the command queue.

So, this would be the command for that CL create command queue with properties. So, you are creating a command queue for this context, which has been provided. And since we have discussed so that are inside our context, I can have multiple devices. For each device, I need to have at least 1 command queue, which will issue commands for the device. So will, create now that command queue for the context.

And also for some specific device in the context I am trying to impress upon why we should also have the device id here. Because although here I am creating a context with a single device. I could have created a context in general with multiple devices. So then when I am creating command queues, I need to specify for which contexts which device I am creating the queue.

(Refer Slide Time: 13:09)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Program Object". Below it is a bulleted list:

- ▶ An OpenCL program consists of a set of kernels that are identified as functions declared with the kernel qualifier in the program source.
- ▶ The program executable can be generated online or offline by the OpenCL compiler for the appropriate target device

At the bottom of the slide, there is a video player interface showing a person speaking. The video controls include play, pause, and volume. The video title is "OpenCL - Working System" and the speaker's name is "Soumyajit Dey, Assistant".

So, the next thing would be a program object. So an OpenCL program consists of a set of kernels that are identified as functions declared with the kernel qualified in the program source. So, it will write the OpenCL kernel with this qualified, kernel in this in before the declaration. As you can see in our OpenCL kernels example now, the program executable can be generated online or offline by the OpenCL compiler for appropriate target device as you can see, 1 option is like I said earlier, the kernel can be specified inside in a way that finally the host program would view the kernel as a character as a stream.

So it will look at it like that and it will compile it and then create the program object and then create commands around it. Now, this idea of creating the program object, we are saying that it can be generated either online or offline by the OpenCL compiler for a suitable device. In this case, we just saw as many the host code where the kernel would be the core from the kernel source code. Creation of the program executable will be done online during the execution of the program.

(Refer Slide Time: 14:31)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Program Object". Below it, a text block states: "A program object encapsulates the following information:[†]" followed by a bulleted list. The list includes: "► An associated context", "► A program source or binary", "► Successfully built program executable, library or compiled binary", "► List of devices", "► Build option used", and "► Number of kernel objects currently attached". At the bottom of the slide, there is a navigation bar with "OpenCL™ Parallel Systems" and "Soumyajit Dey, Assistant". To the right of the slide, a video player interface is visible, showing a man in a blue jacket speaking.

So, what I mean what essentially is there in that program object so, it is less like an encapsulation of the following information like it would necessarily need and context that the program object is also aided with this context. That means it can access the data offers and devices, I mean, on that context. It can be enqueued in the command queues which are on that context like that it will also have a binary approach to execute and the program engineer, also the successfully built program executable a library or a compile library.

And it contains the list of devices on which as I am saying that can execute, what are the build options for the program object. Now, we are not discussing that too far. You can look at it later on and what are the number of kernel objects that are attached so you can have multiple such kernel objects for the problem object.

(Refer Slide Time: 15:28)

The OpenCL APIs used to create a program object for a context-

- ▶ `clCreateProgramWithSource`: Load source code into that object
- ▶ `clCreateProgramWithIL`: Load code in an intermediate language into that object
- ▶ `clCreateProgramWithBinary`: Load binary bits into that object
- ▶ `clCreateProgramWithBuiltInKernels`: Loads the information related to the built-in kernels into that object

OpenCL - Runtime System Soumyajit Dey, Assistant

Now, for creating such a program object for a specific context, there are some functions which need to be supported by OpenCL API. And they are essentially this CL create program with source the function of this would be that it loads the source code into that object. You have defined that object and you are loading the source code there. `clCreateProgramWithIL` load the code the intermediate language into that object again we are not going into the details of this.

Now and next thing is `clCreateProgramWithBinary`. You load the binary bits into objects. So essentially depends on in what form the program is available to you . Like I said a string, we will see some examples, or is it available in the form of an intermediately compiled language like from the source you have made some compilation to in kernels of representation. And from that you want to create the program.

For the program focus, we want to create a program object or even the binary is available, and you want to just create them from CL program binary. So, as I am said, his id of OpenCL online or offline execution would mean that. You may perform runtime compilation, you may perform runtime intermediate codegeneration or you may have the binary available. For all those 3 possible options. Your host program will actually use 1 of these options to compile the program to actually create the program object using the format.

In which the program is available whether from source, from intermediate level code or from the binary for creating the program object and then you have CL build program create program with built in kernels. So, this loads the information related to build in kernels you may have kernels which are already built in to load information about them because for those are already available to them. So, the program object and does not need to create those kernel objects but obtain the information about them.

(Refer Slide Time: 17:39)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Program Object'. Below it, a section titled 'Other OpenCL APIs related to Program Object-' lists several APIs:

- ▶ **clBuildProgram**: Build (compile and link) a program executable for all the devices or a specific device(s) in the OpenCL context associated with the program
- ▶ **clCompileProgram**: Compile a program's source
- ▶ **clLinkProgram**: Link a set of compiled program objects and libraries and create a library or executable
- ▶ **clGetProgramInfo**: Return information about a program object
- ▶ **clGetProgramBuildInfo**: Return build information for each device in the program object

At the bottom of the slide, there is a footer bar with the text 'OpenCL - Runtime System' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right, along with the IIT Kharagpur logo.

Now, there are some other OpenCL APIs which are related to program objects. For example, you have CL build program so, what does it do? It essentially compiles and links a program executable for all devices or a specific device in the OpenCL context associated with the program and then FCL compile program which is compiling the Source Link program. So, essentially as you can see, you can build the overall process or you can compile program, and then you can link a set of compiled program objects and libraries and create the overall library or executable.

Then you have **clGetProgrammingInfo** for which can actually return you back some information about an already compiled and created program object or you can see I will get program building for which will give you build information about the process that was followed and all that for creating the program object for each device. So these are like last.

So the first 3 will be standard compilation link programs, useful if you want to do these things by the host program. If you want the information extracted from the project that is available in the context in the runtime, then you would need to execute this last 2 commands.

(Refer Slide Time: 19:02)

The screenshot shows a presentation slide with a dark purple header bar containing icons. The main title 'Kernel Object' is centered at the top. Below the title is a large, empty white area where bullet points are listed. At the bottom of the slide, there is a footer bar with the text 'OpenCL - Training Session' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right, along with the IIT Kharagpur logo.

- ▶ A kernel is a function declared in a program in OpenCL.
- ▶ A kernel is identified by the `__kernel` qualifier applied to any function in a program.
- ▶ A kernel object encapsulates the specific `__kernel` function declared in a program and the argument values to be used when executing this `__kernel` function.

So what is the kernel object? Kernel function is declared in the program? The kernel is identified by this `__kernel` qualifier as we have seen earlier. Now this corresponding kernel object encapsulates this function, which is declared in the program. And also it encapsulates the argument values to be used when the kernel function would be executed.

(Refer Slide Time: 19:30)

The screenshot shows a presentation slide with a dark purple header bar containing icons. The main title 'Kernel Object' is centered at the top. Below the title is a large, empty white area where bullet points are listed. At the bottom of the slide, there is a footer bar with the text 'OpenCL - Training Session' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right, along with the IIT Kharagpur logo.

OpenCL APIs related to Kernel Object-

- ▶ `clCreateKernel`: Create a kernel object
- ▶ `clCreateKernelsInProgram`: Create kernel objects for all kernel functions in a program
- ▶ `clSetKernelArg`: Set the argument value for a specific argument of a kernel
- ▶ `clGetKernelInfo`: Return information about a kernel object
- ▶ `clGetKernelArgInfo`: Return information about the arguments of a kernel
- ▶ `clEnqueueNDRangeKernel`: Enqueue a command to execute a kernel on a device

So just for manipulation of kernel type objects, you again would have some specific OpenCL API support. For example, you want to create the kernel object or you want to create kernel objects for all kernel functions in a program of course, you can have multiple kernel functions. And the next important thing is, if you want to set the argument values of a kernel. Now, what I mean you, we know that for a function we can wrap to specify a set of arguments.

Now the way it is done for an OpenCL Kernel is you fire multiple instances of OpenCL kernel argument to you with each instance you add in a new argument, for a specific argument of the kernel. And if you want to get information about the kernel object that will do that you can get from `clGetKernelInfo`, if you want argument information about the kernel you can get it from `clGetKernelArgInfo` for returning information about the arguments of a kernel.

And then if you want to enqueue a command, which will actually execute a kernel in our diversity. In a context for that the actual command is `clEnqueueNDRangeKernel`. So you are enqueueing substances and enqueueing of a command as we have understood, unlike CUDA well just launch a kernel here things are a bit more system specific. So even a kernel execution is a generic open command.

So it is equally known by the name `clEnqueueNDRangeKernel`. Since it is a queuing operation, you do not really get to execute it directly. You enqueue, an execution type command in a command queue. That is why you call it `clEnqueueNDRangeKernel`.

(Refer Slide Time: 21:19)

OpenCL Host Code Cont.

```
Create and build program object

/* cl_program clCreateProgramWithSource(cl_context context,cl_uint count,const
   char **strings,const size_t *lengths,cl_int *errcode_ret) */
cl_program program = clCreateProgramWithSource(context, 1, (const char **) &
   KernelSource, NULL, &err);

/* cl_int clBuildProgram(cl_program program,cl_uint num_devices,const
   cl_device_id *device_list,const char *options,void (*pfm_notify)(
   cl_program, void *user_data),void *user_data) */
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

/* cl_kernel clCreateKernel(cl_program program,const char *kernel_name,
   cl_int *errcode_ret) */
cl_kernel ko_vadd = clCreateKernel(program, "vadd", &err);
```

OpenCL Runtime System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So let us see how the OpenCL host code actually continues from the point where we left where we just identified the platform ids and the divided device ids. So and also we created a context with a single GPU drive device. So the first thing we will be doing is we are going to build the current program object. So for that will be calling the `clCreateProgramWithSource` command, where we will be providing this as you can see, ignored the other command but observe these are command which is the characters command.

So, essentially kernel source is like a character array, which is containing the actual source code of the kernel. So with this program you get started with this function `clCreateProgramWithSource` you are expecting that character to past content, the real source code and it is creating the program out of it. Now, once you have the program object, since in this type of compilation we are following here that you are going to build a program object.

From the source that is why we fired this function as we discussed earlier, if you are already having the binary available, or the intermediate code available the other options of CL create program would be used. As we have seen earlier in this case, we are just showing this example and then once the program object is ready, you like to build it that is compile and link. So since the object is with name `program` of type `cl_program`, these are type for program objects.

The next you execute this command to build it so, once that building of the program object is done, you have to create the actual kernel objects which are going to be enqueued for execution. So now you pass this program object that has been built to the clCreateKernel command and it will return you this OpenCL Kernel type object ko_vadd.

(Refer Slide Time: 23:36)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Buffer Object'. Below the title is a bulleted list of five points describing buffer objects. At the bottom right of the slide is a small video frame showing a man in a blue jacket speaking. The video frame has a play button in the center. The bottom of the slide features a navigation bar with the text 'OpenCL - Working Session' on the left and 'Soumyajit Dey, Assistant' on the right.

- ▶ Stored as a block of contiguous memory and used as a general purpose object to hold built in types (such as int, float), vector types, or user-defined data used in an OpenCL program.
- ▶ Can be manipulated through pointers much as one would with any block of memory in C.
- ▶ A buffer object stores a one-dimensional collection of elements.
- ▶ Elements of a buffer object can be a scalar data type (such as an int, float), vector data type, or a user-defined structure.

So this is how you create a kernel object finally, using the CL create kernel, before that you build a program before building the program, you create the program object. So again, I will just repeat you we are considering that kernel is available as a source in a character array, use that to create program object. Use that to build the program use that to create the kernel once the kernel object is created, now, the other thing is just like kernel, you also need the memory space to be declared as buffer objects.

So, these are stored as block of contiguous memory like normal C. And they are used as general purpose object to hold built in types like in float vector types or you can also have a user defined type in open scale problem, they can be manipulated to pointers, much as one node with any block of memory in C and of course, also, it stores essentially it is a 1 dimensional collection of elements like elements of our object can be as scalar data type, or a user defined structure.

As we know now one thing I would like to point that most of this information we have mined out from the Khronos specification, that is that is available in the khronos website. So that is quite

openly available. Most of this information has been mined out from those sources and also the references we are provided at end of this presentations.

(Refer Slide Time: 25:01)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Buffer Object". Below it, the text "APIs used for buffer objects-" is followed by a bulleted list of six commands:

- ▶ `clCreateBuffer`: Creates a buffer object
- ▶ `clEnqueueReadBuffer`: Enqueue commands to read from a buffer object to host memory
- ▶ `clEnqueueWriteBuffer`: Enqueue commands to write to a buffer object from host memory
- ▶ `clEnqueueCopyBuffer`: Enqueue a command to copy a buffer object identified by source to destination buffer
- ▶ `clEnqueueFillBuffer`: Enqueue a command to fill a buffer object with a pattern of a given pattern size
- ▶ `clEnqueueMapBuffer`: Enqueue a command to map a region of the buffer^I into the host address space and returns a pointer to this mapped region

At the bottom left is the footer "OpenCL - Runtime System" and at the bottom right is the name "Soumyajit Dey, Assistant". A small portrait of Soumyajit Dey is visible on the right side of the slide.

Now, if I am trying to create some buffer objects, just like for kernel objects, we have a set of commands to execute for buffer objects. Again, we have some commands to execute. So first you have the CL create buffer, which will create the buffer objects you have been buffer, to which you can enqueue commands pertaining to read or write to be done on buffer objects. So, for example, CL in queue read buffer, these enqueue commands to read from a buffer object to the host memory.

So that is important, you have the host you have 1 of the devices identified as your host device. So you want to read back data that has been computed on the device by an OpenCL kernel. Now the buffer object is pertaining to the memory region in that device in which the OpenCL is executed. So to read from that, to the host side memory, you have to execute command I will say that unlike a CUDA command for read/write to and from the host program, things are like here you have to enqueue or read command.

So you have to read from the device side, buffer object to the host side, memory you enqueue or read command CL enqueue or read buffer, you have to write from the first memory to the device buffer object. You will have a ceiling to write buffer object from the host memory. Now, of course, you have to remember that we are not talking about normal GPU things here like CPU, GPU, best

execution, like it can also be that the host and the device have a shared memory space on which things are going on.

We will see some examples of such kinds of devices later on now coming here, so apart from CLN, to read and write buffers, you also have this `clEnqueueCopyBuffer`. So essentially, you can copy a buffer object identifiable source to destination or some destination buffer, you can also fill a buffer with a pattern if you give a pattern and the pattern size. And also you have the map operation to map a region of the buffer object into the host address space and return a pointer for this map region.

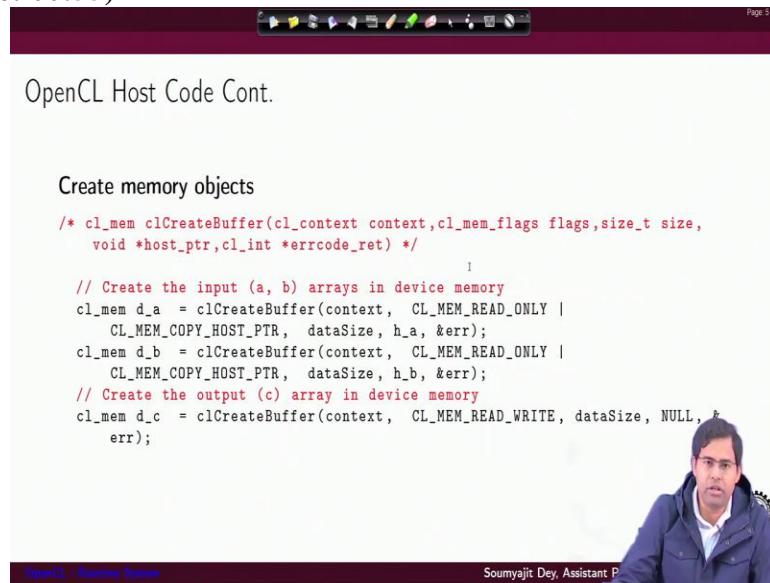
So, you have a region of the buffer object which you can map to the host address space and return the pointers. So that is also something that will be possible. So these are the different buffer manipulation commands that are there maybe we this will like to end this presentation. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 45
OpenCL - Runtime System (Contd.)

Hi, so let us welcome back to the lecture on GPU architectures and programming. So if you remember in the last lecture we have stopped midway in the kernel and buffer object creation commands that are available in OpenCL. Now so once we have got those definitions in place, so let us actually use them to create the read write buffers in which we can install a read from read from and write to where for the OpenCL kernels. So, these are will be creating a CL memory object.

(Refer Slide Time: 00:55)



OpenCL Host Code Cont.

Create memory objects

```
/* cl_mem clCreateBuffer(cl_context context,cl_mem_flags flags,size_t size,
void *host_ptr,cl_int *errcode_ret) */
{
    // Create the input (a, b) arrays in device memory
    cl_mem d_a = clCreateBuffer(context, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR, dataSize, h_a, &err);
    cl_mem d_b = clCreateBuffer(context, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR, dataSize, h_b, &err);
    // Create the output (c) array in device memory
    cl_mem d_c = clCreateBuffer(context, CL_MEM_READ_WRITE, dataSize, NULL,
        err);
```

Soumyajit Dey, Assistant Professor

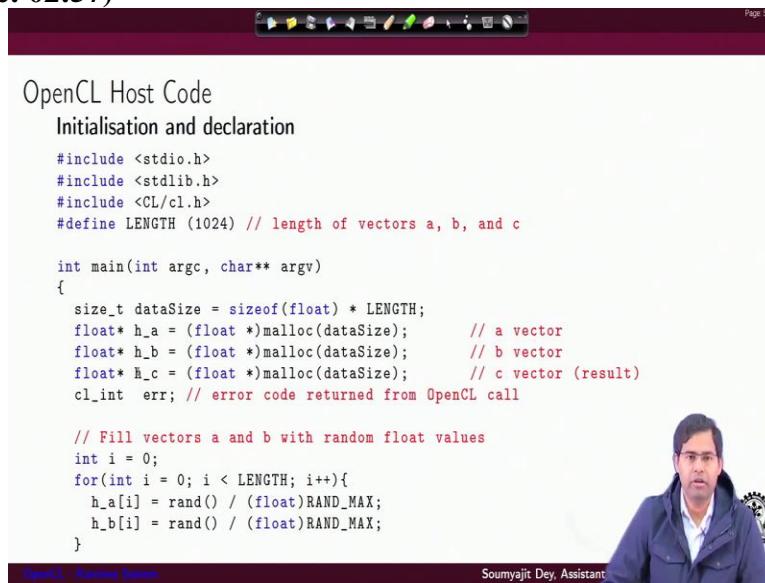
So you have a data. So, suppose you are creating the input arrays in the device memory, where you will transport data from the host program. So, you execute the commands `clCreateBuffer` in that context and for the device and what you do is you have to specify the operation type. So, this is specified that it can either to `CL_MEM_READ_ONLY` or it can be `CL_MEM_COPY_HOST_PTR`.

So, this flags this memory flags actually tell that what are the operations? That are allowed in the buffer like you are allowed to copy data from the host side. So, with that you are creating these

two OpenCL memory objects d_a and d_b using the CL create buffer command inside the context. So, the buffers are created inside the context will be usable by the devices in the context you also create the output buffer where the system would be writing the output as you can see that again you create the CL create buffer in the same context.

But the memory flag is said CL_MEM_READ_WRITE. Because now for this you also want that device side code to write its output in OpenCL kernel. And the size of the buffer is given by this data size parameter using as you can see that size_t type argument is there and of course, they have to provide the host pointers that they are going to get information from which your host site code now, which host site memory objects, but if you remember, so just to recall things.

(Refer Slide Time: 02:57)



OpenCL Host Code

Initialisation and declaration

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
#define LENGTH (1024) // length of vectors a, b, and c

int main(int argc, char** argv)
{
    size_t dataSize = sizeof(float) * LENGTH;
    float* h_a = (float *)malloc(dataSize);           // a vector
    float* h_b = (float *)malloc(dataSize);           // b vector
    float* h_c = (float *)malloc(dataSize);           // c vector (result)
    cl_int err; // error code returned from OpenCL call

    // Fill vectors a and b with random float values
    int i = 0;
    for(int i = 0; i < LENGTH; i++){
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
    }
}
```

OpenCL: Handling Systems Soumyajit Dey, Assistant

We have got this whole side array is, h_a, h_b and h_c defined. So the h_a and h_b are containing the input arguments, and h_c, we plan to write back the data after their computation in the device side OpenCL code. So, here you have also specified that what are the operations to be done? And they have to be related with which host side data structures, h_a, h_b and all that. And then in this context, you have also created the buffer where you want the output back that is d_c.

(Refer Slide Time: 03:38)

OpenCL Host Code Cont.

```
Set kernel arguments

/* cl_int clSetKernelArg(cl_kernel kernel,cl_uint arg_index, size_t arg_size,
const void *arg_value) */

// Set the arguments to our compute kernel
err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
err = clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
err = clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
err = clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
```

OpenCL Host Code Cont.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, now the other thing that we will do is you want to pass arguments for the kernel. So, if you remember, do you have the OpenCL kernel, they are available with you and this is the kernel object that we have created with the build kernel command earlier that we have discussed. Now, for this kernel object to our secreting this argument least by calling this function as many times as are the number of arguments.

So you want these to be provided with the 2 input buffer pointers, the output buffer pointer and the number of elements bound? So that is why you have 4 arguments. And you have these multiple calls. Now, the issue is why do we have multiple calls, it may happen that in each of these ideas, each of these initiatives of trying to provide arguments to this kernel, you face an error? So for each of those operations, you can have separate calls here.

So that in case you get errors in any one of them, it will be flagged out here. So I can write some handler code out of this that if in any of these operations, I get an error. If they do an order of this, then you exit from here.

(Refer Slide Time: 04:50)

OpenCL Host Code Cont.

Calculate global and local work size

```

/* cl_int clGetDeviceInfo(cl_device_id device,cl_device_info param_name,
size_t param_value_size,void *param_value,size_t *param_value_size_ret
) */

const int count = LENGTH;
cl_uint max_work_itm_dims;
err = clGetDeviceInfo( device_id, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof
(cl_uint), &max_work_itm_dims, NULL);
size_t *max_loc_size = (size_t*)malloc(max_work_itm_dims*sizeof(size_t));
err = clGetDeviceInfo( device_id, CL_DEVICE_MAX_WORK_ITEM_SIZES,
max_work_itm_dims*sizeof(size_t), max_loc_size, NULL);
size_t global_work_size = count;
size_t local_work_size=1;
for (i=0;i<max.work_itm_dims;i++)
    local_work_size*=max_loc_size[i];

```

OpenCL Host Code Cont.

Soumyajit Dey, Assistant

Now the issue is I want the kernel to work, but like a normal radar parallel program as we have seen that I need to pass it the work dimensions of the kernel what is the global work size and what is the local work size for this kernel. So, for this we have to identify still now whatever we have done is we have identified platforms or devices, but we need to identify what is the dimension, the kernel execution that a device can support.

Now, as you can see that this is something again which is very generic in OpenCL specific like inside a device, I can have support for multi-dimensional kernel and up to what dimensions specification of multi-dimensional kernel is allowed for the device may be device specific. So, the first we need to figure out what is the maximum work item dimension that is allowed in the specific device again, and just saying that vectorization is not necessary.

We are trying to give you as much information system level information as possible. To this kind of code, so, for that we have this count variable, which is storing the length of the inputs on which you want to work. Now, here is the important function that will be using CL get device info, it is going to give back some CL type integers and you are passing it some device ID OpenCL device ID and you are passing it some parameter which you want to query that what is the parameter you want to query.

And then you did expect some values to be returned by this function and let us see the function working to get some more idea. So, you have also defined another you have also defined another integer here unsigned integer, which is which you wanted to store the maximum work item dimension, which is supported by the device? So Remember just a brief recall in CUDA you had grids and thread blocks, I mean grid containing thread blocks.

Here, you have a global work size and you say inside that you have work groups inside the work group you can have multi-dimensional arrangement of work items, you want to figure out in a given device, what is the maximum dimensional specification that is allowed for that for that device. So, for figuring that out, you have the max work item dimensions, you make a call to `clGetDeviceInfo` for the specific device ID you pass it this instruction that may, you have to return to the maximum work item dimension you support.

So, you give this flag here, `CL_DEVICE_MAX_WORK_ITEM_DIMENSION` and you want that value to be returned. And so, of course, the other thing you do is you provide here as high specification of the parameter value and that would be a type that so, you have a `size_t` specification here which is like the size of what is the size of a `uint` or unsigned integer of OpenCL data type. So, here you are calculating that size. By size of operations and C and that gives you a value of type `size_t`.

And that is something you are passing here because you want the parameter values size should be known that it perform the function will give you back the parameter value size because that is also something that may be device specific. And so, once that is given, you are also passing it a pair of pointer here, which is going to stand that value. Now observe the thing, this pointer is for data of types `cl_uint`. So, you have to pass the pointer here and you have to also pass the size of the datatype for which the pointer will be initialized.

So this is a plain and simple design wise integer type pointer, you want to give these to the low level and get devising phase of quite low level function it needs to know from the runtime that. When I returned back some value what should be the maximum allow length for the value. So, since it is real type on size. I calculate using size of what is the number of bytes allowed for that.

So that I can get to know in what size I will return with value and the last parameter returns null for this call will see it working later on.

So, with this call, what do we get is the maximum work item dimension parameter initialized like this for this device this is the maximum dimension that is allowed. Now, once that dimension information is provided to you, what you do is you create you are just creating a pointer here max local size and you are locking it with dynamically for each location size of size_t and what do you really want to understand that in each dimension first of all I have already figured out what is the maximum work item dimension, what is the next information I would like to know?

In each dimension, what is the maximum value that is allowed like they work items so, suppose the work item dimension is 3 so, we know that I can have work items packed with entities of x, y and z dimensions or less as 2 dimensions. Now, I want to know, what is the maximum value I can have in this dimension? What is maximum I can have in this dimension to know that I create this max_loc_size array?

And the important thing is for these area how many positions do I need for storage, or you need these many positions each have size_t. And finally, it will return back a pointer pointing to this size of n size_t will of course, so, there is a size of here and for the malloc call, you are not providing it the parameters that that is you give me an array containing space enough to hold data of this type and for these many dimensions.

So, with this since, let us say I am going coming back to the example, if the max dimension is 3, essentially I am going to initialize an array, which can work 2 values? And that is what this call is doing. Why do we want to do that? Because again, so you can see the pattern here. Again, I am going to make a call to clGetDeviceInfo for the device ID. But now I provide it with a different parameter. Earlier, I queried it with work item dimension.

Now I am questioning it with maximum work item sizes. Of course, the implicit thing is in each dimension that is why now the return will not be to a single pointer, but now the return to be an error? I mean, it is not the return value was not returned to a single point earlier to a single position.

Now it has to be a set of values that needs to be written. So that is why the return type has not changed as you can see.

You are expecting this thing to be filled up this max local size you are expecting this to be filled up. So earlier, it give you back this parameter value size. Now, you are expecting it to so, earlier you give it, it just gives you the maximum dimension. Now, you have given it an array for storing the maximum dimensions in each of the maximum values possible in each of the dimensions. So, for that you have max local size which has been initialized and past year.

And here you have given it the size of this array of course, in the C type which will function. So, you just not just do not give the pointer here you also give me the maximum size which you already know by the way from the max marketing dimension multiplied by the size of by of the silence value t. So, with this, you expect that the runtime system will now contain in different positions of max_loc_size 0 max_loc_size 1 and max_loc_size 2 like that it will contain the maximum value of a local thread ids in each of those dimensions.

So, once this is known, you will like to figure out your global works, I mean, you like to figure out how many what is your local work size and all that. So, of course, the global work size is count because that many values are there to be operated on. Now, how does your local work size the all you simply want to do is you have to just identify what are the maximum work item dimensions in each of the work item dimensions what is the maximum value which is already stored in different arguments of this array and you just loop over the array and multiply the values to get the local work size.

(Refer Slide Time: 14:44)

OpenCL Host Code Cont.

Writing input data to device from host

```
// cl_int clEnqueueWriteBuffer(cl_command_queue command_queue,cl_mem buffer,
    cl_bool blocking_write,size_t offset,size_t cb,const void *ptr,cl_uint
    num_events_in_wait_list,const cl_event *event_wait_list,cl_event *event)

//Write the data from host to the compute device
err = clEnqueueReadBuffer( commands, d_a, CL_TRUE, 0, sizeof(float) * count,
    h_a, 0, NULL, NULL );
err = clEnqueueReadBuffer( commands, d_b, CL_TRUE, 0, sizeof(float) * count,
    h_b, 0, NULL, NULL );
```

Soumyajit Dey, Assistant Professor

So, once this is done, that is you have the local work size figured out. The next thing is you are going to write input data to the device from opposed now, the real game of executing the candidate begin. So, for that, you have to start you have to again you understand that an OpenCL everything is through the command queues through which you give commands to the ecommands to the commands queues, those command are going to be executed by the runtime system.

So you have to Enqueue commands for reads and write. The first thing would be that you want to enqueue the host side values to the device side buffers in the context that you have set up. So near context, that you have the command queue command that you have already set up in this command, queue, you are enqueueing read buffer commands, to read from the host side, array h_a to the buffers that you have defined inside your context, which are d_a.

So just being back this way or declarations of buffers memory objects inside your context of execute. And they have already been told that what are the host pointers from where they are which they are going to read? Now you just include the read operations here. So you went to read operations for the two arrays d_a and d_b.

(Refer Slide Time: 16:15)

OpenCL Host Code Cont.

Executing the kernel on device

```

//cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,cl_kernel
    kernel,cl_uint work_dim,const size_t *global_work_offset,const size_t *
    global_work_size,const size_t *local_work_size,cl_uint
    num_events_in_wait_list,const cl_event *event_wait_list,cl_event *event)

// Execute the kernel over the entire range of our 1d input data set
err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global_work_size
    , &local_work_size, 0, NULL, NULL);           0

//global_work_size (local_work_size) point to an array of work_dim unsigned
values that describe the number of global (local) work-items in work_dim
dimensions that will execute the kernel function.

```

OpenCL Runtime System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Once this is done, the next thing would be that you have to execute the kernel. So again, you have to Enqueue the execution command for the kernel. Using clEnqueueNDRangeKernel, there is the enqueueing gives these execution command in the command queue for the kernel ko_vadd does a kernel object we have created earlier. And now you provide it with the global and local work size, global workspace was already known.

We needed that generic code to figure out what is the local work size allowed for this device? Because things in it can vary across devices. So in this case, for this dynamic, using this dynamic is of course. We have figured out what is the work item dimension and then we figured out in each of the dimensions what is the maximum work item size, we are multiplying them to get the local work size.

So, we have provided these global and local work sizes and now the kernel we expect the kernel execution command to be executed by the command queue to be launched to the device and getting executed. So, these global work size and local work size they point to an array of work dim, unsigned values that describe the number of work items in the work dimension that will execute in the kernel function will see examples of this.

(Refer Slide Time: 17:39)

The slide has a dark header bar with various icons. The main title 'Work-pool' is centered at the top. Below it is a bulleted list of four points:

- ▶ The work-groups associated with kernel-instance are placed into a logical pool of "ready to execute" work-groups called work-pool.
- ▶ OpenCL does not constrain the order how work-groups are scheduled for execution in the actual device from the work-pool
- ▶ Once in the work-pool, independent execution of work-groups can happen in any order and could be interleaved
- ▶ For each device there can be only one work-pool used by all command-queues associated with that device

In the bottom right corner of the slide, there is a small portrait of a man, Soumyajit Dey, Assistant, with the OpenCL logo above his name.

So next comes the idea about the work groups. So essentially, the work group that you have. So in OpenCL when you have launched a kernel now with the global and local work size information, the work groups associated with the kernel instance are placed into a logical pool of ready to execute work groups called work pool. Again, I will just repeat one thing just in case you are wondering why we did all this thing to figure out the local work size essentially we figured out what is the maximum number of what is the maximum work group size that is supported by the device.

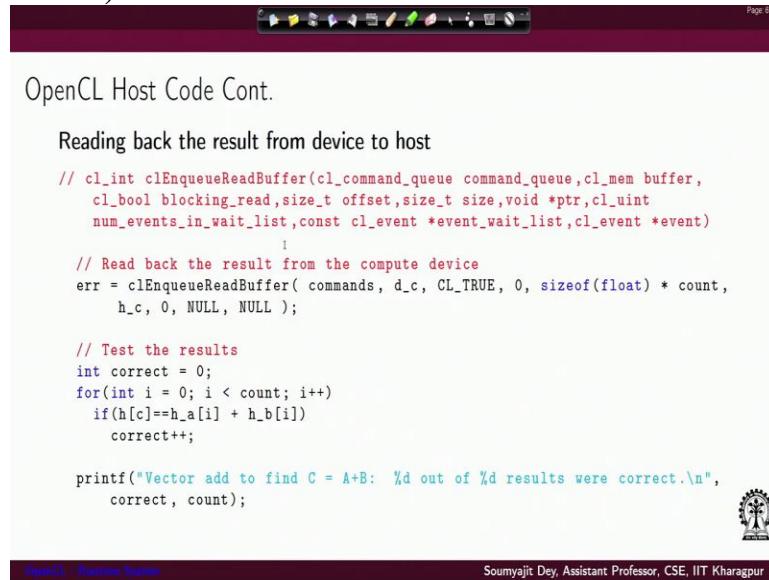
If we pass it here, then we are expecting that the devices will execute very efficiently it will pack as many work-items as possible in the local work groups and execute them. So, that gives you the most efficient execution. So, this work pool is essentially the ready to execute work groups of course have to understand that just like in CUDA, you can have, you can launch multiple thread blocks, different thread blocks can be at any at any given time point in different states of execution like that in OpenCL you have work groups associated with the kernel instance.

And you are essentially creating a logical pool of ready to execute work groups and they are called work pool. Now OpenCL does not constrain the order on how work groups are scheduled for execution in the actual device from the work pool. So there is just like CUDA, you leave it to the onboard scheduler and runtime system to manage this execution of workflows. So, once in the

work pool, you have independent execution of work groups, which can happen in any order and they can interleaved among each other.

So, some work pool execute up to some point and then maybe they are waiting for some long latency operation to complete and by that time some other work from other work or pool can be launched and they can be executed and so on, so forth. For each device, there can be only one work pool used by all command queues associated with the device. So this is important for each device in the OpenCL runtime system, you can have only 1 work pool, which is used by all the command queues associated with the device. There cannot be more than that many that work. I mean, you can have multiple work pools which are used by this command queues which are associated there.

(Refer Slide Time: 20:09)



OpenCL Host Code Cont.

Reading back the result from device to host

```
// cl_int clEnqueueReadBuffer(cl_command_queue command_queue,cl_mem buffer,
    cl_bool blocking_read,size_t offset,size_t size,void *ptr,cl_uint
    num_events_in_wait_list,const cl_event *event_wait_list,cl_event *event)
{
    // Read back the result from the compute device
    err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0, sizeof(float) * count,
        h_c, 0, NULL, NULL );

    // Test the results
    int correct = 0;
    for(int i = 0; i < count; i++)
        if(h[c]==h_a[i] + h_b[i])
            correct++;

    printf("Vector add to find C = A+B: %d out of %d results were correct.\n",
        correct, count);
}
```

OpenCL - Handling System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

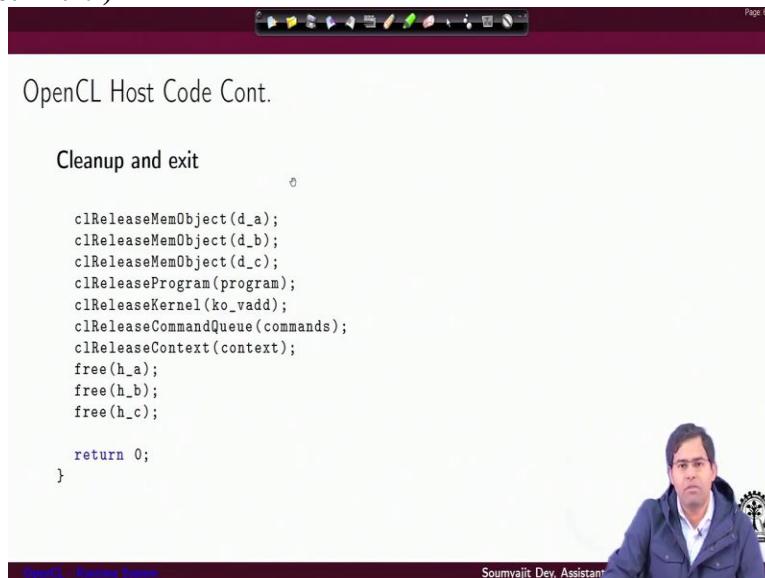
So now let us come back to the execution of the host program that we are discussing. So once we have in NDRanged the kernel, this execution need to be finished before the reading back the values that have been actually written by the kernel. So, something we are expecting that the kernel is going to read the specification of this kernel is that is going to read its input arguments, because we have if you remember, we have already provided the arguments.

So it knows what is the location from where it is going to read and of course, by the definition, the kernel is going to back in the `d_c` memory buffer. So, at this point, we know that after the kernels execution for including a read buffer commands, so, once the conversation completes, I can get

the data in d_c transferred back to h_c. So, that read buffer command is executing now include in the command queue.

So, again, you will see that read command also do not directly read. You enqueue the read command in the command queue. Here some small test code here for testing whether the values are really correct or not after the communication and go for execution in open cell runtime. So you can do this is something as simple host size code, it will execute in the host size and you can just compare whether the h_c value that have been computed by the device size code, they compare with the host size additions.

(Refer Slide Time: 21:49)



OpenCL Host Code Cont.

```
Cleanup and exit
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseProgram(program);
    clReleaseKernel(ko_vadd);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

OpenCL - Runtime System Soumyajit Dey, Assistant

Now once this is done, as we can see that you are done with your main computation. So before for leaving the system you would like to free all the objects that we have created. So, you have `clReleaseMemObject` command to actually release all the memory objects, you have `clReleaseProgram` to release the program object and you can also release the kernel object and can release the command queue.

And finally, so, this has to be done in this sequence you remove memory objects then you do program object once you know kernel is done and the command queue is also released, then you remove the context finally. And also, you release the host side arrays h_a, h_b, h_c , becomes free from all the defined and allocated data and the new return back gracefully from the cont.

So, as you can see that these are quite complex OpenCL host code, but again I will repeat the complexity primarily comes from the way we are viewing a complex architecture by ourselves. First of all assuming that the architecture is complex and we have to execute a lot of commands to create the context of execution, we can select, which are the devices from different vendors that I can pack in the context and all that.

But we have to understand that most of this code is going to be fixed? If you are going to use that same device, same platform, same GPU or same CPU for programming with some other execution? You just need to change few things. Maybe your definitions of buffers, your definition of kernel and all that setting of platform setting up context and everything can remain same, unless and also if we are going to use that device again, maybe you do not even need to change the command queue and all those definitions.

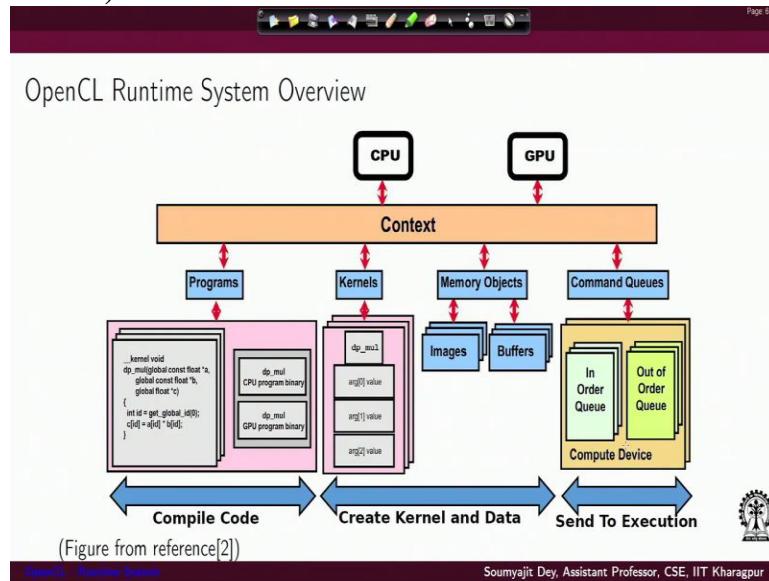
So the point I am trying to make is most of the code of for the host side is going to remain same, if you are going to write another functionality. Write, another set of kernels are orchestrated execution of another set of kernels in the same platform defies command you write so most of the code is going to be same as a developer, only thing you need to do is , modify the other parts of the code.

So that brings the question of this CUDA versus OpenCL issue with all these things and why do OpenCL at all? Well, let us remember that CUDA programs will execute only on NVIDIA GPU. But OpenCL code will execute in any system where your devices are compliant with the khronos specified OpenCL specification. It can be a collection of CPUs even it can be Nvidia GPUs, it can be AMD GPUs, Intel CPUs, AMD CPUs or some other devices.

Some other accelerators which support in all these OpenCL specification and all that. And as we know that Xilinx based FPGAs also support OpenCL. So it can be a heterogeneous system comprising multiple such devices. So there is the advantage so you do not get specific to one vendor. But that also makes it much more system level programming. Of course, if you are using OpenCL bindings like C++ extensions or the few OpenCL the Python extensions of OpenCL.

Then maybe in your program all these low level details you do not need to do you can write a much simpler OpenCL program. With this we have the specification of the host program given here at the lowest level that is a when as c level OpenCL program just to provide you with the more intricate device specific details of an actual OpenCL problems execution.

(Refer Slide Time: 25:46)



So, coming to my summary that we like to make here. So what really is the runtime systems overview what are the different phases that you have gone through so we are trying to show a simple example here. You have a CPU and GPU .These are the 2 devices, let us say you have merited them inside a context, we know how to do it. Now, you have this programs, the programs sources.

The kernel sources, and you can have from the kernel source, you can compile them to create CPU binaries you can compile them to create GPU binaries using suitable different compilation systems or you can actually even use our methods to create kernel programs from the sources? And then you can actually use our methods from using the program objects to create the kernel objects, then, so that is there is a sequence.

You have sources from then you can create program objects from them, you can create kernel objects, then you parameterize the kernel with the different arguments here. So using the set kernel argument commands, you can specify the memory objects that are to be so. All these things go

inside the context of execution this context means everything is here every execution orchestration will be happening inside this abstraction.

Why do we again define this abstraction? Technically we are viewing a more complex system there can be other CPUs, GPUs encapsulated inside another context type object to bring in this abstraction we can have multiple context. So, inside this context we can have program objects kernel objects and we can also have memory type objects of course, they are required.

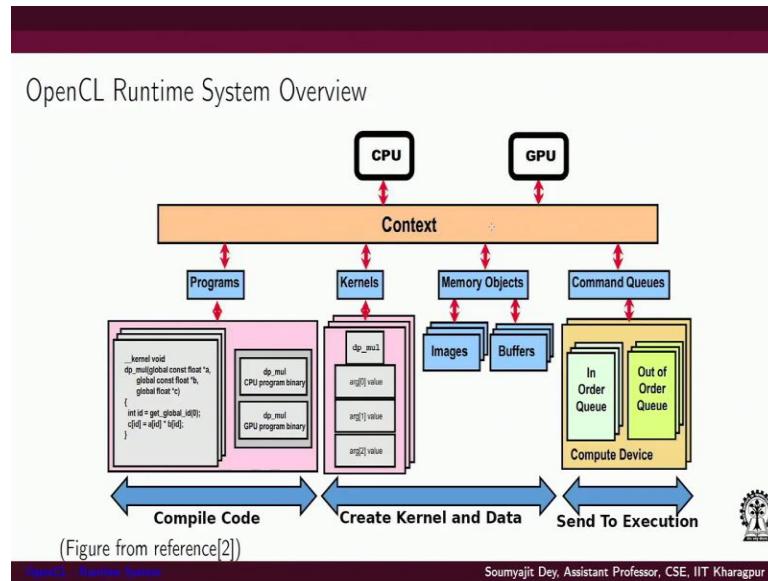
So, that would be normal memory buffers OpenCL also provides insight into internal support for an image type memory objects which are already there. So, with this you have the data read write parts ready, where from to read what kind of data to read and where to write, you have the execution kernels ready and they are created from a program objects. And then the important thing is, as we are saying that you do not really launch the commands, but the runtime systems are going to launch the commands.

But the order in which things would happen you can specify using the command queue that will read write execute in this order. And when you execute those commands, you can actually give instruction whether execute them in order or whether it be executed out of order like that. So, these are the different phases through which a typical OpenCL programs execution would go through. With this will be ending this lecture. Thank you for your attention.

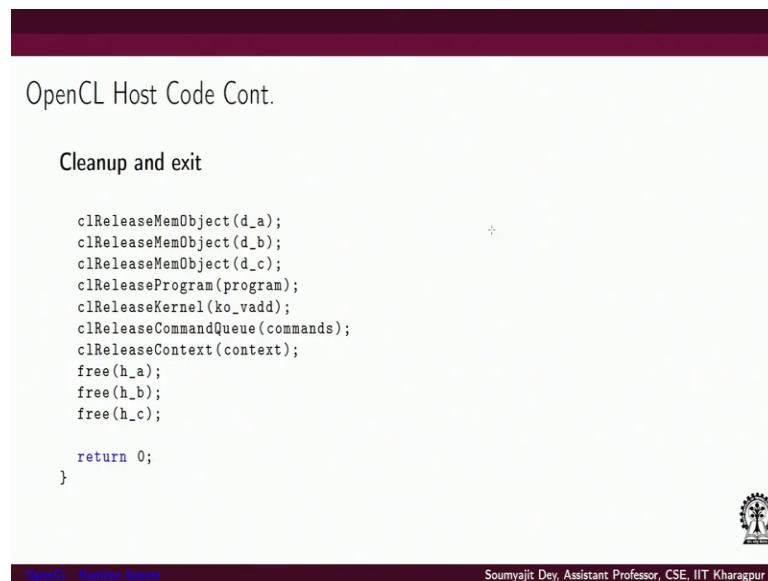
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 46
OpenCL - Runtime System (Contd.)

(Refer Slide Time: 00:33)



(Refer Slide Time: 00:34)



Hi, welcome back to the lectures on GPU architectures and programming. So I believe in the last lecture we provided you with the basic open CL runtime overview. And we gave an example of a complete open CL program enqueueing the host code primarily the host code I would say like how

to compile the code how to build a create the kernel, the different data buffers memory objects, and finally, set up a context and create command queues and then execute the kernel by issuing suitable read , kernel launch and write back and find out final result and all that.

(Refer Slide Time: 01:02)

OpenCL Synchronization

Synchronization refers to mechanisms that constrain the order of execution between two or more units of execution.

- ▶ **Work-group synchronization:** Constraints on the order of execution for work-items in a single work-group
- ▶ **Command synchronization:** Constraints on the order of commands launched for execution



OpenCL - Working Session Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So with this, we will now move over to the next topic, which is again, related to synchronization in open CL so if you remember in work coverage of CUDA programs also we had significant coverage on synchronization among threads. And it is a far more involved topic I would say, when we are talking about open CL synchronization. So here, it primarily refers to mechanisms that will constrain the order of execution between 2 or more units of execution like.

So just like if you remember in CUDA the synchronization primitive we are primarily using was `syncthreads()` using which we could actually synchronize among threads inside a thread block. And if we wanted to actually have global level synchronization that has to be managed through multiple launches of kernels through the host program. So equivalently in openCL, we have this concept of synchronization of work groups, which essentially constraints on the order of execution for work items inside a single work group.

So these are concepts, which is essentially equivalent to thread block synchronization in CUDA. And, ofcourse, we also have this idea of synchronization of commands, which gives constraints on the order of commands launched for execution. Because if you remember in OpenCL, at the

end, what you have is a command queues which are being set up with this sequence of commands to be executed.

So you do not really launch commands directly, but rather, you are just issuing commands enqueueing them into the queue. And it is the runtime systems job to pick up commands from the queue in that order and execute them. So you also have this scope of synchronizing the order of commands launched.

(Refer Slide Time: 02:53)

The screenshot shows a presentation slide with a dark purple header bar containing icons for back, forward, search, and other controls. The title 'Work-group Synchronization' is centered at the top. Below the title is a list of bullet points. On the right side of the slide, there is a video player showing a man with glasses and a hoodie speaking. Below the video player is a small circular logo. At the bottom of the slide, there is a footer bar with the text 'OpenCL™ Runtime System' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

- ▶ Synchronization between work-items in a single work-group is done using following command -
 - ▶ void barrier(cl_mem_fence_flags flags)
(work_group_barrier from OpenCL 2.0 onwards)
- ▶ Options for flags are-
 - ▶ CLK_LOCAL_MEM_FENCE
 - ▶ CLK_GLOBAL_MEM_FENCE
- ▶ All the work-items in a work-group must execute the barrier before any are allowed to continue execution beyond the barrier
- ▶ Work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all

So let us first move into the idea of group synchronization. So this is essentially about synchronization between work items as we discussed inside a single work group. So, for this the open CL function will be using is barrier, and for the barrier function takes this cl_mem_fence_flags type variable and from open CL 2.0 onwards, we also have this is a command for work group area.

So, this is a similar command, but it provides you the notion that this is about synchronization inside the work group. And for that you have this a flag as well as another field called scope. For now we will just go with the flag. Now, what are the options for flags? First of all, again, I will just repeat the barrier command or the work group barrier command equivalently OpenCLs current version, they will be synchronizing work items inside work group. In the flag part, you have 2 options.

One is this CLK_LOCAL_MEM_FENCE and you also have this CLK_GLOBAL_MEM_FENCE . The issue with this mem fences with that what is the area of the memory you want to make visible to the work items for synchronizing? So whether it is the local memory or the global memory upon that it depends what flight you are willing to choose. All work items in a work group must execute the barrier before they are allowed to continue execution beyond the barrier.

And it must be ofcourse encountered by all work items of work group executing by the kernel or none at all. Now, this is very important. First of all, let us understand that you have to have it the primary semantics of barrier is that every work item inside work group will reach the barrier and once all of them received only then the any work item can go beyond work group. The next important thing is you may be thinking well if I have a barrier inside an if else block, then maybe some of the work items will be facing the barrier.

And that was work items will be bypassing the barrier will that is not true the semantics are very at ease that you have to write the code in such a way that either all the work items are supposed to execute the barrier, I mean, they have to encounter the barrier, or everybody will bypass them. Because the barrier by it is the runtime system, by his definition of the barrier could wait for all work items to, to be encountered.

If it is partial, then it will be working continuously, whereas the other work items are not even facing it because of divergence that will create a deadlock. So the support up to the programmer to write the code in such a way to ensure this last property.

(Refer Slide Time: 05:52)

Page 1 / 1

Work-group Synchronization Example

Kernel Code

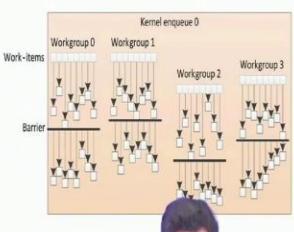
```

__kernel void simpleKernel(__global float *a,
    __global float *b, __local float *localbuf
) {
    //cache data to local memory
    localbuf[get_local_id(0)] = a[get_global_id(0)
];

    //wait until all work_items have read the
    //data
    work_group_barrier(CLK_LOCAL_MEM_FENCE);

    //perform the operation and save in output
    //buffer
    unsigned int addr = (get_local_id(0)+1) %
        get_local_size(0);
    b[get_global_id(0)] = localbuf[get_local_id(0)
] + localbuf[addr];
}

```



(Figure from reference)



OpenCL Programming System

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, just to recall here about the flags, so I believe the barriers, semantics is clear to you use the synchronization primitive just like CUDA, but also like in CUDA, the synchronization primitive the way you use it, you have to make sure that either all the problems have threads in the work group encounter it or none at all. So that you can avoid the deadlock issue. Now, regarding the fence and the flag, as I said that if you said the flag is local mem fence, then you make all the updates in the local memory visible to the fence in the work group.

And similarly, if you set it as global mem fence, then you are essentially sitting in making all the updates in the global memory visible to the work group. So that actually provides you an additional flags, you mean depends on what you are sitting here you are, directing them system that what you really want, what kind of behavior do you want with respect to the memory so the barrier is controlling the way threads synchronize and the memory fence and we are sitting controlling the memory is being feasible at what level to the synchronizing threats.

Also, I would like to add that, in the newer version of work group barrier, there is an additional flag here called scope. If you are interested you mean if you want to go more into the advanced concepts of openCL, well, you can go there. I mean, get into that, that and figure out what is this idea of scope that comes with the flags for mem fence. So now we proceed to some example on one group synchronization.

So, let us take this example of a simple kernel. So, first what you do is, you execute the simple get_global_id and get_local_id calls. Using which you are caching data from the global memory to the local memory. So I hope by now you are familiar with this, get_local_id and get_global_id calls get local ID and global ID respectively in openCL. Essentially, they are helping you to figure out what is the global ordering of the threads and what is the local ordering of threads inside the work group.

Using the local ordering of threads you are figuring out in which position of a local buffer, you would be writing a value. So once you have cached the data you want all the all the threats to reach these barrier to me to wait at this barrier until all the threads have done with the previous job assigned to them. And after this barrier synchronization with respect to the local memory updates here, you are going to the next part of the code where you perform some operation and save the output in the buffer.

So here you have defined some operation that you calculate an address based on this get local ID, get local ID, and then you do something over that value so there is some functionality here, but we are just trying to show that how things are going on. So, if you look into the figure, for this kernel, you have launched multiple work groups, we are trying to convey this fact that this position of the so this is the timescale I would say, in this y axis.

And you have all the work groups being dispatched. As you can see that the barrier is being the synchronization is actually happening at different points of time, across all groups. But inside every work group, all the threads have to hit the barriers synchronize and then move forward.

(Refer Slide Time: 09:58)

The screenshot shows a presentation slide titled "Event Object". The slide content includes a list of bullet points explaining the用途 of event objects in OpenCL:

- ▶ An event object can be used to track the execution status of a command.
- ▶ The OpenCL API calls that enqueue commands to command-queue(s), create new event object that is returned in the event argument.
- ▶ In case of an error enqueueing the command in the command-queue the event argument does not return an event object.
- ▶ Can query the value of an event from the host. For example to track the progress of a command.

At the bottom of the slide, there is a video player interface showing a person speaking. The video player has a progress bar at the bottom and a control bar at the top. The video player also displays the name "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur" and the IIT Kharagpur logo.

So, this is one synchronization primitive using barriers that you can do. Now, there are several other primitives we will see, one of the important things is openCL events. So that brings us to this notion of event objects. So an event object is used to track the execution status of an openCL command like as you know, that every openCL every execution is like enqueueing a command in a command queue.

Every commands execution will have the corresponding status and the status is can be pulled by an event. So OpenCL API calls and enqueue commands to open CL command queue and creates new event objects, which are returned in the event argument. In case of an error, if there is an error in keeping the command in the command queue, then the event argument will not return an event object but otherwise it should be successful in returning an event object.

And the API can query the value of an event from the host. For example, you can track the progress of a command by inquiring to runtime system about the status of the corresponding event the value of the corresponding event.

(Refer Slide Time: 11:11)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title 'Event Object' is at the top left. Below it, a text block says: 'Execution status of an enqueued command at any given point in time can be one of the following:' followed by a bulleted list: ▶ CL_QUEUED, ▶ CL_SUBMITTED, ▶ CL_RUNNING, ▶ CL_COMPLETE. In the bottom right corner, there is a small portrait of a man and the text 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

We will see some examples. For example, for every enqueue command, these are the following valid status that that Open CL runtime system will provide for example, it can be a it can have a CL_QUEUED. So, just to summarize, every openCL command will pass through these 4 statuses. And for this corresponding to the status, you can define an event so that means for every event, it can be attached to a command and the event will have disposable it whether a command is skewed whether it is submitted, whether it is running or whether it is complete.

(Refer Slide Time: 11:51)

The screenshot shows a continuation of the presentation slide. The title 'Event Object' is repeated at the top left. Below it, a section titled 'OpenCL APIs related to Event Object-' lists several functions: ▶ clCreateUserEvent: Create a user event object, ▶ clSetUserEventStatus: Set the execution status of a user event object, ▶ clWaitForEvents: Waits on the host thread for commands identified by event objects in event_list to complete, ▶ clGetEventInfo: Return information about an event object, ▶ clSetEventCallback: Register a user callback function for a specific command execution status. The registered callback function will be called when the execution status of command associated with event changes to an execute status equal to or past the status specified by command_exec_status. In the bottom right corner, there is a small portrait of a man and the text 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

Now the Open CL API relates to the event object and there are a few ways in which you can actually do the following functions using which you can use in an events. For example, clCreateUserEvent is used to create a user level event object. The status of an event can be set by

`clSetUserEvent` status, it will set the execution status of work user event object. `clWaitForEvent` will wait on the host thread in the host side thread for commands identified by the event object in the event list to complete.

So I mean, there are there may be events for which you are waiting and you want to implement that kind of behavior that can be done by CL wait for events, CL get event info, so this is like a query function which will return information about some event object. Now `clSetEventCallback` is related to registering of callback functions for specific command execution status. So there is usage of callback functions is they can exist asynchronously.

So, suppose you want that for a specific command, you want to figure out whether it has reached a corresponding status or not some specific status, whether it is completed whether it is pending or not. And you want that whenever that event happens that it has reached that specific status, then some thread, some specific callback function will be asynchronous, we launched in the host site as a separate thread.

Now, this registered callback function will be called when the execution status of the command associated with the event changes to some specific execution equal to or pause the status that is specified by command status. So just to make it very simple, using this function, `clSetEventCallback`, you can command a system that worked. I want some specific function called a callback function to be launched when the execution status this command becomes some things that I specify.

So when that equality is holding, that means the execution status of that command is equal to whatever I have specified, then this specific callback function will start working. Now, why is that normal function not a normal function, but I call it a callback function because it will execute because the host does not wait for it, it will be called and it will start executing asynchronously in the host side will understand it at the end with some examples.

(Refer Slide Time: 14:31)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Command Synchronization'. Below the title is a bulleted list of points about command synchronization. To the right of the list is a small video window showing a man speaking. At the bottom of the slide, there is a footer bar with the text 'OpenCL Function Reference' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

- ▶ Command synchronization is defined in terms of distinct synchronization points.
- ▶ A synchronization point between a pair of commands (A and B) assures that results of command A happens-before command B is launched
- ▶ The synchronization points occur between commands in host command-queues and between commands in device-side command-queues.
- ▶ All OpenCL API functions that enqueue commands return an event that identifies the status of command.
- ▶ Value of the event associated with the command is set to CL_COMPLETE when done

OpenCL Function Reference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So here I am just trying to introduce the definitions and concepts. Now, the next important thing that why do we use this idea of events we will figure out that they also help us to provide synchronization of commands. Now, they can be performed by in terms of distinct synchronization points. A synchronization point between a pair of commands ensures that results command happened before command this launched you may always want that I have include commands in multiple command queues.

But the execution of some command in some queue should depend or should have start when something else in some other commandshave happened, you may want some such dependencies to be satisfied. How do you do that? Well, you have to synchronize between commands. So let us try and understand this is different from synchronization among threads, like we have done in CUDA, and like we have seen as examples in open CL.

But here it is a bit different concept. We are trying to synchronize among commands that have been enqueued in the command queue. The synchronization points occur between commands in host command, queries, command queues, and between commands in device said commanders so synchronization points will occur again, I will repeat between commands in host command queues and between commands in device that command queues will see some examples between them.

And all openCL functions that enqueue commands, return an event that identify the status of the command. Like we have this earlier that every command, I mean, every command, which is being enqueued for it, an event is returned. And by pulling that event or by querying that event, I can always figure out the status of execution of that command, the value of this event, which is being associated with the command, while enqueueing, the command will be set to CL complete when the execution is done.

Now, this is very important like so, I will just summarize that all this idea of events which have we should have been defining the primary reason is you are enqueueing a command at that moment you are it will return the event that events will be one of this based on the execution status of the command. And when it is everything finishes, you will it will reach this CL_COMPLETE status.

(Refer Slide Time: 16:58)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title 'Command Synchronization' is centered above a text block. Below the text is a portrait of a man, Soumyajit Dey, and his name and title are displayed at the bottom right. The text block contains two bullet points about synchronization points in OpenCL.

The synchronization points defined in OpenCL include:

- ▶ **Completion of a command:** A kernel-instance is complete after all of the work-groups in the kernel and all of its child kernels have completed. This is signaled to the host, parent kernel or other kernels within command queues by setting the value of the event associated with a kernel to CL_COMPLETE.
- ▶ **clWaitForEvents:** This function waits on the host thread for commands identified by event objects in event_list to CL_COMPLETE. The events specified in event_list act as synchronization points.

OpenCL Parallel Systems

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now this synchronization points that we have defined in openCL they include the following for example, completion of a command a kernel instance is complete after all the work groups in the kernel and all of its child kernels have completed with all work groups must complete. So, or and we need to understand that inside a kernel you will be launching child kernels those also has to be completed. Now, only when that is done this is signal to the host, parent kernel or other kernels within command queues, so for every kernel there may be a parent kernel or other kernels.

So, this is signal by the corresponding event that was defined while in enqueueing this kernel execution command. And so, this finishing of the task of execution of the kernel is signal to the host on the parent kernel and it is done by stating of the event associated with the kernel to CL complete. So, just to summarize, you have launched a kernel, you doing that enqueueing operation of the kernel, you have got an event automatically associated to it or you can actually specify which event as it to it.

That event status will be set to CL complete when all of work groups of the kernel and also if there were any child kernel whether all those were most of all those child kernels everything is complete, only then that event associated with the enqueueing of the kernel will be set to CL complete. Now we also have this other function `clWaitForEvents` and this waits on the host thread for commands identified by event objects independent list to `CL_COMPLETE`.

Event specified in the event list will primarily act as synchronization points. So I believe this should not be there. So just a minor correction. So, when I have CL wait for events this function will wait on the host. So, when I have this function the second last one here wait for events, this is simply I mean suppose you have made it to wait for a set of events that would mean until and unless those event objects will reach this status `CL_COMPLETE` individually all the all those events, then only this function job is done otherwise it is still waiting there.

So that is another synchronization primitive. So just to understand using completion of a command, you are attaching or you may possibly attach your event definition to a kernel launch, when that kernel launch and his child kernels have all completed. You get that event being attached, we are being automatically assign the `CL_COMPLETE` flag by the runtime system. The other option to synchronize would be a code you have a `clWaitForEvents` function and you provide that function with a set of events to synchronize on.

When the runtime system ensures that those set of events reach the status seal complete, then this function will release and lead the execution move forward.

(Refer Slide Time: 20:29)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Command Synchronization'. Below it, a text block says: 'The synchronization points defined in OpenCL include:' followed by two bullet points about blocking commands and command-queue barriers. On the right side of the slide, there is a portrait of a man, Soumyajit Dey, wearing a light blue hoodie. At the bottom left is the text 'OpenCL Parallel Systems' and at the bottom right is 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

Now, apart from this, we also have this notion of blocking commands. So these are also synchronization points because we have to understand that if a command blocks that means until unless that command's execution is finished, the program straight cannot go forward. So command execution can be blocking or non-blocking. If it is a blocking function, then the events API functions that include commands.

Do not return until a command has completed some of the blocking commands are saying `clEnqueueBuffer` enqueue read buffer and enqueue write buffer both of them if they are called with the option `blocking` read and `blocking` write, we set to `CL_TRUE`. So, as you can see that when you were in making this enqueue commands have read buffer and write buffer like defined earlier, they also had a field where you can set this flag of `blocking` read and `blocking` respectively.

And in those cases, they are blocking commands that means if you do not set it, then they will fire synchronously that means without this read buffer or before finishing the host program and move forward to the next command. Because they will just keep on happening without blocking the actual flow of the execution. But if you set them to `blocking`, `read` or `blocking`, then until unless these commands are complete, the host programs execution will not go forward.

And then you have command queue barriers. They ensured that all previous commands to a command queue have finished execution before any following commands

enqueueing a command queue you can begin execution. So, you have the command queue ensures that commands start executing in that order. But if you put a barrier in between it ensures that all the command queue functions we should have started executing in their order they should finish before anything out I mean after the barrier.

Even starts executing the open CL API functions for this cl enqueue barrier with waitlist cl enqueue marker with waitlist. So these are so essentially you are enqueueing a barrier in the command queue. And of course, you can put the barrier with a waitlist that barrier first quits for the events in the waitlist, and then the barrier is active, and then all the commands previous to the barrier are forced to finish before fundamental commands can be launched.

(Refer Slide Time: 23:00)

The synchronization points defined in OpenCL include:

- ▶ **clFlush:** All previously queued OpenCL commands in command_queue are issued to the device associated with command_queue. clFlush only guarantees that all queued commands to command_queue will eventually be submitted to the appropriate device. There is no guarantee that they will be complete after clFlush returns.
- ▶ **clFinish:** All previously queued OpenCL commands in command_queue are issued to the associated device, and the function blocks until all previously queued commands have completed. clFinish does not return until all previously queued commands in command_queue have been processed and completed.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

You also have synchronization points defined by clFlush. So, you have clFlush, then all previously queued openCL commands in a specific command queue or issue to the device associated with the command queue and clFlush will guarantee that all these commands in that command queue will eventually be submitted to the appropriate device. And of course, there is no guarantee that they will complete after clFlush returns.

So it guarantees that all the commands that in the queues will be submitted to the appropriate device starting from I mean, so you have enqueue them and you have submitted mean clFlushes ensuring that all of them gets submitted to the device for execution. And then you have clFinish.

So here whatever `clFinish` commands have been queued, they are issued to the host device and the function blocks until all these functions have completed.

So unlike `clFlush`, which is providing you a guarantee that whatever commands have been enqueued earlier, they are all submitted to the device from the command queue `CL` finishes as if it is executed and it returns successful. It is guaranteeing that whatever commands were enqueued, and I mean, all of them have completed. So this function blocks until all previously queued commands have been completed.

It does not return until all the previous queued commands in the command queue have been processed and completed and so I hope this is the difference that you can understand between flash and finish.

(Refer Slide Time: 24:42)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Command Synchronization'. Below the title is a bulleted list of synchronization primitives:

- ▶ Command-queue barrier:
 - ▶ Ensures that all previously queued commands have finished execution and updating memory objects before subsequently enqueued commands begin execution
 - ▶ Can only be used to synchronize between commands in a single command-queue
- ▶ Waiting on an event
 - ▶ All OpenCL API functions that enqueue commands return an event that identifies the status of command
 - ▶ Value of the event associated with the command is set to `CL_COMPLETE` when done
- ▶ `clFinish`: Blocks until all previously enqueued commands in the command queue have completed

In the bottom right corner of the slide, there is a small video frame showing a person speaking, and below the video frame is the text 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

And then for the purpose of synchronization, as we discussed that you have command queue barriers and you have worked on an event primitive and command to barrier ensures that all previously queued commands have finished execution and they have updated whatever memory objects or memory objects for commands, which will begin execution afterwards. So, this is the primary reason why you will like to have barriers like the command queue ensures that you start issuing commands in that order.

And if you put the barrier, you can actually ensure that though they finish all the commands that have been included the actually finished and they are before going forward and this can be used for important reasons, for example, as has been specified here that you suppose you need some memory objects to be set up before further command queue execution goes forward and you want some memory objects to be updated and ready by some kernels to have executed earlier and only then you will be executing the next kernel.

This kind of execution we have here you can ensure by using the command queue here barriers. Now the important thing is inside a command queue if you put a barrier it will help you to synchronize between commands inside a single command queue and not across multiple command queues. Now, the other thing is waiting on events. All openCL API functions that enqueue commands, they return an event that identifies status of command.

Like we have discussed earlier, that events are returned by the enqueueing command operations only and the events can have values starting from waiting, submitted and all that up to CL complete when the event finishes execution. And finally, you have this CL finish synchronization primitive which blocks until all previous to enqueue commands in the command to have completed like I mean, we have already discussed the will finish and the difference between CL finish and CL flush. So, overall these are the different primitives that will be using.

(Refer Slide Time: 27:03)

Out-of-order Execution of Kernels and Memory Object Commands

- ▶ The OpenCL functions that are submitted to a command-queue are enqueued in the order the calls are made but can be configured to execute in-order or out-of-order.
- ▶ The properties argument in `clCreateCommandQueueWithProperties` can be used to specify the execution order.
- ▶ In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued.

Speaker: Soumyajit Dey
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, the next topic would be out of order execution of kernels and memory objects. So far we have assumed that, you have open CL command queue, you have been enqueued them and they can just go and execute in order. And that is not going to hold in general. Maybe we will end the current lecture here and resume from this point in the next lecture. Thank you.

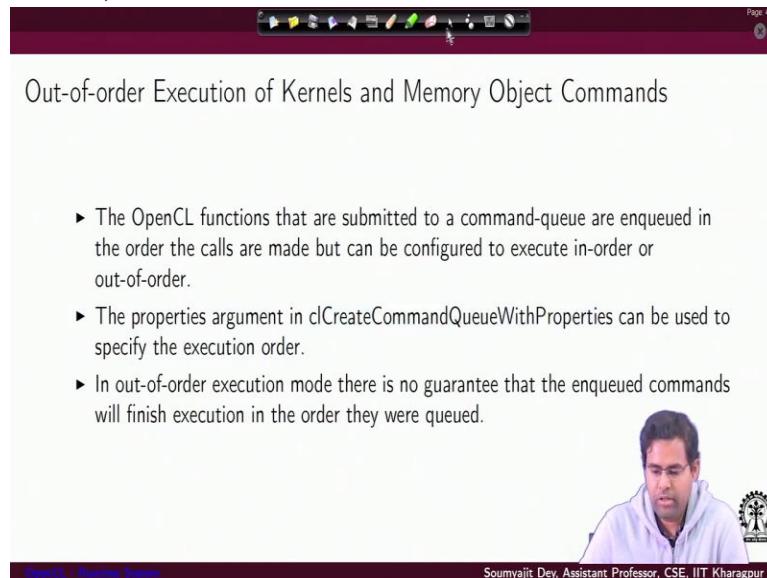
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 47
OpenCL - Runtime System (Contd.)

Hi, so, in the previous lecture on a GPU programming and architecture we have been discussing on in order execution of commands and different synchronization primitives. For example, we discussed how work groups can be synchronized, how you can put in barriers inside a command queue. And then there was this idea of how you can use `clWaitForEvents` primitive because with every incoming command you have an event return that event will have different status messages.

And this status you can make you can have a CL works for event function using which you can query the event status and unless a specific event reaches a specific status, you can also wait at those points. And you also have these primitives like `clFinish` and `clFlush`, which are also useful from that perspective. And now, the next thing we want to discuss was that well even if you commands in a command queue, we usually assume that.

(Refer Slide Time: 01:22)



Out-of-order Execution of Kernels and Memory Object Commands

- ▶ The OpenCL functions that are submitted to a command-queue are enqueued in the order the calls are made but can be configured to execute in-order or out-of-order.
- ▶ The properties argument in `clCreateCommandQueueWithProperties` can be used to specify the execution order.
- ▶ In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued.

Page 4 / 4

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

The commands will be issued to the underlying device for which the queue has been made in an in order fashion, which is not true, it will also happen that you can if you want you can execute the

commands out of order and you let the runtime system decide exactly in which order the commands will be executed. So, just to give an example, what we mean suppose this is the command queue, and you have executive and you have been queued, read command, then a kernel launch point kernel has been included.

And then, you have let me just redraw this figure suppose, I am trying this graphically. So, you have cl_command_queue write buffer followed by kernel launch , then you have a read buffer command after this, then you launch another kernel k prime like that. So, this if you have a normal command queue, which is to be executing in order, this is exactly the order in which the different commands will be launched to the underlying device right.

But, if I want, I can give the command queue an out of order property. In that case, the runtime system will decide whom to launch when. And you may be wondering why somebody would want to do that, because the runtime system we often have certain units which are free, but the command which is sitting in front of the command pipeline may not be for those units, those compute units but for somebody else.

Whereas there may be succeeding commands there who could have engaged the corresponding execution units. For example, there may be separate read write units and compute units and the compute unit is not free, but, at the same time I could have take out a read operation from the queue. And in that way I could have achieved some parallelism in using the runtime system, but for that the command queue should also be able to issue the commands in a out of order way.

And just trying to motivate how out of order execution from the command queue may give me better performance. So, the properties are whom as in these clCreateCommandQueue with properties has to be set to an out of order flag. And so, in order to specify what should be the execution order of commands from the command queue in out of order execution mode, there is no guarantee that the include commands will finish execution in the order they are queued as we have been discussing.

And so the takeaway from this would be that this is primarily being done by setting the `clCreateCommandQueue` with properties command while setting up the command queue itself.

(Refer Slide Time: 04:10)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "In-order Execution of Kernels". Below the title, there is a text block and a bulleted list. On the right side of the slide, there is a video player showing a person speaking, with a small circular logo next to it. At the bottom, there is a footer bar with text and logos.

If the `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` property of a command-queue is not set, the commands enqueued to a command-queue execute in order. For example,

- ▶ If an application calls `clEnqueueNDRangeKernel` to execute kernel A followed by a `clEnqueueNDRangeKernel` to execute kernel B
- ▶ The application can assume that kernel A finishes first and then kernel B is executed.
- ▶ If the memory objects output by kernel A are inputs to kernel B then kernel B will see the correct data in memory objects produced by execution of kernel A.

In case of out-of-order execution, there is no guarantee that kernel A will finish before kernel B starts execution.

OpenCL™ Programming Systems Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So we will see if we are talking about in order execution of kernels. Then the `cl_queue_out_of_order_exec_mode_enable` property of any command to is not set and the commands in queue are executed in order. Consider an example these are normal in order execution of kernels were talking about first you can calls `clEnqueueNDRangeKernel` to execute some kernel A followed by another call to `clEnqueueNDRange` to execute a kernel B so you have include 2 kernel on launch A and B.

The application may I assume that kernel A finishes first and then Kernel B is executed this assumption from the application side. If the memory objects to be output by A are inputs to kernel B then B will see correct data in memory object produced by execution of it, because in order execution, it will finish and then we will start executing. But in case it is an out of order execution, you do not have such a guarantee that A will finish before B starts execution.

So that is the problem, right. If it is out of order as we said if I make it out of order, then I lose such guarantee. But then again, I will like to sometimes make things out of order, so that the runtime system can give me better performance if possible.

(Refer Slide Time: 05:29)

The screenshot shows a presentation slide with the title "In-order Execution Example Code". The slide contains the following C code:

```
// Perform setup of platform, context and create buffers
...
// Create queue leaving parameters as default so queue is in-order queue
clCreateCommandQueue(context,devices[0],0,0);
...
clEnqueueWriteBuffer(queue,bufferA,CL_TRUE,0,10*sizeof(int),a,0,NULL,NULL);
clEnqueueWriteBuffer(queue,bufferB,CL_TRUE,0,10*sizeof(int),b,0,NULL,NULL);
// Set kernel arguments
...
size_t localws[1] ={2}; size_t globalws[1] = {10};
clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalws,localws,0,NULL,NULL);
// Perform blocking read-back to synchronize
clEnqueueReadBuffer(queue,bufferOut,CL_TRUE,0,10*sizeof(int),out,0,0,0);
clEnqueueReadBuffer(queue,bufferOut,CL_FALSE,0,10*sizeof(int),out,0,0,0);
clFinish(queue); // cl_int clFinish(cl_command_queue command_queue)
```

So, here we have a sample code of in order execution. So, there are certain parts which are commented because we are just training what are the functionalities you have to do there and those functionalities are things that we have already discussed earlier. So consider that you have already set up the platform context and created the required region write buffers and you have created the queue living parameters as default so that the queue is an in order queue. So the command queue was already there.

I mean, so this is something we doing here and this command is saying that you have already performed a setup of platform context and buffers and all that. And this is the command through which you are creating the command queue for a device, which has been already discovered and the device ID is stored in these device, the device belongs to this context, which has also been defined and other parameters are being set to default to maintain that desire in order queue.

So, you Enqueue a write buffer command, you Enqueue another write buffer command. So essentially you will be copying this buffer A and buffer B . You are including commands for that in the command queue. Then you will have suitable commands for setting kernel arguments. Once the kernel arguments are set, you will set the global and local work sizes as usual in the OpenCL host program. And then you clEnqueueNDrange on one of the kernels so let us say it is the first kernel in the queue, right with respect to global and local work size.

And then you perform a blocking read back to synchronize. So essentially, I am just trying to say that after this command, you have a `clEnqueueND`, a read buffer command for the same queue, where you are going to read back the outputs of the kernel right. And while having that read buffer command, you said the blocking read property to true so that unless this command executes, the host program will not move forward to the next command.

So here you are achieving synchronization using the blocking grid. The blocking grid is ensuring that you do not go beyond to the next kernels read buffer commands. Unless this happens, right. So that you again you may have read buffers performed here `clEnqueue read buffer` like this and then again a `clEnqueue read buffer`. So, these are the blocking reads who are performing right once this kernel has executed.

And after that you will see you have the `clFinish` command. Now, the `clFinish` command will again ensure that inside this queue whatever commands have been include have all finished so far of course, the blocking read will these blocking reads if they are running routes then anywhere you cannot go beyond each of them. And this additionally says that will I you can go beyond this point only when all these commands are finished.

So, this is the way you will ideally write the program in case you have a simple in order code. So, just to summarize like you have a normal in order execution of kernel, the application can assume that kernel if you finishes first and then kernel B is executing just as we have discussed and the memory objects which are output by kernel A are inputs to be and then we will see the current data and objects produced by the execution of A, I mean, so, if I want a normal in order execution.

So, what will I do is after A has finishes? After A has finished, let us say, I am having a blocking read and then if required, I will again to a blocking write so, in that way I can synchronize to the host program or here I am performing a blocking read back to synchronize from the buffer outs.

(Refer Slide Time: 09:43)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Out-of-order Execution of Kernels". Below the title is a bulleted list of points about command execution. On the right side of the slide, there is a video feed of a man speaking, identified as Soumyajit Dey. At the bottom left is the text "OpenCL™ Runtime System" and at the bottom right is "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

- ▶ If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is set when the command-queue is created, the commands enqueued to a command-queue execute in out-of-order.
- ▶ In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued.
- ▶ It is possible that an earlier clEnqueueNDRangeKernel call to execute kernel A identified by event A may execute and/or finish later than a clEnqueueNDRangeKernel call to execute kernel B which was called by the application at a later point in time.

And then, let us come to the other option, which would be this out of order execution of kernels. So, for out of order execution of kernels, you will have this CL QUEUE OUT OF ORDER EXEC MODE ENABLE property of the command queue is set at the write at the time when you were creating the command queue right. So, in that time, the commands included in the command queue may execute out of order. So, out of order as we have discussed earlier that you do not have any guarantee that the few commands will finish execution in the order they are Enqueue, right.

I mean, so, they will start executing outside out of the order it is possible that an earlier NDrange kernel call to execute kernel A identified by some event A may execute and are finished. So later then a clEnqueueNDrangekernel to execute kernel some kernel B which is called by an application at a later point of time. So it may happen that you I mean whatever commands you have execute include there.

They are executing outside that are normal order, enqueued to a command queue execute. So maybe you have enqueue you have executed this two commands clEnqueueNDrangekernel for a kernel A and against clenqueueNDrangekernel for a kernel B but the kernel B gets actually launched to the device before kernel A has been launched or kernel has been launched, but it is still not finished just to go back why this issue was not there in order execution for primarily for this blocking read.

So, suppose you have a code where you have copied you have written 2 buffers again to the device memory and you have executed a kernel which would have worked on the buffer A and B and then you are reading the buffers performing or blocking read back to synchronize. So, if you have done it in such a way so, this is a blocking read as you can see, and then so that is actually ensuring that the kernel A finishes.

And after that, if you launch another kernel, it is right. But so if you do not if you have an out of order property enable, then you have to actually keep in mind that will the kernels may get launched in a different order, although you have include them in order, but they can technically get lost because again, I will just try to remember remind you this program is what it is doing is just saying.

What is the order in which I am including the kernel here 1 kernel is in queue, maybe at some point I am going to enter into another command. The host program is executing commands in this sequence right. So this is just a sequence in which the host program is executing inside the command queue, all these right buffers kernel again, the read these are all common that are getting Enqueue in the command queue.

So this is the execution order of the host program, right is execution order of the host program. But when I have in order execution from the command queue, then these commands are submitted to the device exactly in this order. Only in that case, I have execution in order. Otherwise, if I have an out of order setting in the command queue, then just by in queueing them in order to the host program I do not have a guarantee.

Because still in that case, you see, maybe I am performing a blocking read, but does not matter. This indeed inch content has been Enqueued here, right? It does not mean it is being issued to the device or it has started execution it will finish and then it will start right here by performing the blocking read or by the CL finish all that we are doing is that we are delaying the incoming of the second kernel in the command queue right because we have a guarantee that we have the execution will be in this order right.

But if the command queue is set in an out of order and if I am not having this sequence of commands to be executed, I may not have such a guarantee. So, we will see that how what happens in that case.

(Refer Slide Time: 14:29)

Out-of-order Execution of Kernels

To guarantee a specific order of execution of kernels, following can be done-

- ▶ Wait on an event A by specifying in the event_wait_list argument to `clEnqueueNDRangeKernel` for kernel B.
- ▶ Marker (`clEnqueueMarkerWithWaitList`) or barrier (`clEnqueueBarrierWithWaitList`) command can be enqueued to the command-queue. This ensures that previously enqueued commands identified by the list of events to wait for (or all previous commands) have finished.
- ▶ `clSetEventCallback` to call registered callback function when the execution status of command associated with an event changes to given status.

OpenCL™ Training Session

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, to guarantee a specific order of execution of kernels, this falling can be done. So, you may wait on an event A by specifying the event in the event list argument to the `clEnqueueNDRangeKernel` for kernel B. So, for every kernel, I can have an event where at least that only when those events as you must specific value, then the kernel will execute. So, I can have all the other options would be that I put in a marker or a barrier using the `clEnqueueMarkerWithWaitlist` or `clEnqueueBarrierWithWaitlist`.

So, this as we have seen that I can enqueue barrier on marker commands in the command queue. If I put in such markers then they will answer any ensure synchronization inside the command queue and prevent some out of order executions scenarios which I would want to avoid those which I want to happen to enhance performance because I can keep those that I want to avoid that I can enforce a suitably positioning the markers are the barriers in a command to us.

This ensures that previous to include commands identified by the list of events to wait for have finished or all previous commands I would say. Now the other event, either way will be as we have discussed that I can have a callback function registered and I can have CL set even callback so that

when the execution status of a command associated with an event changes to a given status, then this function would actually let the system know and I can make some progress forward right.

So, whatever synchronization primitives we have discussed earlier, they may also be used to actually provide sequencing inside command queue. In case the command queue is already set for out of order execution. So, again I will just repeat if you have in order execution property for the command queue that is fine. So, commands will be getting executed in the sequence. And so, I do not have a problem, because you have writes followed by 1 kernel executing then you can have a blocking read.

So, unless that read is finished, your host program does not go to the next including operation right. So, and there is also important we will need to say that suppose I was not having a blocking retail then what happens? Well, the host program is following this sequence. It will include the read buffer command, then it will also include the letter say you are also including the next corner for executing.

So again, the host program is just Enqueuing right? So it is not waiting for the read command to complete before it will possibly include the next kernel after at any of these points right. Now, due to that, even before all the reads happening to an input buffer for the next Kernel, the kernel may get launched that is why to ever that you will like to have the read set to a blocking read property fine.

So, in that way, in this code, as you can see, we will have this blocking read and then the blocking read would be followed by another `clEnqueueNDRange` kernel command for the next kernel right. So, this is about the in order execution. And then just to summarize, in out of order execution, you have a lot of problems, because unlike the in order execution, where the commands you are Enqueueing in the order in which the host program is executing.

And the actual execution and actual launch of commands are going to follow that exact sequence that gets broken, right because in the out of order execution, although your host program will move like this, and enqueue commands in a specific order, the commands may actually get start

executing in a different order because it is an out of order execution. So, this is something again, I will keep on repeating I am in enqueueing commands in order here.

Since I am enqueueing commands in the order here, and the command queue is set to in order execution, the commands will also be launched in order. But here also you have to be very careful. Again, I am repeating I know, but is important. I have to be careful because I am launching the commands. I am including them in order. So that will ensure the commands get launched in that order.

But still, it does not ensure that the previous command finishes before the second command executes because they are getting launched in the order right. For example, as I said, this read followed by the next kernel launch, there is a reason why I make the reader blocking read so that the data is available for the kernel launched even in case of in order execution. But as you can understand the problem is far more pronounced in case of out of order execution.

Because the host problem is in feeling in an order. But even though you are enqueue it an order your commands may not be submitted for execution in that specific order. And so, even in order execution we had that issue right that you may yes, you may be submitting them for execution in a specific order, but you are not waiting for previously issued commands to finish. Here the issue is even more I mean grave I would say, because first of all, you are not even submitting commands for execution in the host specified order.

And again, so, you are submitting them in some order to the runtime system wants, apart from the order in which the hostess include. Moreover, the order in which they start execution may not be same as the order in which they finished execution. So, that would mean this kind of a bad scenario in a worst case I would say. Let us say the host has enqueued commands in an order. With in order scenario I had already have a problem. Let us say this is first execution, but, if the second issued command has a dependency on the first command, for example, or read event has to be completed before a kernel has to execute, then I have to either put in a blocking call or I have to put in a barrier or I have to put in some other synchronization prior to all that. So, that is already the

problem in order. Now, you have the additional problem that actually, although you may have the first command, followed by the second command.

When they are getting issued, maybe the second command has been issued, followed by the first command has been issued. And that is actually followed by the third command that is issued. And maybe the second command has a dependency on the first command. So the second has been issued, first has been issued third has been issued. This time and the second have a dependency on the fast that is also not specified.

So, you may have multiple issues right because the second having a dependency on first was already the problem and in order for that you have to choose to use blocking and all that here, you are further complicating the problem with this out of order execution, because you have to somehow make the system know that 2 has to depend on 1 and all that. So, how to ensure this as we have discussed that we will be using we will be making use of event wait lists, we will be making use of markers and barriers, and we will be making use of callback functions.

(Refer Slide Time: 22:46)

Out-of-Order Execution Example Code

```
// Perform setup of platform, context, queue and create buffers
...
cl_event writeEventA;
cl_event writeEventB;
cl_event kernelEvent;
cl_event readEvent;
clEnqueueWriteBuffer(queue,bufferA,CL_TRUE,0,10*sizeof(int),a,0,NULL,&
                     writeEventA);
clEnqueueWriteBuffer(queue,bufferB,CL_TRUE,0,10*sizeof(int),b,0,NULL,&
                     writeEventB);
// Set kernel arguments
...
size_t localws[1] = {2}; size_t globalws[1] = {10};
// Wait on both writes before executing the kernel
cl_event eventList[2];
eventList[0] = writeEventA;
eventList[1] = writeEventB;
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So let us see what are these? So pencil will first of all, let us consider that we have already set up the platform context Q and the buffer circulated as usual practice and you have some open CL events defined as write event A write event B and you have an event defined for kernel event, you have an event for event and all that. Now, we will start creating this normal enqueue commands

and we will be specifying some events to be attached to them, which we have we are not doing earlier.

So let us say we have a write buffer command. So, the So, the write buffer command in the in a defined command queue, you are writing the buffer you are including a command to write the buffer day into a device I mean in for the specific device to is the command gives attached blocking rights, you have the size and other specifications and you attach a command away an event right event A with this.

So that the status of this command can be found from the status of this write event A, similarly, you also attach the write event B for the other write command for buffer B. So, you have include 2 write commands in the buffer in the command queue they are writing in the device memory buffer A buffer B. The status of this writes can be known from the value that the runtime system will as you actually assigned to write event A and write event B.

After this you have usual definition of kernel arguments your usual definition of local and global work sizes. And now, let us say you create an event at least by packing these 2 events write event A and write event B into this .

(Refer Slide Time: 24:36)

Out-of-Order Execution Example Code

```
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, globalws, localws, 2, eventList, &kernelEvent);
// Decrease reference count on events
clReleaseEvent(writeEventA);
clReleaseEvent(writeEventB);
// Read will wait on kernel completion to run
clEnqueueReadBuffer(queue, bufferOut, CL_TRUE, 0, 10*sizeof(int), out, 1, &kernelEvent, &readEvent);
clReleaseEvent(kernelEvent);
// Block until the read has completed
// cl_int clWaitForEvents(cl_uint num_events, const cl_event *event_list)
clWaitForEvents(1, &readEvent);
clReleaseEvent(readEvent);
```

Source : NVIDIA System

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, why would we want to do that you want the kernel enqueueing to happen in depending on whether this two write events have been have actually completed. So, what you do is now you

enqueue the kernel for execution in the command queue providers parameter settings global and local work sizes and you also give it a eventless to wait on and you attach with this command the kernel event.

So if I try to draw a graphical picture here, what is happening this is the queue so you have executed you have enqueued a write buff command. You have another write buff command. And you have enqueueuer the kernel with this write buffer command. So these are 2. These are the 2 write buffer commands and talking about this I has been enqueue right now with this 2, we have attached to OpenCL events. And those 2 events now start appearing into the event list of this kernel. What are those OpenCL events write event A and write event B.

So, these are OpenCL elements write event A and similarly B and they are here in the event list A, B. So, now you do not have the problem because the event leads to something like only when these events are finished they reach their CL finish only then this command which has been enqueued can execute. So, now, even if the command queue start I mean the runtime system try to optimize execution of the command queue. Even if it tries to initiate dispatch of this kernel before this it cannot, because now you have explicitly specified the dependency of this write event A and write event B.

This should be write event A and this is B here explicitly specified the dependency of write event A and write event B with the execution of the kernel using this signals are the OpenCL events write event A and write event B. So, now, when these 2 are done there they actually force that even this cannot execute out of order, it can only execute after write events A and B are done and they reach the status of clfinish.

Now, when they are done, you see this `clEnqueueNDRangeKernel` command when you are enqueueing this in the queue. In the last argument, you have the kernel event. So that is another the last flag that would mean that with this command itself, you are attaching another event called kernel event. So that means with write event A and write event B finishing, they can start in new order right I have not given any order it can is an out of order queue that these 2 can start in any order.

When they finish only then the kernel execution can start which is being forced by this event list here, when the kernel executions ends, then you have also associated this kernel event type OpenCL event kernel event with the queuing of this ND range kernel the queuing of this kernel. So, now, only when the kernel execution finishes, then this kernel event will get this finished status and this can be used for upstream further execution.

Which are dependent on the kernels execution. So, for example, after this point, you do not have any importance of write event A and write event B so you can just release those events right. So you do a CL release event. That is an OpenCL function. Now you want the second kernel to execute. So, when do you want to read the output that has been computed by this kernel.

So, for that you will enqueue a read operation. So, now you will enqueue read operation. So, you have a read operation. Now, this read operation is waiting for this kernel event and this read operation will output the result. So, that is the associated so, maybe you will say that it is waiting for the kernel event and it is associated with this read event. So, when this read operation is included, it is restricted from starting earlier by this dependents on the kernel event right this is a dependency on the kernel event only when this is done then read operation can start when read operation means there associated read event will be true. At this point you do not have any requirement of the kernel event so you can release that event. And you can actually use this to defend to have some further dependent executions going on here, right.

If you want them you can add few more commands with the dependency of this. So, here if you want to block until the read has completed, your option would be you use the function `clWaitForEvents`, unless you are you do not want to give another command which will depend on the dead. You just wait for this event using the CL wait for events. So you just wait here. So the option we choose here is we are not enqueueing any more command so you are just waiting.

And then when it is done, then this function is a blocking the execution here, it will let the execution progress and then you release everything. So here we are showing that how using these events and dependency lists of every enqueue ND range command. So this enqueue commands earlier, we

have been just enqueueing commands into the queue. But now you are showing how enqueue can be done with dependency lists.

The last 2 arguments of every enqueue command used for this event dependency list. The last argument is for the output dependency that well this is done this event will now go on. So this is actually specifying the event that you are attaching with the command that we have been enqueued. And these events status will keep on changing with the different execution status of this command.

And before this event, the last but one argument is the event list. This variable event may contain a set of events and when all those events actually get the clFinish. Then only this command can start executing so this is how you can mark out dependencies and force an ordering.

(Refer Slide Time: 32:01)

Out-of-Order Execution Example Code

Another approach

```
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, globalws, localws, 2, eventList, &kernelEvent);

// Read will wait on kernel completion to run
clEnqueueReadBuffer(queue, bufferOut, CL_TRUE, 0, 10*sizeof(int), out, 1, &kernelEvent, &readEvent);

// Set the callback such that callbackFunction is called when readEvent
// indicates that the event has completed (CL_COMPLETE)
// cl_int clSetEventCallback(cl_event event, cl_int command_exec_callback_type,
// void (CL_CALLBACK *pfn_event_notify) (cl_event event, cl_int
// event_command_exec_status, void *user_data), void *user_data)
errcode=clSetEventCallback(readEvent, CL_COMPLETE, callbackFunction, (void *)ipargs);
clReleaseEvent(readEvent);
```

Source: [Source](#)

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

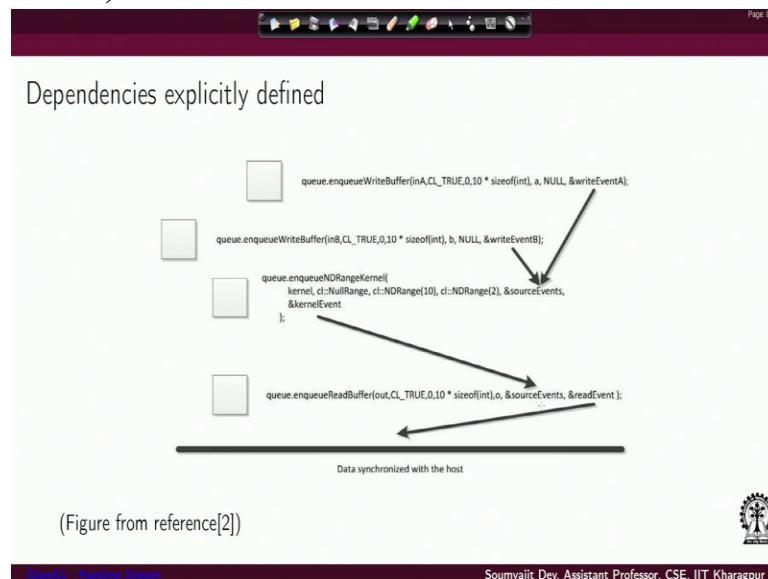
Now, there could have been another approach, which is that you suppose you have this `clEnqueueNDRangeKernel`. So you enqueue the kernel, you have the event list, and you have the kernel event. So we are still speaking from that point. Now, the read will wait on the kernel competition to run. After that you use this kernel event to include the 2 as a dependency to include the read event. When the read event is done, you get the redefined call. So the read event status will turn `CL_COMPLETE`. Here also you had the read event up to this is fine. After that you were doing a wait for events instead of wait for events.

Alternatively, what you could have done is that you could have used the callback function. So this is our approach we thought, right? So what the callback function will do is it is indicating. So that callback function will be registered with this event using this CL set event callback primitive. So all you do is with this event CL read event with this event read event you are associating a function called callback function.

So the way to read it will be when this read event will get the status CL complete, then this CL function this callback function will execute asynchronously, with some arguments. So, the definition of the function prototype `clSetEventCallback` is that it is the event to whom you can look that whether this event has attained these execution status. When it has when some event like in this case, the read event has attained an execution status, in this case CL complete.

Then some callback function like this of type CL callback. In this case, we are just calling the callback function name as callback function will execute with some argument whose which can be specified here right now, so in that way I can have another callback function to execute. And finally, I will just release the read event that we have defined earlier right. So, that can be another way to synchronize things in out of order execution.

(Refer Slide Time: 34:23)



So, these are pictures we have taken from this reference, which we have let us reference is the openCL book on OpenCL and OpenCL 2 points where I would say so. As you can see that we have this write buffer commands and these are the associated events that we have defined. The

`clEnqueueNDRangeKernel` is enqueueing this kernel and it can only execute when these events are done.

And it will output the kernel event to CL complete and using this kernel event. I can initiate the execution of read buffer and when the buffer is done it will get as output the redefined and in that way of the data synchronized with the host. So, this is the other option of dependency.

(Refer Slide Time: 35:12)

Profiling Operations on Memory Objects and Kernels

- ▶ Profiling of OpenCL commands can be enabled by using a command-queue created with `CL_QUEUE_PROFILING_ENABLE` flag set in properties argument to `clCreateCommandQueueWithProperties`.
- ▶ When profiling is enabled, the event objects that are created from enqueueing a command store a timestamp for each of their state transitions.
- ▶ The OpenCL APIs used for profiling is `clGetEventProfilingInfo`
- ▶ Given by 64-bit value that describes the current device time counter in nanoseconds when the command identified by event is done

OpenCL Profiling System

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Apart from this, we have an option of profiling operations on memory objects and kernels in OpenCL like so, the way it works is you can profile OpenCL I mean, in case you want to profile the execution of OpenCL program, you have to set a command queue with CL queue profiling enable flag. Now, once that is done, and then the event objects that are created in the queue from enqueueing of a command, they will I mean, they can also store as timestamp for each of their state transitions.

So, just to remember, with every command that you enqueue, you can have an associated event object for that event object. You can have as we know that the runtime will keep on changing the status of the event object right. Now, as we have discussed earlier that every enqueue command will return an event right. Now, if you actually give a name in that field of enqueueing of CL enqueue commands, then for a specific event.

Then that return event will be in the name in that event, right. And so, you can use that event name to sample the different possible status like CL_COMPLETE, CL_SUBMITTED etc. like that to know what is the status of that command. Now with profiling the system will also I can also give you a timestamp when the status changes from something to something.

So for this you can use the OpenCL API clGetEventProfilingInfo. So given by a 64 bit value that describes the current device to encounter in a nanosecond resolution that you can get to know when some event can attain a specific status.

(Refer Slide Time: 37:09)

Profiling Operations Example

```
...
clCreateCommandQueue(ctx, dev_id, CL_QUEUE_PROFILING_ENABLE, &status)
...
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, globalws, localws, 2, eventList, &
execEvent);
...
//Get profile info related to the event execEvent
//cl_int clGetEventProfilingInfo(cl_event event, cl_profiling_info
param_name, size_t param_value_size, void *param_value, size_t *
param_value_size_ret)
//Give the current device time counter in nanoseconds when the command
identified by event is enqueued, submitted in a command-queue by the
host, starts and finish execution on the device
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Let us take an example suppose you have created a command queue and it is in profiling enabled mode. Now, you are enqueueing a kernel in that queue and that kernel has event list and you are attaching an execEvent and with that kernel now, you So essentially, this will help you I mean, the I mean, since you have the queue with profiling and even more, so, the runtime system will ensure that for this execEvent, during the lifetime of this event, whenever it is switching these different modes, the corresponding timestamps are collected by the runtime system.

And you can use profiling information of API to collect those values. So you can use this CL get even profiling info function to get those values. And so, what we can do is give the current time count device time comes in nanoseconds so that is what it does, and it will sample it out when the command identify the event is included, and submitted in command 2 by the host and starts and

finishes execution on the device. So, these are the different states through which the command will go through. And although state whenever those states are reached those values can be collected.

(Refer Slide Time: 38:17)

```
//current device time counter in nanoseconds when the command identified  
// by the event is in the specified state.  
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_QUEUED,  
    sizeof(cl_ulong), &queued, NULL);  
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_SUBMIT,  
    sizeof(cl_ulong), &submit, NULL);  
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_START,  
    sizeof(cl_ulong), &start, NULL);  
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_END,  
    sizeof(cl_ulong), &end, NULL);  
...  
printf("clEnqueueNDRangeKernel profiling details:\nQueued: %llu, Submit: %  
    llu, Start: %llu,Finish: %llu ",queued,submit,start,end);  
...
```

For example, you can use this `clGetEventProfilingInfo` API for this exact event and see that, this is the way you have to do it. So, for this you have to write it like `execEvent` and is the event and CL profiling command queue. So, you so essentially, you should return what is the exact time when this exact event reaches the status that it has got actually queued? What is the exact time when it has been submitted for execution?

What is the exact time when it starts execution and what is the time when the command is sent? So these are the different statuses that can be example and when the status is an example, they would actually get returned in this. So, this is basically the time so CL profiling command and so, you have queuing time submitting time submission of for execution to the device, start time for execution, actual execution and the intent for execution.

And all these values get stored in this corresponding addresses for queue submit start and end as we have been defining here. So it is this field, right? It is this field parameter value field in which those values get stored. And finally, if you want to print them, you can just use and use this as a normal printer function and print the exact timings for that specific kernel getting queued. The kernel getting submitted. The kernel is starting and the continuity is going to finish because all of them have got stored in these variables.

(Refer Slide Time: 40:06)

References

- ▶ Khronos OpenCL Working Group. *The OpenCL Specification Version-2.1*. 2018 February 13,
- ▶ Kaeli DR, Mistry P, Schaa D, Zhang DP. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann; 2015 Jun 18.

Soumyajit Das, Assistant Professor, CSE, IIT-B 2025

So these are some important references we did like to you may want to visit for knowing in more detail about OpenCL. So, we have made exhaustive use of these two resources. The first one is the openCL specification made by Khronos, OpenCL Working Group Khronos is freely available in the internet. The second one is a very nice book on heterogeneous computing with OpenCL by these authors as written here. So I hope this was a nice journey to the basics of OpenCL. Thank you for your attention.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 48
OpenCL – Heterogeneous Computing

Hi, welcome back to the lecture series on GPU architectures and programming. So I believe in the last week lecture, we have been talking a lot about OpenCL, as programming model and language, which is kind of runs parallel in compute capability of CUDA with some differences in terms of application platforms. So, this week, we will just continue a bit on using OpenCL for a heterogeneous computing environment.

Although I would say that, if , although we will be talking about openCL and heterogeneous compile computing, but realize that CUDA is also the compute language which is used more often than not. We will also talk about how CUDA programs can be written in such a way that you can have multiple GPU devices on which the program runs given that some host CPU is orchestrating these different devices. So, starting with that motivation.

(Refer Slide Time: 01:23)

The slide has a dark blue header bar. Below it, the word 'Recap' is centered in a white font. The main content area is white with a thin gray border. At the top right of this area, there is a small circular icon containing a question mark symbol. In the center of the slide, there is a bulleted list of three items:

- ▶ Introduction to OpenCL
- ▶ OpenCL runtime system
- ▶ Synchronization in OpenCL

At the bottom of the slide, there is a dark footer bar. On the left side of the footer, the text 'OpenCL - Heterogeneous Computing' is visible. On the right side, the text 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' is visible, along with the IIT Kharagpur logo.

So, just want to say that in this case, if you want to recap, you can just look into the lectures we did earlier on introduction to OpenCL the details of the runtime system and the different synchronization primitives in OpenCL.

(Refer Slide Time: 01:40)

Heterogeneous Computing

- ▶ Computing platforms with more than one kind of devices (processor or cores) are called heterogeneous platform
- ▶ Heterogeneous Computing utilise this heterogeneity to gain performance or energy efficiency
- ▶ Concurrency is where applications execute functions concurrently on multiple processors
- ▶ OpenCL is an ideal programming language for heterogeneous computing implementation as it support programming across multiple computing devices, such as CPU, GPU, and FPGA from different vendors

OpenCL - Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So coming to heterogeneous computing, what fundamentally is a trillion computing? So, essentially, we are talking about compute platforms which have got more than one kind of devices, it may mean processors are at a light level, we can talk about multiple different course, because I can have a compute platform with multiple different courts, and each court is kind of optimized to do different kinds of processing.

I can also think of a compute platform where on the PCI bus, I have different kinds of devices which are attached and they together are being also attached with a CPU device in the PC attached in the PCI bus and which is kind of orchestrating the execution of different kernels in these in this set of different devices. So, the idea of what radius computing is that how to make good use of this heterogeneity in terms of working platforms to gain in performance as well as energy efficiency.

So, if I have to do that, then I need to have multiple application kernels which are executing concurrently in these different devices. So for that, OpenCL is conceived as an idea programming language do because for because it supports programming across multi compute devices like CPUs, GPUs, as well as SPGs provided from different vendors of course, the vendor needs to support OpenCL, ICT loaders, essentially that the implementation of the origin urban specification that has to come from the vendors.

And there are a lot of vendors in recent times we have subscribed to this ideas. Just to let you know, in my in modern days, there has been an evolution in the OpenCL community and there has been talked about a new programming paradigm called Vulkan, which is kind of an OpenCL derivative popularly used to kind of develop data parallel applications in the context of android like mobile devices and mobile devices, we subscribe to android like operating systems.

(Refer Slide Time: 03:45)

Heterogeneous Computing

- ▶ Available processors in the system should be efficiently used in heterogeneous computing
- ▶ Challenge is to identify preferred task to device mapping, minimize overhead due to data transfer, synchronization etc. in such heterogeneous system
- ▶ To take full advantage of this heterogeneity to gain performance or energy efficiency, programmer need to handle these challenges efficiently

Soumyajit Dey, Assistant Professor, CSE, IIT

So coming back to our ideas on heterogeneous computing, here we are talking about processors which are available in the system and we multiple of them are available and we want to use them efficiently. But of course, this brings in a lot of challenges. So if I want to characterize a problem here, let us consider the classical architecture to application mapping problem. So we have an application, which may comprise multiple data parallel kernels.

So if I am just trying to draw semantics of what we are trying to mean here, so let us say I have a set of data parallel kernels, which they are dependencies. So this is K_1, K_2, K_3 and like that K_4 this is the dependences. And let us say so this is my application. Let us say I have a target platform, we are on the bus, I have multiple processors, or let us open in terms of OpenCL, OpenCL devices, which are connected.

And let us say so they can be of different types. This can be a CPU or GPU type device, let us say, GPU device. And let us use a CPU device. And it is running the host program. And so, the objective

is suppose we have this kind of a platform, we have got this kind of an application, how to map this application to this kind of a heterogeneous compute platform. So I call it a heterogeneous compute platform.

Because of this heterogeneity in terms of different devices that we have there can be much more variety. For example, if there can be let us say a device before, which is some neural computing in some other kind of accelerator card which is there which also understands how to execute an OpenCL kernel. So, that is the important thing we are subscribing to the idea. There are these all different SMD compute platforms, they can be CPUs, GPUs, some other accelerator cards.

The constraint these all of them have onboard an implementation of the open source software specification. So that if this host program can discover this device, create a context for it or create a context with multiple instances of such devices and dispatchers kernels towards them, they will get executed. So, that would be what we call the classical application to architecture mapping problem restricted in the context of heterogeneous compute.

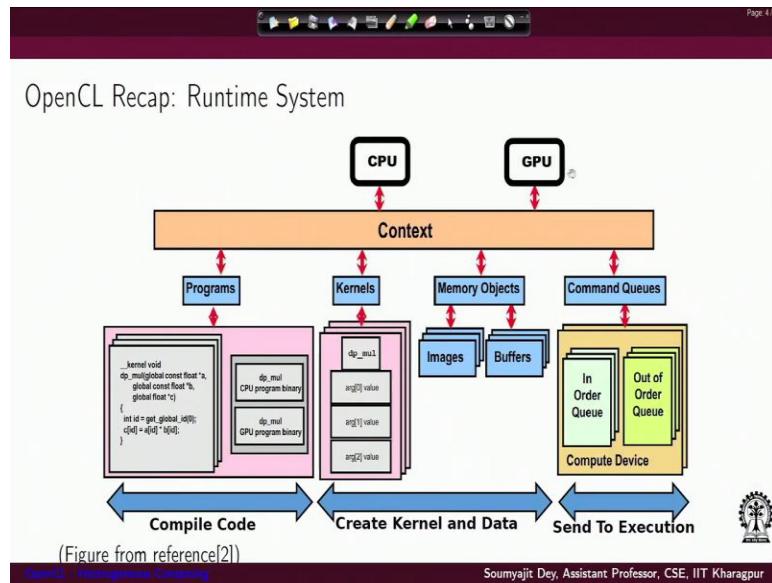
So, the challenge in this case is to identify preferred tasks to devise mapping, maximize, and minimize the overhead due to data transfer and synchronization in such a system. So that is also very important. When suppose, we consider again, let me just draw the first figure that if you have this kind of data panel kernels and you are trying to decide whether to map them in a single device or across multiple devices.

And the issue that may come suppose you decide on a mapping decision that you map two of these kernels in one device and the other in another device then you have a data transfer over it from this kernel to the rest of the kernels. But otherwise or also the execution here has to wait for the execution of this kernel to complete. So, you have a synchronization over it, you have additive transfer over it and based on how you do the mapping, you have an overall throughput.

So, depending on how you which aspect you want to optimize, you have to choose what kind of tasks will be phase mapped you want to have so, in order to take full advantage of this

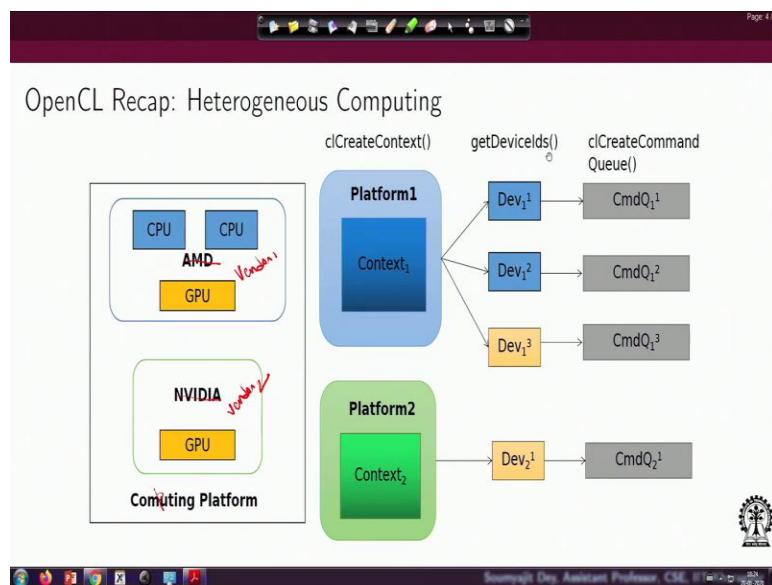
heterogeneity, and to gain performance in terms of performance or energy efficiency, the programmer will need to challenge handle these challenges very efficiently.

(Refer Slide Time: 08:20)



So, this is an overview of the OpenCL runtime system, which we discussed earlier from our last lecture. So, and that was that, , that was actually trying to give you an overview of the different aspects that are really there, like you have to compile the code, you have to create kernels, and then you have to orchestrate the execution across devices and handle the transfer across devices. That is the overall job of this underlying runtime system.

(Refer Slide Time: 08:48)



Now, coming to this idea of heterogeneous compute, so, sorry for this, it will be a computing platform here and coming to this idea of heterogeneous compute, we are trying to give a very high level view here. So, consider that you have a platform with multiple vendor provided compute elements. Of course, they are connected and through the some share some shared memory or through based communication, they are communicate connected.

And you are using OpenCL to orchestrate the execution here. So, one possibility would be, for example, you can you are building we are just trying to give an example of how to build context like we have discussed earlier. So maybe I choose to create context for this platform from the vendors of AMD. We are just again taking examples here. And then we create another platform from the vendors of NVIDIA.

So, again, I will just repeat this our hypothetical is a very hypothetical example, we are thinking that such a computing platform is available is a totally theoretical thing. It does not actually mean of course, this kind of platform will never exist. So maybe we can just think that will we have things from different vendors. And we are creating, an OpenCL abstraction of these compute platform. Maybe we can create one context for platform 1, another context for a platform 2, we go on to discover devices in each of the platforms.

So we will discover 3 devices here maybe we will discover one device here and P also go on to create command tools for each of them is like a very high level view of how things will look like we are trying to say that the because there are different kinds of compute elements which are present, the abstraction, the definition of OpenCLs support, things coming from different vendors, though, again, I will say that I do not think there are any example devices available where you have things from different vendors from the same. These are very hypothetical here.

So, it could have been just a vendor 1 and vendor 2. So, maybe we can say this is again from some vendor 1 and this is from some vendor 2 and you have created their corresponding devices. So, assuming that you have such a platform available, this is how you go about creating context discovering devices to the separate this commands that we have discussed and all that so on so forth.

(Refer Slide Time: 11:41)

The screenshot shows a presentation slide with a dark purple header bar containing icons for file operations like Open, Save, Print, etc., and a page number 'Page 5/5'. The main title 'Command Queue and Device' is at the top left. Below it is a bulleted list of points about command queues and devices. At the bottom right is the IIT Kharagpur logo, and the footer bar includes 'OpenCL Heterogeneous Computing' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

- ▶ A command-queue can be associated with only one device
- ▶ Single device may be associated with single command-queue (already discussed)
- ▶ Single device may be associated with multiple command-queues with same context
- ▶ Single device may also be associated with multiple command queues associated with different contexts within the same platform
- ▶ Different devices may also be associated with multiple command queues associated with the same context
- ▶ Different devices may also be associated with multiple command queues associated with different contexts

So, we know that there is this idea of command queues, which can be created in a pod device manner, every device should have its own command queue. So, once those things are set up, there are a few restrictions we need to get examine. For example, A command queue can be associated with only one device. A single device may be associated with a single command queue, which we use a case we have already discussed.

However, it may be associated with multiple command queues also in the same context. So that can be another possibility. A single device may also be associated with multiple command queues, which are associated with different contexts within the same platform. Because if you remember we discussed earlier that even inside a platform I can create different contexts. So, maybe we can just go about and create one context.

Using this set of devices, we can create another context again these set of devices, so, that would be two context here. So, in that case, I can have two different command queues, elevated with different contexts. Both of them are trying to access one device which in sitting in one of the context, again, these are all theoretical possibilities that opens your route support. For example, again, I will just repeat that you may not have this kind of a platform, but this is the abstraction that is supported.

And it also depends on whether the vendors themselves have their own even sell implementations. So, you can also have this issue that there are different devices associated with multiple command tools, which I will say that again with the same context. And different devices, which may also be associated with multiple command queues. And those commands queues stated with different contexts.

So, there is just understood the differences, you have command queues, you have devices and you have context. Every device is a command queue, every command queue should subscribe to some device. We are trying to say what are the different options here, I can have multiple commands queue for a single device, I can have multiple command queue bringing to coming from different contexts mapping to some device.

They are giving conditioning commands to some device in contains one context, I can also have in the same context multiple command queues. And I can have these multiple command queues with different devices. And I can also have the same thing from the point of view of different contexts.

(Refer Slide Time: 14:28)

Multiple Command Queues With Same Context In Single Device

- ▶ Multiple command-queues can be mapped to the same device to overlap execution of different commands or overlap commands and host-device communication
- ▶ They execute commands independently
- ▶ They allows applications to queue multiple independent commands without requiring synchronization as long as these objects are not being shared

OpenCL Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

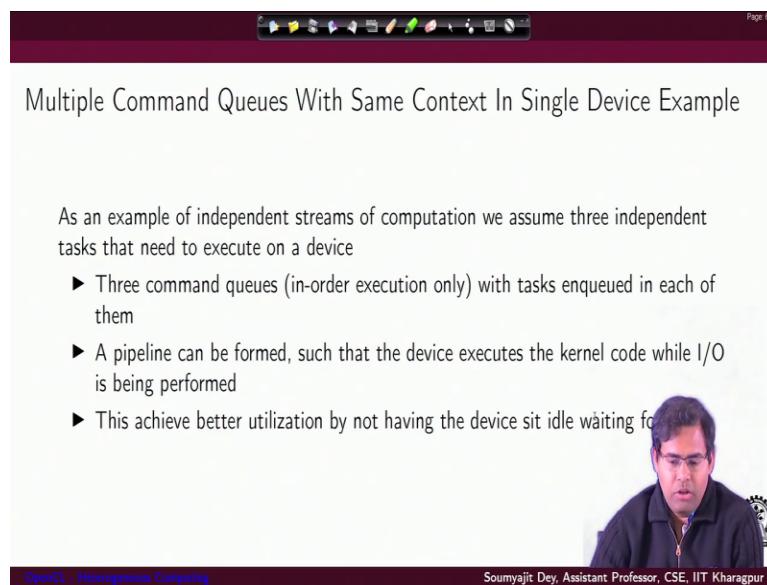
So let us take these cases one by one. So, considered that I have multiple command queues with same context in a single device. So essentially, we are saying that multiple command queues can be mapped to the same device. But why? The fundamental reason would be that where the device

is getting commands which are issued from the command queues, now, inside recently, the command queues so you are issuing commands in order. If you want to overlap the execution from the different common tools, then you may have multiple common tools mapped to the same device.

So that let us say one command queue is in queuing, from one command queue has a read or write operation and another command queue as a kernel launch operation and they are overlapping on the device that is a possibility. So, essentially what happens in this case, they are executing commands independently. Each of the command queues are throwing up their own commands to the device that those commands are executing independently among command.

There is an independence among commands coming from different command tools. They allow applications to queue multiple independent commands without recording synchronization as long as the objects are not shared. So there is also an advantage that if you have such multiple command queues, you are able to independent commands without requiring synchronization why because the moment you have commands in the queue, you have some ordering already being present.

(Refer Slide Time: 16:12)



Multiple Command Queues With Same Context In Single Device Example

As an example of independent streams of computation we assume three independent tasks that need to execute on a device

- ▶ Three command queues (in-order execution only) with tasks enqueued in each of them
- ▶ A pipeline can be formed, such that the device executes the kernel code while I/O is being performed
- ▶ This achieve better utilization by not having the device sit idle waiting for

OpenCL - Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

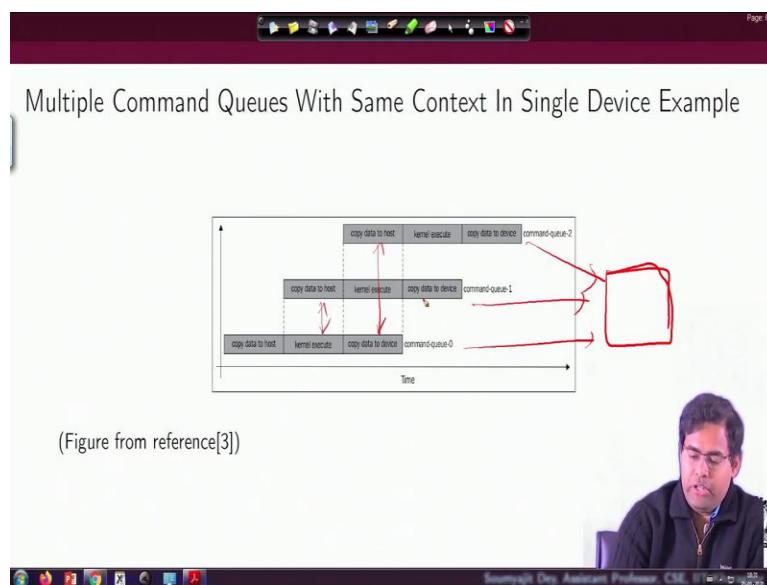
Now, consider this case you have multiple command queues with same context in single device as an example of that, you can have independent streams of computation and you assume that three independent tasks are there which need to execute on a device. So, we are saying that you have one device and I want a device available for executing 3 tasks concurrently in an independent way.

So, I will just set up 3 command queues in order execution with tasks include in each of them that would form a pipeline.

So, that each device executes the kernel code file and the I/O is transferred. So, let us say from one command queue I am issuing an I/O command like in read or write command there are only from the other command queue, I am issuing a kernel launch command, since the device will have separate hardware for read/write and execution using the streaming multiprocessors these operations can overlap they can be done concurrently.

So this will help in achieving that vision by not having the device sit idle and waiting for data, because while it is the copying doing the reader read or write operations, I have the device executing the kernel, some other kernel coming from one of the command queues.

(Refer Slide Time: 17:34)



(Figure from reference[3])

So if I have this kind of multiple command queues within the same context, and they are issuing commands to a single device, this is how it helps me to overlap executions is just like a pipelining with speed up our movement in computer architecture. So let us say a command queue 0 is providing this command sequence copy data to device kernel execute copyright to host and there is a simple sequence for another kernel. And those commands are coming from command queue 1.

Similarly, I have such a sequence from command queue two when I have this kind of execution and this execution sequence is going to the same device. They are going to the same device sitting in the same context, then I can see that this data and execution can happen in parallel copy that copy data to host copyright to device and execution can also happen in parallel provided you have got multiple copying engines, which are often available.

So one copying in takes of copying from the device of the host and other copying in this case of copying from the data from the host to the device. So in that way also I can have this kind of overlap execution. So that is the advantage in OpenCL of having multiple command queues with same context in a single device.

(Refer Slide Time: 18:54)

Multiple Command Queues With Different Contexts In Single Device

- ▶ Yes, we can create multiple contexts for the same device on the same application.
- ▶ Typically it has no benefit
- ▶ Useful when an application that uses OpenCL for certain operations also uses some third-party library that also happens to use OpenCL internally to accelerate some algorithms

OpenCL - Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So in the mean, we can create such multiple context for same device on the same application. And but what if I have multiple contexts? Is there a benefit? Typically? No, because it does not. So this is another case where I have such multiple command queues but the command queues are coming from different contexts. So the kind of over left execution advantage I get inside the context, that would not happen now? Typically, that is why there is no benefit.

It is useful when an application that uses OpenCL for certain operations will also use some third party library. That also happens to use OpenCL internally to accelerate some algorithms. So that so you have one context for the application. You can have, you can create a separate contest where

you load that third party library. We use it OpenCL internally for an algorithmic acceleration. So those are the cases where for providing support, you can land up in this kind of a setting.

(Refer Slide Time: 20:11)

Multiple Device Programming

Multiple device programming with OpenCL can be summarized with two execution models:

- ▶ Two or more devices work in a pipeline manner such that one device waits on the results of another
- ▶ A model in which multiple devices work concurrently, independent of each other

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, coming to this device, the programming of multiple devices. So this is all about single device, multiple command queues, are sitting in the same context and different context. Now let us talk about multiple devices. So, for that, I will have two execution models, I may have two or more devices, which work in a pipeline manner, such that one device waits on the results of another alternative like and have a model where the multiple devices work concurrently.

They are independent of each other. So, there does not consider different devices, which are in the same context and we have multiple concurrent command queues for these different devices. So for multiple devices in a system, for example, a CPU and the GPU or multiple GPUs, each device will need each device will need its own command queue? So you have a context where you have multiple devices, and each device will need its own command queue.

A standard way to work with multiple devices on the same platform is creating a single context as memory objects are globally visible to all devices within the same context. And an event is only valid in a context where it is created. So just to remember that if you have multiple devices, you may want to create this kind of a single context. Because then the memory objects which you

create before actual kernel execution, they are global inside the context, they can be accessed by all the devices. And same thing will hold for events that you define.

(Refer Slide Time: 21:51)

Multiple Command Queues With Same Context In Different Devices

- ▶ Within same context, sharing of objects across multiple command-queues will require the application to perform appropriate synchronization
- ▶ Event objects visible to the host program can be used to define synchronization points between commands in multiple command queues
- ▶ If synchronization points are established , the programmer must assure that the command-queues progress concurrently and correctly establish existing dependencies

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, inside the same context, you can also have this sharing of objects that you define across multiple command queues, but it will record the application to perform appropriate synchronization. So that dependencies are preserved. Now, event objects that are visible to the host program, they can be used to define synchronization points between commands in multiple commanders. I like we have seen some synchronization examples earlier. But those were for a same command queue. We are just trying to say that since this synchronization events there the event lists in OpenCL.

They are global data structures inside the same context they are actually available across command queues. So I can have events in events that have been triggered in one common queue, they can actually be used to trigger events in another command queue. So, if the synchronization points are established, then the programmer must ensure that command queues progress concurrently and establish the current dependencies of course like we have seen earlier, that using the event model.

That event lists and triggering events using this programming model to establish existing dependencies is something that is on the hand of the programmer and the programmer has to ensure that it really happens nicely.

(Refer Slide Time: 23:14)

Multiple Command Queues Creation With Same Context In Different Devices Example

```
//One platform having one CPU device and one GPU device
cl_uint num_devices;
cl_device_id devices[2];
cl_context ctx;

//Obtain devices of both CPU and GPU types
err_code = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[0], &num_devices);
err_code = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[1], &num_devices);

//Create a context including two devices
ctx = clCreateContext(0, 2, devices, NULL, NULL, &err);
cl_command_queue queue_cpu, queue_gpu;

//Create queues to each device
queue_cpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_gpu = clCreateCommandQueue(context, devices[1], 0, &err);
```

OpenCL Interoperable Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, consider some examples now for example, multiple common queues which are created with same context in different devices. So, you have one platform having one CPU device and one GPU device. So you have these devices area so what do you do , by just giving an example of a similar code. First, the difference only will be creating multiple command queues for multiple devices. So first to discover the devices using this CL get device Ids.

So you discovered the CPU type device and the GPU type device. So and then, we are assuming that we already know that in there we have this one CPU device and one GPU device. Then you click context where you use both these two devices. So there is a single context that you create for two devices? And then inside this context so then inside this context, you will be creating cues for each of these devices. And that is done in the next two commands. So you have CL create command queue here for creating the command queues for the same context.

And then so, but one of them is for the GPU. So as you can see that for the same context, you have, the two devices use the GPU device. So for that you have queue GPU and similarly for queue CPU.

(Refer Slide Time: 24:52)

Page 77

Multiple Device Programming In Pipeline Manner Example

Multiple devices working in a cooperative manner on the same data such that the CPU queue will wait until the GPU kernel is finished.

(Figure from reference[2])

OpenCL Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then it will be possible using this programming model that for these two devices, you can have overlapped execution that you should be able to launch two kernels. So that one kernel is executing in the GPU device. The other kernel is executing in the CPU device, but they are sharing data through a memory buffer. So, it will help us to the point that the multiple devices will be working in a cooperative manner on the same data says that the CPU will wait until the GPU kernel is finished.

So as you can see from the dependency diagram, you have kernel 0 in the command queue. Kernel 0 goes from dispatch to kernel 0 starts running after retains. So the consider timing here in this axis. So you in the memory buffer, it is accessed and this is the data is written after the data is written. You may have the data being accessed by the device one where the kernel was waiting and it will start running.

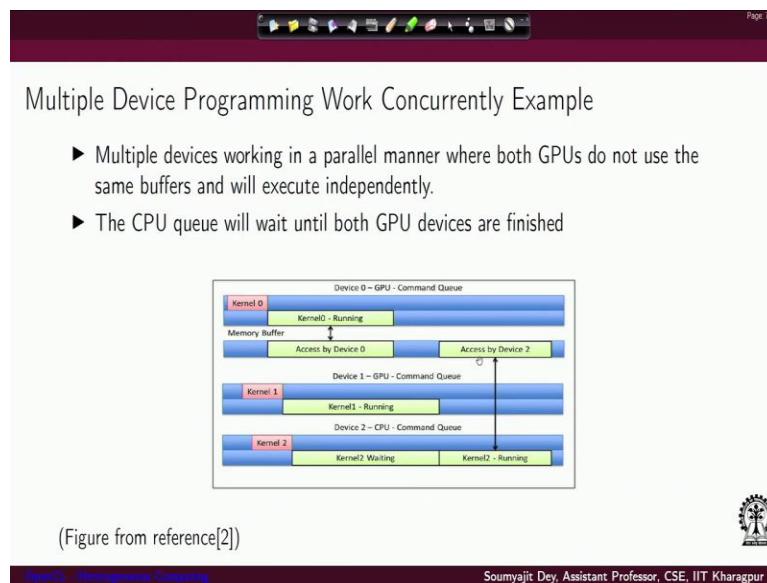
So we know how to write such quotes. And we will, go into looking into such programming models in a deeper way. And maybe we will end this lecture by stopping at this point of time. And that would be all thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 49
OpenCL – Heterogeneous Computing (Contd.)

Hi, welcome back to the lecture series on GPU architecture and programming. So, coming to the continuation of the earlier lecture, I believe we have been talking about multi device programming, where we have built context containing multiple devices, we have set of queue about multiple command queues each one queue for one of the device and we are trying to see how the devices can execute kernels concurrently.

(Refer Slide Time: 00:47)



Multiple Device Programming Work Concurrently Example

- ▶ Multiple devices working in a parallel manner where both GPUs do not use the same buffers and will execute independently.
- ▶ The CPU queue will wait until both GPU devices are finished

Device 0 ~ GPU - Command Queue

Device 1 ~ GPU - Command Queue

Device 2 ~ CPU - Command Queue

Kernel 0 Kernel0 - Running

Kernel 1 Kernel1 - Running

Kernel 2 Kernel2 Waiting Kernel2 - Running

Memory Buffer

Access by Device 0 Access by Device 2

(Figure from reference[2])

So, in that way, this was our example.

(Refer Slide Time: 00:50)

Page 7 / 7

Multiple Device Programming In Pipeline Manner Example

```

// A pipelined model of multidevice execution with single context
// The enqueued kernel on the GPU command queue waits for the kernel on the
// CPU command queue to finish executing
cl_event event0_cpu, event1_gpu;

// Starts as soon as enqueued
err = clEnqueueNDRangeKernel(queue_gpu,kernel1_gpu,2,NULL,global,local,0,NULL
    ,&event_gpu);

// Starts after event_gpu is on CL_COMPLETE
err = clEnqueueNDRangeKernel(queue_cpu,kernel2_cpu,2,NULL,global,local,1,&
    event_gpu,&event_cpu);

// clFlush only guarantees that all queued commands to command_queue will
// eventually be submitted to the appropriate device. There is no guarantee
// that they will be complete after clFlush returns
clFlush(queue_cpu);
clFlush(queue_gpu);

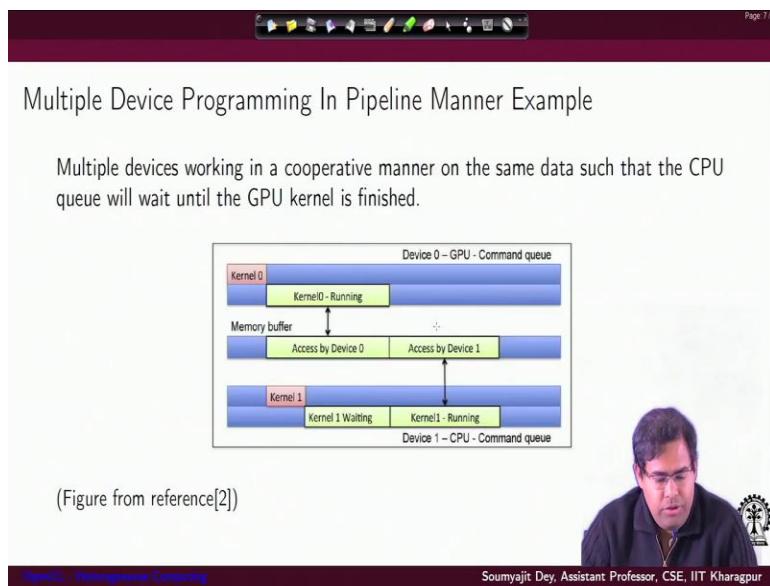
```

IIT Kharagpur

OpenCL: Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

That I have got this program to which we are setting up to the queues, one for the CPU and one for the GPU. And we are trying to have this kind of we have here that the kernel running on the GPU device who has got some output.

(Refer Slide Time: 01:08)



The kernel running in the CPU GPU device has got some output, and that output is going to be used by the kernel running on the CPU device. So just a small change here like spurred this model, what we have done is the CPU has device 0, GPU is device 1, I believe that has got exchange. So I hope that can be figured out anyway. So coming to the problem further, so suppose you want this execution and that you enqueue first in the GPU.

So let us say we call device 0 as GPU here although it is a bit different here device 0 CPU, but let us go with the figure here. I believe the concept would be clear so, for device 0, that is the GPU and you want to enqueue the kernel. So this is how you will do it, you include the kernel, but what you do is along with that you have a declared events, events 0 CPU event, 1 GPU, and maybe I believe we will just say we will just say these are events GPU and event CPU.

So this event CPU event GPU. So, when this kernel executes, it will just output and event GPU and all you want is the second kernel should start only when the event GPU is available to it. So it is sensitive to event GPU that is why for the `clEnqueueNDRangeKernel`, you are in queuing this kernel at the point , you have enqueue this kernel for into the CPU queue, queue CPU, and for this kernel, you are waiting for the GPU kernel to finish.

So on when the GPU kernel finishes it outputs the event GPU event. Since I have it in the event list as I make it make these commands sensitive to even GPU. So when this is received, I will have the other kernel which is enqueued in the CPU queue this will start executing, and when this finishes, it will emit this event CPU. Now, then we have these two `clflush` commands like discussed earlier to flush both of the queues.

And this only guarantees that all queued commands to this to the command queues they will eventually be submitted to the appropriate device. And what of course there is no guarantee that they will complete after the `clflush` returns. So this is something we have already discussed that if I reach these are all in a synchronous commands such commands I am getting enqueue here in the host program like this is the host program I am using into enqueue commands.

But once these commands are enqueue when I do a `clflush` of each of the queued the `clflush` is ensuring that whatever commands have enqueue for the CPU whatever commands have enqueued for the GPU, they are dispatched to our devices, whether they are finished or not that I am not ensuring for I will have to put in something else like `clFinish`. So, this is a demonstrative quote to give you the idea that what is the ability here.

So, please mark this correction and also here, just like we are device 0 and device 1 for the CPU and the GPU, and this is a bit different, so, that will be another here now, without going to the detail less than a week, let us say that how I can make it more generic. For example, I want multiple devices programming working concurrently. And I have more number of devices let us say, I have got two devices, three devices now, and I have multiple devices working in a parallel manner.

And both devices, both of these GPUs do not use the same left buffer and they will execute independently. And the CPU queue shall wait until both the GPU devices finish write. So let us say that is what I want to happen. So maybe for the idea is that GPU kernel 1 will execute kernel 0 will execute in device 0, it will output something in the memory buffer, then kernel an in parallel kernel, so kernel 0 is executing in device 0 that is, that is the command you have one GPU.

Kernel one is executing in some other device, which is device one, which is the command to have another GPU. So as you can see that they are independent kernel, they can execute in parallel. And after they have executed let us say that they copy the data to the memory buffer. And then the requirement is I have a kernel 2 which is supposed to execute in device to here which is a CPU device. And I have enqueue this execution command here in the command queue of this CPU device with a dependency that this kernel should wait.

And it should start executing when the kernel 0 and kernel 1 both of them have completed their execution, they will then access them and then and then they will output the result.

(Refer Slide Time: 06:34)

Multiple Device Programming In Concurrent Manner Example

```

// A concurrent and pipelined model of multidevice execution with single
// context on a single platform having 2 GPU and 1 CPU device
// Create 3 command queues, 2 queues for the 2 GPUs and 1 queue for the CPU
// The enqueued kernel on the CPU command queue waits for the kernels on the
// GPU command queues to finish
// Both the GPU devices can execute concurrently as soon as they have their
// respective data as they have no events in their waitlist

cl_event event_gpu[2];
err = clEnqueueNDRangeKernel(queue_gpu_0,kernel_gpu,2,NULL,global,local,0,NULL
    ,&event_gpu[0]);
err = clEnqueueNDRangeKernel(queue_gpu_1,kernel_gpu,2,NULL,global,local,0,NULL
    ,&event_gpu[1]);
// CPU will wait till both GPUs are done executing their kernels
err = clEnqueueNDRangeKernel(queue_cpu,kernel_cpu,2,NULL,global,lo
    event_gpu,NULL);

clFlush(queue_gpu_0);
clFlush(queue_gpu_1);
clFlush(queue_cpu);

```

OpenCL:异构并行计算 Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, we are trying to see that how that can be done. So, overall in a summary, this is a implementation example of a concurrent and pipeline model of multi device execution, where they have a single context on the platform, but with two GPUs and one CPU device. So what we are doing is we are creating three command queues two queues for the two GPU And one queue for the CPU the kernels are enqueue on the CPU command to the wait for the kernels which enqueue on the two GPU command queues that are to finish.

And when both the GPU devices have executed concurrently and they have their and they have done their job they are not only I want the CPU kernel to execute, but these two GPU kernels they have no input dependencies so they can just go and executing without any event for which they have to wait so they do not have any waitlist. So since they do not have any waitlist, I can have this queue for GPU 0 and Q for GPU one, I can just immediately into the kernels I do not have any waitlist so the event waitlist is null for both of them.

But when this one is done, this kernel GPU kernel this kernels are done for GPU 0 and GPU one. They will be emitting the event GPU 0 and event GPU 1 now the functional they want to implement is that the CPU has should enqueue a kernel in a queue CPU and this kernel should wait for both of these events. So, that is captured by this event list. So, I have event GPU this should use waitlist containing two of these fields.

So only when both these events go to their complete status, then this kernel which has enqueued here can start executing so kernel CPU can only start executing after kernel GPU CPU in both the devices have executed and this dependency can be captured like this. I do not want any more dependency like there is no further execution based on the completion of the execution of kernel CPU.

So I do not have any further dependency this is set as null. And like earlier, we can just flush all the queue that means these commands are ensuring that all the previous commands from each of these queue have been dispatched. So this is something I will keep on saying this is again a host program, it is just giving that the kernel for execution. It is not forcing them to execute and the queuing is done along with dependencies.

So when the program reaches here, I will wait here until I ensure because it is a clflush is a synchronization primitive like the host program will go past this only when it is ensured that GPU 0 this queue the kernel that has been ensure include here that has been dispatched. So that is ensured then I will wait in clflush queue GPU one unless and until the kernel that was included here that has been dispatched.

And finally I will wait here and I will get first this point only when this kernel which is kernel CPU which is waiting here that gets dispatched that means it has to ensure that the previous kernel has finished and the events are fired. So that this can get a start executing. So in this way, the clflush actually forcing the execution to happen and with otherwise the host program cannot go beyond this point. So, with that we have our discussion on multiple device programming with multiple command to support for a stem context.

(Refer Slide Time: 10:17)

Multiple Command Queues With Different Contexts In Different Devices

- ▶ Context is created with respect to a particular platform
- ▶ For different devices from different platforms, we create multiple contexts
- ▶ For separate contexts created for different devices, synchronization using events would not be possible
- ▶ Only way to share data between devices would be to use `clFinish` and then explicitly copy data in and out of a given context and across contexts via host memory space

OpenCL Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, let us talk about a scenario where I have multiple command queues with different contexts in different devices. So, the typical scenario you will want to do is you have context you did with respect to particular platforms, for an in different platforms, you have different devices like that hypothetical example, we were trying to demonstrate earlier. In each of these contexts, you can have different devices and four different devices, you can create separate contexts and you can synchronize an instance you have now devices sitting in different contexts. You cannot synchronize execution of events across context that is not possible.

So in that case, you have to go through the host program. So the only way to share data between devices would be to use `clFinish`, because if you have `clFinish`, then those command queues have to actually, , I can go beyond the `clFinish` synchronization primitive, only when all the commands that have I have enqueue up to that point of time in that queue, all of them commit and finish their execution.

Then once that is done, I have the output data of that execution in that device available I have to explicitly, copied from one context to the other context. And then inside that context, I can start in giving commands for using that data in this separate context. So in this way, the idea would be that I have to orchestrate data movement across contexts and I have to use primitives like `clFinish`, to ensure that execution has actually finished in one of the context, so that the data is ready, which can be moved across to a different context.

(Refer Slide Time: 12:06)

Creating Multiple Command Queues With Different Contexts In Different Devices Example

```
cl_uint numPlatforms;
cl_uint numDevices;
cl_device_id * deviceIDs;
size_t size;
clGetPlatformIDs(0, NULL, &numPlatforms);
cl_platform_id platformIDs[numPlatforms];
cl_context contexts[numPlatforms][16] = {0};
cl_command_queue commands[numPlatforms][16] = {0};
clGetPlatformIDs(numPlatforms, platformIDs, NULL);
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if we do look at such examples that you have, you are creating multiple command queues with different contexts that are sitting in different devices, you have different contexts sitting in different devices, and you are creating a different set of multiple queue for the devices. And in this example, as you can see, the typical code would be like you get the platform idea like earlier. And then of course, you should have a context at it and you have a set of queues right now CL command queue array for each of the platforms.

So, as you can see you have a context array and you have a command queue array. So, you have context array for proper platform. And you can also have this command queue array as we discussed in the first thing, you will do is you will try to discover the set of platforms and their platform IDs.

(Refer Slide Time: 13:06)

```

for(int p=0; p < numPlatforms; p++)
{
    errNum = clGetDeviceIDs(platformIDs[i],CL_DEVICE_TYPE_ALL,0,NULL,&numDevices
                           );
    deviceIDs = (cl_device_id *)malloc(sizeof(cl_device_id)*numDevices);
    errNum = clGetDeviceIDs(platformIDs[i],CL_DEVICE_TYPE_ALL,numDevices,&
                           deviceIDs,NULL);

    for(int d=0; d < numDevices; d++)
    {
        contexts[d] = clCreateContext(NULL, 1, deviceIDs[d], NULL, NULL, &err);
        commands[d] = clCreateCommandQueue(contexts[d],deviceIDs[d],0,0);
    }
}

```

OpenCL Interoperable Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then once you have got the platform IDs, you are trying to get the device IDs. So, this is the kind of code we are a bit familiar with. So, inside the loop you are looping inside the number of platforms. For each platform you get inside the loop, you identify all the devices in the platform using this command. As you can see that if you are using CL get device IDs with the flag, CL device type all is discovering all the IDs of all the devices have in total number of devices and that is pushed into this non device variable.

And then you allocate suitable amount of memory which is equal to the number of devices multiplied by the size of CL device ID you look at that must be able to this device IDs at it. And then again, you give a call to CL get device IDs. So that for this specific platform, we are inside a loop in each iteration, we are inside one platform. So for that platform, all the devices, ideas that have been discovered, , they are actually provided in into this device ID array.

Now, for all of these devices together, what you can do is, you run this kind of loop using this loop, you are creating a set of contexts, separate context for each device as you can see. So, technically what we are doing is we have a set of platforms. For each platform, you discover the set of devices for each device, which is whose ID is stored in this device IDs and this is done in the last line for each device, you store its IDs in the device IDs array. For each of the device IDs, you create one context for each of the device IDs inside each context, you create a command queue so these are you move forward.

So this is again, this is an example problem, we are trying to show examples of pipeline execution of multiple kernels in different command queues in this case. So what we have done is we have defined two contexts for two devices. And each of these devices have their own command queue. So let us say this is one platform, one of the platforms inside this platform, I have two devices, D1 D2. For each of these devices, I have created a context or better to sit here and I find and for each of these I have inside each of these contexts these devices, and they are command queues, .

So I would say I have a queue 1 and I have a queue 2. This is the kind of setup we are looking at. So now considered that I want to do execute things in such a way that there is a dependency between kernel 1 and kernel 2 that is output of kernel 1 is user input to kernel 2 what we do is we have a CPU and a GPU in the platform. And kernel I assign kernel 1 to the CPU and I send kernel 2 to the GPU. So for doing this whatever code is required, let us say that has been already written here.

And in that way, and I have set up both of these queues. One is the queue for the CPU, which is one of the devices one is the queue for the GPU use other device and they have been assigned the task executing kernel 1 and kernel 2. So, first I do a write buffer. So, in the queue of CPU, I do a write of some buffer A and then in the queue of the CPU, I do a write of the other buffer B and then I execute kernel one in the CPU which will be working on the data of buffer A and buffer B.

And once it is done, if will output this kernel event, now, I have this read buffer command here. So that is a way I can ensure that well, this has then this command which I enqueue for a launching a kernel which is kernel alone, and that has been finished when this kernel event shall here, and this is in the event list for speed buffer. So only when this Colonel has been actually launched from the queue to the device, the CPU device and it has finished execution, I will have these event ready and with this event firing now I have the read buffer command read execute.

And with that read buffer, I will have the buffer see which is the output of kernel one it will be transferred from the it will be transferred and once it is transferred, I will have the read event this read event set and at the end I have this CL finish here if I go with this the host program goes

beyond this point. That means, all these commands that have queued up in the CPU queue, they are actually dispersed even though they are not finished execution.

So, here I am blocking it until all previously queued commands this is clfinish or clflush. So, I am actually blocked here until then unless all these commands which has been queued up they have been dispersed as well as all of them finish. Because is clfinished, there is not a clflush. So, they have their finished execution in their stated device.

(Refer Slide Time: 13:06)

The slide title is "Multiple Device Programming In Pipeline Manner Example". Below the title is a block of C code:

```
clEnqueueWriteBuffer(queue_gpu,bufferD,CL_TRUE,0,10*sizeof(int),h_c,0,NULL,&
                     writeEventA);
clEnqueueNDRangeKernel(queue_gpu,kernel2,1,NULL,globalws,localws,2,eventList,&
                      kernelEvent);
clEnqueueReadBuffer(queue_gpu,bufferOut,CL_TRUE,0,10*sizeof(int),h_out,1,&
                    kernelEvent,&readEvent);
clFinish(queue_gpu);
...
```

Below the code is a video frame showing a man with glasses and a dark jacket. At the bottom of the slide, there is a footer bar with the text "OpenCL Heterogeneous Computing" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So, all that is happening is almost same. Only thing is here we are showing that we are working with different contexts, we have this kernel data, which is there in different contexts. , since they are in different contexts, so I have to make these reads and writes. So I put the data in the context of the CPU. And then, and so for that, I have to enqueue commands. And then I execute the kernel, after executing the kernel, have to treat it back, because it is not in there. , the GPU device is not in the context.

And then I have to read back the data. From I have to read back this the data to the context of the GPU. So, for that, I will now again have this write buffer command executing. So, it will copy some buffer d to the queue of the GPU and then I will have kernel 2 executing in the GPU. So, as you can see that this is going to execute only when so, I have this kernel to which is executing

here. Now, for this I have a guarantee the since I have put in a clfinish so, all these operations that were done earlier, they are finished.

So, after that I have this kernel 2 executing and once I have content to complete it, which is being ensured by the event kernel event, which is there in the sensitivity list of this command for this read buffer command. So since so only when kernel event fires, that means this come this is done, then only I am going to read back the output. Assume that is there in some buffer out. I am going to read it back. And then I give again, I fire the read event command. So again, I will put in a clfinish in the queue GPU.

(Refer Slide Time: 21:23)

Multiple Device Programming In Concurrent Manner Example

```
//Two contexts for two devices, each having its own command_queue
// There is no dependency between the two kernels and can execute concurrently
...
err = clEnqueueNDRangeKernel(queue_gpu,kernel1,2,NULL,global,local,0,NULL,NULL
);
err = clEnqueueNDRangeKernel(queue_cpu,kernel2,2,NULL,global,local,0,NULL,NULL
);
...
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So I have this two context for the two devices, each of them have their own command queues. And there is no dependency between the two kernels and they can execute completely concurrently. So if I want that kind of behavior, then I would just enqueue them in parallel without any kind of dependency coming back here. So if I have this line of the code, the execution is pipeline. That means kernels which is executing in the CPU, once it is done.

(Refer Slide Time: 22:04)

So, this is the other kernel is executing in the CPU. The write commands actually ensure that buffer A and buffer B we are copied in the context of the CPU and then the kernel execution of CPU and then the output of the kernel which is buffer C is copied from the context of the CPU to the host side and then once I have copied this, so these are the host side data points maybe we were not showing how they are initialized assume that h a, h b or h c are available in the host side memory.

They have been declared null of h a, h b and h c has been initialized. So, we copy the content of h a and h b into buffer a and buffer b which are in the context of the CPU device, execute the kernel and then after the kernel executes, I have the data available. So, I have a read buffer now, which is ensuring that the kernel executed through this event dependency of kernel event and then it reads back and it reads back the content of buffer C to the host side memory h c.

And then this is being written to the there is a write buffer command, which is writing h c in some buffer D, which is there in the context of the GPU. So, maybe if we just repeat what things are going on so, I have these two contexts they are they are separate contexts. That is why we are doing all this reads and writes. So, you have CPU, you have the queue for that we have in queue in commands, you have the GPU, you have just put them in separate contexts.

And we are trying to see that since they are in separate contexts, I have to set up the offers the buffer A and B, where I copy the data from the host side h underscore A and underscore B they

will the kernel will execute it will have the data in the buffer C from that I will copy back to h c host side memory then I will be copying h c to the GPU side buffer D and now I will have this ND range kernel GPU which is going to start execution and once it has executed you will see that it will output again another kernel event.

So, it will again output and kernel event here and this kernel event is actually being used for ensuring that now when to start that I will do the read buffer again So, This output is available here in some buffer out the buff output of this kernel two is available in some buffer out and then it is copied back to the host side memory which is h out. So, these are things are happening and finally, we have a clfinish to flush out both this queue and the other queue here.

So, this is how we are trying to show that how things are pipeline that means, when the execution is done, we are able to actually copy things to the next device. Of course, when and how this is useful. That is something that you have to figure out where then is at all useful for the application. We are just trying to show that how things can be done that how you can have device executing in one context and you can get output from that context and you can copy it to the other context.

We are trying to show that it is necessary when you have devices sitting in different contexts, then the memory objects are not shared. So, whatever is the output of a corner executing inside one context which is to be copied to the host side memory and it has to be copied back to the other context. Now, of course, if there are no dependencies here, you but as you can just enqueue stuff in two different contexts accused in two different contexts and they can just continue execution.

(Refer Slide Time: 26:34)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Concurrent Kernel Execution". Below the title is a bulleted list of points about concurrency. To the right of the list is a portrait of a man with glasses and a dark shirt. At the bottom of the slide, there is a footer bar with the text "OpenCL Heterogeneous Computing" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

- ▶ Concurrency is property of a system in which a set of tasks in a system can remain active and make progress at the same time
- ▶ Programmers need to identify the concurrency in their problem and efficiently schedule in the host program
- ▶ The concurrent tasks can be running-
 - ▶ Different kernels from different independent applications
 - ▶ Different kernels without dependency between them from same application
 - ▶ Partitioned instances of same kernel that are SIMD in nature

So, overall, if we are talking about such concurrent control execution, so then this concordance is a property of a system in which you have a set of tasks and they can remain active and make progress. So that has seen at the same time, the programmer would need to identify the concurrency in their problem in case you are you want to develop a program for such concurrently execution in heterogeneous platform. And if the point is the program id up to remember the differences.

The abstractions and accordingly do the mapping of the workloads or the kernels and their dependencies into different devices with through the abstraction of device kernels and platforms. And also the programmer has to use this idea of events and their synchronizations to efficiently schedule the host program. And the concurrent tasks that may be running will be of the following kind, you can have different kernels for different independent applications, you can have different kernels without any kind of dependency between them.

They are from the same application that is also possible to have two dependent kernel two independent kernel running maybe on they are both of their outputs, something else can depend. But these two kernels have no dependencies, there is also part possible. So this is also when you can have concurrency of tasks or kernels and you can have situation the one kernel, it has been partition that is let to the kernel requires 1024 threads to execute.

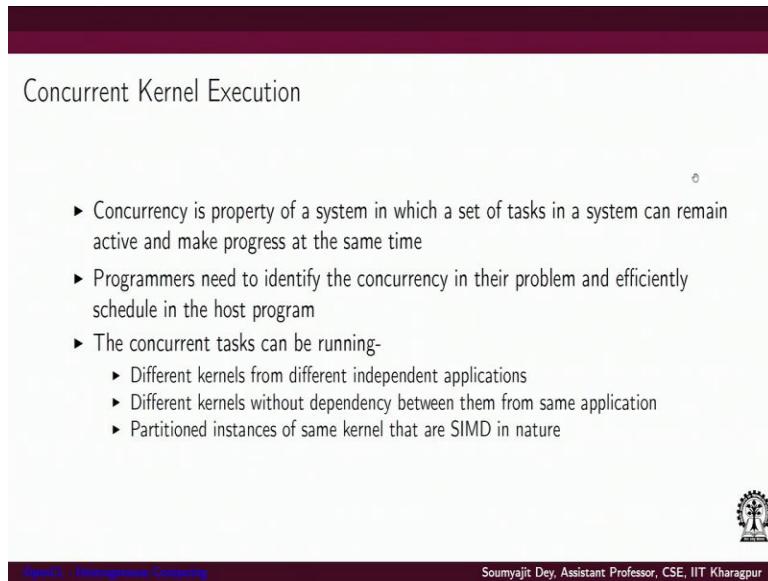
You just partition the input data space to two different devices, you copy it in a partition way to two different devices. And you launch half of the number of threads into device 1 how the number of threads in the device to and execute them. So that is also a partition instance of the same kernel and this also like concurrent tasks execution. So these are the possibilities in which you have concurrent all execution in a multi device platform with this, we like to end this lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 50
OpenCL – Heterogeneous Computing (Contd.,)

Hi, welcome back to the lecture series on GPU architectures and programming. So, I believe in the last lecture, we have been some are kind of discussing more around concurrent kernel execution and how concurrent kernels can be mapped.

(Refer Slide Time: 00:38)



Concurrent Kernel Execution

- ▶ Concurrency is a property of a system in which a set of tasks in a system can remain active and make progress at the same time
- ▶ Programmers need to identify the concurrency in their problem and efficiently schedule it in the host program
- ▶ The concurrent tasks can be running-
 - ▶ Different kernels from different independent applications
 - ▶ Different kernels without dependency between them from same application
 - ▶ Partitioned instances of same kernel that are SIMD in nature

OpenCL – Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

To an underlying OpenCL device. So, in that context we talked about the following possible options like if you have concurrent tasks, then 1 option can be that you have different these tasks are all different kernels which are coming from different independent application so they do not have any dependency and so, also the other option is that will there is a same kernel, but multiple paths will be the same application, but from the same application multiple kernels have been launched.

And they do not have really any dependency among each other. So then you have them, you can just run them concurrently across multiple devices. Also, suppose you have 1 kernel and you want to exploit more compute capabilities of the hardware another option can be that you partition the

kernel across multiple devices and execute them. So we will fine will I mean, there has to be some motivation like while why you will want to do that. We will discuss on the motivation.

(Refer Slide Time: 01:42)

The slide has a dark blue header bar with the word 'executing' in white. Below the header is a white background area containing a list of bullet points. At the bottom right of the slide is a photograph of a man with short dark hair, wearing a light-colored hoodie, sitting at a desk and looking towards the camera. The slide footer contains the text 'OpenCL_Heterogeneous Computing' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

- ▶ Multiple applications can executing concurrently across multiple devices
- ▶ Heterogeneous computing can efficiently exploit both CPU and GPU devices by invoking OpenCL's data transfer APIs, query memory objects, and data/work partitioning between the multiple devices
- ▶ Technique is used to partition the workload of a single kernel across multiple available OpenCL devices

So, in general, you can have as we are discussing that you have this multiple kernels or multiple kernels of the same application are they belonging to different applications they are executing concurrently but in case you want to increase the concurrency more 1 option is in case you have also available devices, you partition the kernel across devices. But the issue is how can it really be done.

First of all, the good thing about doing it is that you can exploit both CPU and GPU devices by using both the devices to kind of execute the same kernel. And this is 1 advantage that OpenCL has, because an OpenCL kernel does not make a distinction between CPU and GPU devices; it is built in for heterogeneous computing. So you can actually split the kernel into two sister kernels, I would say and dispatch them to multiple devices.

And I mean, of course, the 2 devices have to be available in the system for doing that, but assuming they are available, then what is happening is you are exploiting more amount of parallelism in that way. And this can really be done using OpenCL returns for APIs query or I mean memory objects and other and support for multiple devices that is available.

(Refer Slide Time: 03:11)

Partitioning

0

- ▶ Some kernels perform better on either CPU or GPU devices
- ▶ While executing a kernel on a specific device the other available devices may remain idle
- ▶ Partitioning is a technique used to partition the data efficiently and distribute them across multiple OpenCL devices
- ▶ Then launch same kernel with partitioned data across multiple OpenCL devices
- ▶ The partitioned kernels can run concurrently on different devices reducing execution time



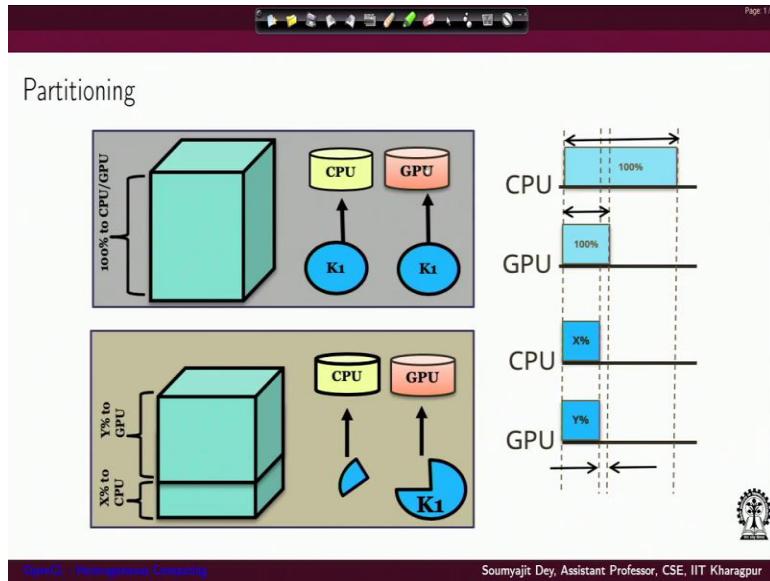
OpenCL - Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, to motivate a bit more, in general, you can have kernels, which are GPU friendly kernels, I would say, because let us say the kernel has too much of parallelism. And inside each parallel thread, there is not much of control oriented dependency. So then it is a very much GPU oriented kernel. And it makes sense to execute in the GPU whereas you may have an OpenCL kernel with lot of branches, divergences and control dependencies inside the kernel.

So that will make up case for CPU only execution. So overall, you have a design space, like whether you want to execute a kernel fully in the CPU, whether you want to execute the kernel across CPU and GPU or whether you want to execute the kernel fully in a GPU. So, just to create a problem space here this is what we can do like suppose.

(Refer Slide Time: 04:20)



You have this kind of kernel and you are trying to submit and this is the full job, there is a full data space on which this kernel K 1 is going to work. So, it is essentially defining a transformation which will be carried on the internet data space. You can dispatch that kernel to the CPU and get the work done on this data service. You can dispatch it on the GPU, you can get the job done on the on this full data space. Or you can split this data space to individual memory segments.

And get to Kernel 1, Kernel dispatches to CPU 1 Kernel dispatches to GPU and then you have both of these devices engaged in parallel for doing the job, which is a good thing if you disperse it only since CPU the GPU is idle plus in case there is no other application or some other kernels application some other kernels which are running here, . But in case it this is free, there is no other kernel which is looking to execute here there, why not to split it like this and execute some part on the CPU and some part of the GPU.

(Refer Slide Time: 05:29)

The screenshot shows a presentation slide with a dark header bar containing various icons. The main title 'Partitioning' is centered at the top. Below the title is a bulleted list of six points. At the bottom of the slide, there is footer information including the OpenCL logo, the name 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur', and the IIT Kharagpur logo.

- ▶ Some kernels perform better on either CPU or GPU devices
- ▶ While executing a kernel on a specific device the other available devices may remain idle
- ▶ Partitioning is a technique used to partition the data efficiently and distribute them across multiple OpenCL devices
- ▶ Then launch same kernel with partitioned data across multiple OpenCL devices
- ▶ The partitioned kernels can run concurrently on different devices reducing the total execution time

OpenCL Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, we will first try to figure out when and how these things make sense. So, there is a primary motivation that it may happen that some other devices are available, they are idle, so, it makes sense to partition the data space and dispatch multiple kernels on the partitions data space across engaging with multiple OpenCL devices. And effectively what you are doing is you are kind of exploiting the systems available concurrency more efficiently. But let us also discuss when and how this thing makes sense.

So suppose you have this data space. And you have. So let us say this is the host memory and since it is attached to the CPU device. So, you are bringing the data to the CPU cache and then you are executing it on the CPU pipeline here. And then you have an off chip connection to the bus. And on this bus, you have a connection to the GPU memory, let us say that device memory and the device in general. So this is like the entire device. There may be multi devices.

So what is the overhead of actually partitioning the data space and executing in the CPU, or I mean across the CPU and the GPU. So, if you do a CPU only execution, there is no data transfer involved. So you will launch as many I mean, you are OpenCL Kernel will launch as many threads as the number of total number of parallel assembly operations supported by your CPU. And those threads will actually execute in parallel, and they will compute on these data service.

So, if it is a I mean, brand heavy Kernel, I would say, then make sense to have it in the CPU by the computational nature, its CPU friendly kernel. So then it is makes sense not to send it to the GPU. Why because first of all, GPU are not very efficient. They do not have speculative execution and other supports that you have in CPUs. So, for handling branches, the CPU as a device is much more efficient.

Now, when you are thinking of executing on the GPU side, then you have the problem, which is that this is then you have the host program which is executing on the CPU, this host program has to first copy the data from is from the host side memory, which is this 1 will have to copy the data from this side memory to and to the GPUs DRAM which is here. So, there is some additional data transfer over it that you have.

So, from the memory that is referenced by the CPU, you have to first you have to first define buffers in this memory and you have to copy that data into this memory. Once the input buffers are set up, then you are going to launch the kernel . So, let us say that CPU only execution time for a kernel is $exCPU$. The GPU only execution time for the kernel is $a exGPU$ and let us for the sake of generality assume that in total you have 100 threads.

And if you are working with x number of threads, let us say a factor of x , then the execution time in the CPU would be x multiplied by $exCPU$ and the rest of the threads. So let us say you have $100 - x$ threads, which are executing in the $exGPU$ assuming that you are splitting this number of threads in parallel across the devices question is when will I do it? I have I will leave now the issue is there is this data transfer time .

So, this factor x have to choose in such a way that the execution time it is multiple I mean for x number of threads in the CPU should actually equal the execution of time of the rest of the threads in the GPU plus the data transfer overhead of this $100 - x$ of the requisite memory elements for this $100 - x$ threats, because you have to send these many elements to the GPU memory and you want to do the compute, .

So then I would be able to say that let us say the origin or execution time is this or in this GPU it is this but when I partition it, the execution time becomes equal to this x multiplied by this ex CPU. So just a small correction, there is 2 here. So, if we are saying that x is a factor, so it is like $1 - x$. So, you multiply it by 100 here, and you multiply it by 100 here. So let us say x is 0.2 that means you are executing 20 threads in the CPU and the rest of the 80 threads in the GPU.

And you are in case it happens that executing 20 threads in the CPU takes same time as a time required to copy the 80 threads corresponding data elements to the GPU copied back and also do the operations then this is execution from that is required . Now, it may so happen that the data transfer for it is too high. For some kernel, in that case, this is not a good option . Because if the data transfer of it I mean, outwits whatever benefit you can have to parallel execution.

Then this does not make sense you will only like to have it we do not like to be here if the available parallelism for whatever threats you are now launching in the GPU is much more with just the 2 is over it . Otherwise this kind of execution will not make sense. But this is for the kind of devices where we have a discrete GPU. So let us say you have an Intel CPU, an Nvidia GPU or some in AMD GPU, let some other vendors GPU use there, they are connecting through a PCI Express Bus.

So both are separate chips. Now, in that case, this was execution timing, I mean some possible in these are all very, we are making a lot of assumption here. We are saying that the execution time is proportional with the number of threads and all that and so and also since this is a factor, so and this is the execution time for 100 threads, so we can remove it is just x multiplied by this and $1 - x$ multiplied by this and, but as we are saying that for integrated GPU, this execution model is not going to hold.

Because here we actually capture the data transfer Raiders for over it. Now the question is what is an integrated GPU? Well, you have modern CPUs and come I mean other I mean devices from different vendors, we are on the same die you can have a CPU device and a GPU device and also shared memory . So, there is no data transfer over it when you are defining buffers for this class of devices.

So, when you partition corners for this class of devices, that data server is not there. So the calculations have to be done in a different way. So all these points to the idea that maybe in certain cases, it may sense that you really partition the data space and launch multiple kernels, in case you have multiple devices, which are freely available in the system. So with this motivation, if we go forward here, and let us see how really this can be done.

So in this classic example, what we are really trying to say is, let us break it 1 into 2 parts. So, you split that in a way that x number of CPU threads go to CPU y percent of threads go to GPU, total is 100 here. And accordingly, you create 2 different kernels, 1 for the CPU and one for the GPU. And let us say for the system, the CPU only execution time is this the GPU only execution time is this but if split and execute, since they have some parallelism, they will execute in a shorter amount of time.

So here, the overall execution time was this because up to this point, the whole job is not done and GPU overall execution time was this so this is like a picture for a GPU friendly kernel. That is why I make a small split, just to load balances since anywhere GPU friendly Kernel, I sent smaller number of threads to the CPU and more number of threads to the GPU. So, that after this much time, both of them are getting done.

(Refer Slide Time: 15:19)

The screenshot shows a presentation slide with a dark header bar containing icons for navigation and search. The main title is "Manual Partitioning Using Example". Below the title, there is a list of bullet points:

- ▶ Vector addition of two vectors of size LENGTH
- ▶ Partition across 1 CPU device and 1 GPU on 2 separate contexts
- ▶ Partitioning across CPU and GPU (20% to CPU and 80% to GPU)

In the bottom right corner of the slide area, there is a video frame showing a man with dark hair and a light beard, wearing a light blue hoodie, speaking. At the very bottom of the slide, there is a footer bar with the text "IIScC1 Parallel Computing" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So let us take an example of such a piece of code, considered this idea of how I mean you can I mean as a programmer, partition a kernel into 2 kernels for execution in parallel. Of course, you can write a tool which will automate this transformation, you can do a write a compiler kind of tool, which will take as input a kernel and take a partitioning factor and create multiple kernels for multiple devices which can execute in parallel.

Now, 1 important thing I would like to say that when we are assuming we are doing this kind of partitioning, we are, we are also thinking that there is no dependency among data in the data space. So when I launch a thread where we are, thinking that the kernels are such that the thread does per unit data computation. So, the I th thread, let us say is working on elements from A_i and B_i in the input buffers.

So the IS threat is not accessing let us say, $A_i - 100$, that will actually complicate the speeding process. So, of course, in that case, also you can write a split code. But in our examples, we are making that simplistic assumption, just to motivate the idea here. So, here we will show an example. Where do you do vector addition of 2 vectors, 2 sizes of length equal to some constant, and you partition across 1 CPU device and 1 GPU device, you create 2 separate contexts, 1 for each device, and second 20% of our threads to the CPU and 80% of the third GPU.

(Refer Slide Time: 16:53)

Manual Partitioning Using Example

Initialisation and declaration host data

```
...
size_t dataSize = sizeof( float ) * LENGTH ; //LENGTH is size of vector
float * h_a = ( float * ) malloc ( dataSize ); // a vector
float * h_b = ( float * ) malloc ( dataSize ); // b vector
float * h_c = ( float * ) malloc ( dataSize ); // c vector ( result )
cl_int err ; // error code returned from OpenCL call
// Fill vectors a and b with random float values
int i = 0;
for ( int i = 0; i < LENGTH ; i ++ ) {
    h_a [i] = rand () / ( float ) RAND_MAX ;
    h_b [i] = rand () / ( float ) RAND_MAX ;
}
...
```

OpenCL: Unconventional Computing

Page 3/3

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, initially, I mean these are simple code. Which is just that you are just by doing a partitioning of data space executing across devices. So, you have a data size and corresponding to that you initialize 3 vectors in the host or 3 buffer for storing the input vectors in the host memory and also the output vector space. Now so, you initialize the 2 input vectors, h a and h b.

(Refer Slide Time: 17:22)

Manual Partitioning Using Example

Create buffer object for CPU

```
...
//Partitioning the vector across CPU and GPU
size_t dataSize_CPU = sizeof(float)*(LENGTH)*(20/100) ;
size_t dataSize_GPU = sizeof(float)*(LENGTH)*(80/100) ;
// Create array in CPU memory
cl_mem d_a_CPU = clCreateBuffer(context_CPU, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, dataSize_CPU, h_a, &err);
cl_mem d_b_CPU = clCreateBuffer(context_CPU, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, dataSize_CPU, h_b, &err);
cl_mem d_c_CPU = clCreateBuffer(context_CPU, CL_MEM_READ_WRITE,
dataSize_CPU, NULL, &err);
```

OpenCL Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then you create suitable data spaces for I mean for partitioning, what you do is you calculate what is the data size for the CPU and for the data size of the GPU. So, assuming we are doing an 80 and 20 split, then you need to create suitable buffers in the CPU memory. So, what we do is, you create a using CL create buffer, you create a buffer in the CPU memory inside the context of the CPU . So, you create in the context of the CPU. So, here again we are assuming that all these devices have been nicely set up.

And you are just having 2 context setup one for a CPU device and one for a GPU device . Also we are assuming that the host program is not running in any of them that is why we will actually be transferring data to the CPU and the GPU both . So you do a CL create buffer two and four using which you create a buffer in the CPU for storing data of the size of the partition that we have defined is the size of the partition and the CPU. Similarly, to the size of the partition in the GPU, you create a buffer in the GPU side .

So these are the 3 calls, which are all on the CPU side for creating 3 input buffers for the CPU, 2 of the input buffers, and 1 for the writing the output data in the CPU, using CL create buffer. As you can see, for all of them, the context you CPU and buffer sizes, data size which is calculated here. And since these are input buffers, so they are in read only form.

(Refer Slide Time: 19:07)

Manual Partitioning Example

Create buffer object for GPU

```
// Create array in GPU memory
cl_mem d_a_GPU = clCreateBuffer(context_GPU, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, dataSize_GPU, h_a, &err);
cl_mem d_b_GPU = clCreateBuffer(context_GPU, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, dataSize_GPU, h_b, &err);
cl_mem d_c_GPU = clCreateBuffer(context_GPU, CL_MEM_READ_WRITE,
    dataSize_GPU, NULL, &err);
...

```

OpenCL Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And once this is done, you also do a same thing on the GPU side GPU device, again will just repeat we are assuming the host programming is not executing in any of these devices. That is why you have to create buffers in both of these devices. So you in order to set up the problems for the GPU device, you also create the input buffers and output buffer where it will write the result.

(Refer Slide Time: 19:30)

Manual Partitioning Example

Writing data from host to device

```
...
//Write the data from host to the CPU device
err = clEnqueueWriteBuffer( commands_CPU, d_a_CPU, CL_TRUE, 0, sizeof(float)
    * dataSize_CPU, h_a, 0, NULL, NULL );
err = clEnqueueWriteBuffer( commands_CPU, d_b_CPU, CL_TRUE, 0, sizeof(float)
    * dataSize_CPU, h_b, 0, NULL, NULL );

//Write the data from host to the GPU device
err = clEnqueueWriteBuffer( commands_GPU, d_a_GPU, CL_TRUE, 0, sizeof(float)
    * dataSize_GPU, h_a+dataSize_CPU, 0, NULL, NULL );
err = clEnqueueWriteBuffer( commands_GPU, d_b_GPU, CL_TRUE, dataSize_CPU,
    sizeof(float) * dataSize_GPU, h_b+dataSize_GPU, 0, NULL, NULL );
...

```

OpenCL Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then you do the data transfer of data size underscore CPU amount into the CPU device. And this is the command to the CPU command. So you set up the 2 input, the input vectors, input buffers of size. So these are the buffers d a CPU and d b GPU. I mean, they both are the input buffers d a CPU and d b GPU, which are of size data size CPU. And they are essentially taking values from h a and h b which are the original input that is sitting in the host side memory.

And then you do the thing that you just do similar write buffers for the GPU device. So again from the host side memory, you copy data of size data size GPU to the GPU device. As you can see, you are copying both from h a and h b, but you are copying from an offset because starting from the h a address you have already copied data up to size data size CPU. So now you copy with an offset of from h a + data size CPU, and the amount of data you copies data size GPU, .

So this is how you do it so you execute and essentially you are enqueueing this write buffer commands. So you have enqueue these 2 write buffer commands here and to the CPU device we have also include to write before comments to the GPU device.

(Refer Slide Time: 21:03)

The screenshot shows a presentation slide with the title "Manual Partitioning Example". Below the title, there is a section titled "Executing and reading output from device to host". The slide contains the following C code:

```
...
size_t global_work_size_CPU = dataSize_CPU;
size_t global_work_size_GPU = dataSize_GPU;
size_t local_work_size=512;
err = clEnqueueNDRangeKernel(commands_CPU, ko_vadd, 1, NULL, &
    global_work_size_CPU , &local_work_size, 0, NULL, NULL);
err = clEnqueueNDRangeKernel(commands_GPU, ko_vadd, 1, NULL, &
    global_work_size_GPU , &local_work_size, 0, NULL, NULL);
    ...
err = clEnqueueReadBuffer( commands_CPU, d_c_CPU, CL_TRUE, 0, size
    * dataSize_CPU, h_c, 0, NULL, NULL );
err = clEnqueueReadBuffer( commands_GPU, d_c_GPU, CL_TRUE, dataSize
    sizeof(float) * dataSize_GPU, h_c+dataSize_CPU, 0, NULL, NULL );
...

```

At the bottom of the slide, there is a footer with the text "OpenCL Heterogeneous Computing" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur". There is also a small portrait of a man in a purple hoodie on the right side of the footer.

Once this is done, you have to execute and read the outputs from then I mean devices to the host. So first you execute the kernels. So you have to execute the kernels in the comment queues of I mean f by enqueueing suitable Kernel launch commands in both the commands queues of the

commands CPU as well as the commands GPU. So as you can see, we have 2 Enqueue ND range kernel calls, one for the command CPU queue, and then for the commands GPU queue.

So once this is done, then the host program we like to copy that the results from the 2 output queue output buffers that have been set up in the CPU and the GPU device. So those are the buffers. This is CPU and this is GPU, so you copy them back again, by enqueueing suitable read buffer comments into the individual comment queues of the CPU and GPU device.

(Refer Slide Time: 21:58)

The screenshot shows a presentation slide with the title "Manual Partitioning Example". Below the title is the subtitle "Checking output". The slide contains the following C code:

```
//Synchronize
clFlush(commands_CPU);
clFlush(commands_CPU);

// Test the results
int correct = 0;
for(int i = 0; i < count; i++)
    if(h[c]==h_a[i] + h_b[i])
        correct++;
printf("Vector add to find C = A+B: %d out of %d results were correct\n",
       correct, count);
...
```

On the right side of the slide, there is a video feed of a man with glasses and a purple hoodie, identified as Soumyajit Dey. The video feed has a small red dot in the bottom right corner. At the bottom of the slide, there is a footer bar with the text "OpenCL Programming Examples" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So once that is done, you will want to synchronize and flush with the command queue of both the devices . So, just to ensure that all the comments have been dispatched for both the devices and finally is the results. So, that would be our idea of doing the partitioning of an OpenCL program. And as you can see that we have already explained that what is the situation in which you have some advantage?

First of all the data transfer time should be such that it is not too heavy, but you have that you can you are doing enough work in the GPU in parallel to either transport time and overall you have an advantage by exploiting the parallelism. So, that also depends on the style of the kernel. So, what is the nature of the kernel supports this part I mean is such that partitioning gives you an advantage and you have multiple devices available.

Then it will make sense to do the partitioning. Also, as we said it may happen that you have devices let us say you have several devices like modern mobile associates, where you have associates with 1 chip CPUs and GPUs, multiple CPU devices, multiple GPU devices available which shared memory was interference with shared memory interfaces. So that they can just I mean copy data quickly across a shared memory,

I mean, they can produce the outputs and write to a shared memory. So there is no question of data transfer reference for it for such kind of devices. And they are partitioning also does not have too much overhead with respect to the data transfer. So depending on what kind of device you choose, you have to select whether we have to first decide whether partitioning is actually helpful or is it the kernel is the kernel is very much CPU friendly, or very much GPU friendly, I would say that it does not make sense to partition.

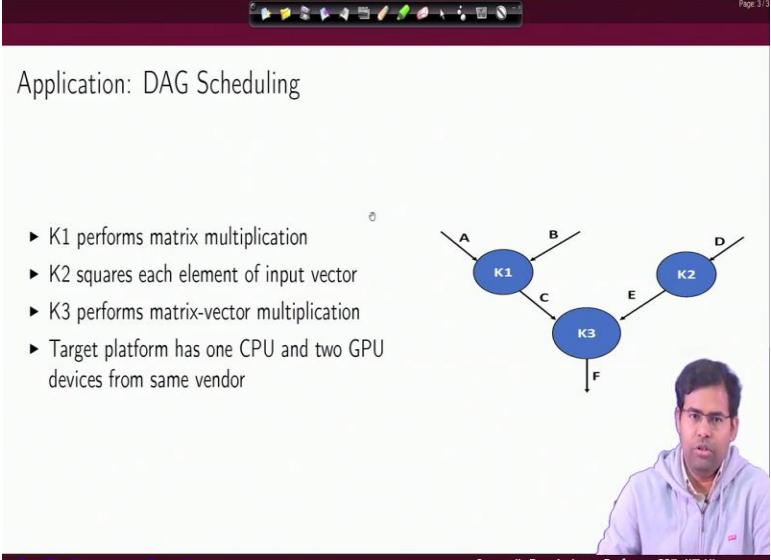
There are kernels with different kinds of characteristics. So you have to understand the characteristics of your kernel, whether it is a branch checking Kernel or compute heavy and parallel Kernel and also, whether there is enough number of independent processing for each kernel thread, so that the data space can be split nicely and dispatched across devices. So, you gain you may gain depending on the kernel characteristics by exploiting the parallelism provided you have multiple devices available. So, that is the advantage of partition execution of OpenCL programs. And with this we will like to end this lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 51
OpenCL – Heterogeneous Computing (Contd.,)

Hi, welcome back to the lecture series on GPU architectures and programming. In the last lecture we have covered how OpenCL kernels can be partitioned across multiple devices and executed in case and when it is advantages and how that helps in terms of gaining more exploiting more concurrent execution. And we have also understood that this is a facility available with OpenCL because the OpenCL kernels are common in case the vendor provides OpenCL implementation for any device. The same kernel code can be mapped through different possible devices provided you have suitable OpenCL libraries and devices.

(Refer Slide Time: 01:00)



Application: DAG Scheduling

- ▶ K1 performs matrix multiplication
- ▶ K2 squares each element of input vector
- ▶ K3 performs matrix-vector multiplication
- ▶ Target platform has one CPU and two GPU devices from same vendor

The diagram illustrates a task graph for DAG scheduling. It consists of three circular nodes labeled K1, K2, and K3. Inputs A, B, and D are connected to K1 and K2 respectively. K1 and K2 are connected to K3. K3 has two outputs, C and F. The video player interface at the top shows the slide number as 3/3.

OpenCL - Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

The Next topic we like to touch in is how to use this concept in a general case where you have a wearable object application DAG for directed a cyclic graph. And you want to execute this kind of an application on a system architecture company. Where do you have multiple devices. So what we are assuming in this small example is said this is a task graph. So you have 3 input buffers, A, B, and D. So you have 2, kernels who can execute initially in parallel, let us say K1 and K2, K1

let us say this again, an example performance metrics multiplication, K 2 takes 1 input buffer, all for all the elements. It performs square computation for each of the elements.

And they produce their outputs in buffer, C, and E. And these are basically buffers which are input buffers for the kernel K3, and this kernel will live in performer a matrix vector multiplication. Let us say this is a workload for us. And we want to execute this workload on a system where you have a CPU and 2 GPU devices coming from the same vendor. So you can set up a context with this multiple devices and set up OpenCL queues for each of these devices.

(Refer Slide Time: 02:20)

DAG Scheduling Example

- ▶ Create one context for the platform
- ▶ Create three devices each having its own command queue
- ▶ Enque write, ndRange and read for kernel K1 and K2 to two different devices(choose suitably) that will run concurrently
- ▶ Synchronise until both kernels finish execution
- ▶ Partition the data across the two GPU devices and launch two kernels K_{3_1} and K_{3_2} concurrently
- ▶ Enque write, ndRange and read for kernel K_{3_1} and K_{3_2} for both of these two devices
- ▶ Synchronise again and check the result

OpenCL Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, essentially, we will provide a host program, which will create the context for the platform, it will create 3d OpenCL devices, each having their own command queues. And then you can enqueue the write ndRange and read operations for kernel K 1 and K 2 on 2 different devices that will run concurrently. Now you have to choose that suitably maybe you have a programmer, understand that which kernel is more compute intensive, it has less divergence and all that and accordingly, you can make choices?

So you can map this K 1 and K 2, 2 of the devices and you synchronize until both the kernels finished execution. So essentially word in the host code once both of the kernels finished execution, then you may want to do just an example mapping were talking about that you may

want to map this data space, this input data space, which is C and E into 2 different GPU devices and launch 2 kernels K 3 1 and K 2 concurrently.

So what we are suggesting is, so here, you have the buffer C, full of data produced by K1 and the buffer E produced by data produced by K 2. So maybe we are again talking about a possible mapping. Let us say you have GPU device 1 and GPU device 2 you can copy a part of this here a part of this here, part of this here and then you can launch a kernel K 3 1 here.

You can launch a kernel K 3 2 here produced 2 outputs and then you copy to the host side memory copy these 2 host side memory to get the final output you may want to do that in because you have multiple search devices available and in case this data transfer, not a big over it with respect to the computational gain you may buy execution in parallel. If that is so, then you would like to do this operation .

So, this is some possible mapping that you can make and depending on the kernel compute characteristics it may have some advantage . So, in general we understand that that scheduling is a complex problem, because you have a architecture with multiple kinds of devices each kernel may have a map mean in some specific way to some device like kernel K 1 may is some specific execution time in a CPU it may give some different execution time on a GPU .

So, first as a programmer you need to identify what is a good device will do the mapping and then you should like to set up the command queues and launch the kernels respecting the dependencies. For example, here as we discussed K 1 and K 2 do not have any dependency, but they are the execution of K 3 depends on K 1 and K 2, both of them producing their outputs. So, K 3 , I need to synchronize before executing K 3 and then if required, if it is advantageous, then I may want to partition K 3 and execute.

So, as we can see that there can be many possible options like it will also make sense, let us say it all depends on the runtime scenario that if K 2 is producing something and K 2 is mapped to a device shall I map gets you to the same device, well, when we lay want to do that, so, I will want

to map K2 and K3 has to the same device in a situation because then this data transfer over it from K 2 to K 3 will not exist .

So, we have to figure out what is a good mapping will in case K 2 is idle or, the device work it has been mapped is idle, then that also makes sense . So, how to choose a mapping, what is the best mapping for a given application and an architecture is something that you have to fix. Here we are not discussing that problem. We are just trying to tell that will multiple mappings can be possible given an application and an article lecture as a programmer; you need to figure out what are the mapping characteristics.

What is a benefit, in which case and accordingly decide. So we are just trying to motivate that, if I map K 2 and K 3 together in a single device, it may be good. Because then I do not have this data over head for maybe when there is something else, maybe some other job is there, who for which I will like to use this device. So then I will actually like to use K 3 to , and map it to some other device. So it is all a situation which you have to fix based on the options and all that. So this was one example of that scheduling that we went through.

(Refer Slide Time: 07:47)

Heterogeneous Computing: Factors To Consider

- ▶ Scheduling overhead
- ▶ Location of data: data currently resident on which device
- ▶ Granularity of workloads: How to divide the problem
- ▶ Execution performance relative to other devices

OpenCL Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now so overall, these are the factors that we like to consider that what is the scheduling overhead. If we are scheduling multiple kernels in multiple devices, then we have to do some bookkeeping like we have to wait on events, we have to perform data transfers. So these are the overheads we

have to take care of and we have to say that will we will perform a scheduling decision only when the overhead, actually , the benefits outweigh the overheads.

So, the other issue would also be that the granularity of workloads that how much you will like to divide the problem, if you divide the problem into too many instances, then for each problem, the number of threads become too thin to sustain any overhead that is also added to the kernel launch event. So, this is important that whenever you are launching a kernel that has some overhead with respect to the runtime

Whenever you are actually doing a notify callback, or you are executing a kernel based on the execute code inclusion status of something else that also has a context switch over it. So, all the switching overheads are there. So if you define divide the problem into too many instances and try to solve it then the overheads will keep and we far outweigh the advantage of federalism that you may have.

So these things one has to figure out. And also you have to take care of whether the execution or what is the execution performance of the different devices. So, it is a heterogeneous compute platform, you may have multiple different GPUs. So a colonel executing in one GPU with a performance metric in a different GPU may have a different performance metric . So these are all 4 factors that want us to figure out.

(Refer Slide Time: 09:31)

The screenshot shows a presentation slide titled "Device Fission". The slide content includes a list of bullet points:

- ▶ The ability for some devices to be divided into smaller subdevices
- ▶ Device Fission is supported only on CPU-like devices
- ▶ It is possible to use Device Fission to build a portable and powerful threading application based on task parallelism

Below the list is a video player interface showing a person speaking. The video player has a progress bar at the bottom.

The other topic we like to cover is device fission like there are several CPU devices from different vendors, which support this concept of dividing the device in the OpenCL runtime system into smaller sub devices. For example, in general, if you have one CPU, OpenCL will identify it as a device but an enzyme discipline, , as CPU a model CPU with multiple assembly units, it can be logical partition to several sub devices. In that case OpenCL system can identify them as separate sub devices and manage them separately.

So, till that this idea of device fission is supported only on CPU like devices. And it is possible to use device vision to build a portable and powerful threading application based on task parallelism. We will like to think that if there are more such sub devices, then they can actually perform that overlapped execution to provide a more to explore task level parallelism and give us some optimized execution of our given assembly kernel.

(Refer Slide Time: 10:41)

Page 4 / 4

OpenCL Sub-Devices

We can create sub-devices partitioning an OpenCL device. The API used is `clCreateSubDevices`

- ▶ Creates an array of sub-devices; each referencing a non-intersecting set of compute units within the device, according to given partition scheme. Options:
 - ▶ `CL_DEVICE_PARTITION_EQUALLY`
 - ▶ `CL_DEVICE_PARTITION_BY_COUNTS`
 - ▶ `CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN`
- ▶ The output sub-devices may be used in every way that the parent device can be used, including creating contexts, building programs, further calls to `clCreateSubDevices` and creating command-queues.
- ▶ When a command-queue is created against a sub-device, the commands enqueued on the queue are executed only on the sub-device.



OpenCL:异构并行计算 Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So will not go in great detail about , executing a program in multiple sub devices, but rather let us just discuss how to create such as the prizes. So these are the options that are available till DAG in OpenCL runtime system, you can use this API functions to create sub devices, but how you want to create the device partition, there are several options. So, you can create an array of sub devices using this call, each of the sub device will refer non intersecting set of compute units.

So, suppose you have a set of compute units in each sub device, you will have a set of compute units which will not belong to the set of computers for some other device. And so, this partitioning of devices the internal SM the units of the CPU across sub devices can be has to be managed and you can give as a programmer you can give different directives that how to manage them. So, you can tell the runtime system that you will you use if you use this flag CL DEVICE PARTITION EQUALLY and then it will perform the allocation of sub computers to sub devices following this flag.

You can give it a by counts flag and also there is this option of affinity domain . So, there are different possibilities in which you can actually partition the sub device space. Once the sub devices are created, they can be they are very much like normal OpenCL device. For them, you can create contexts, you can build programs you can launch programs into command queue specified for sub devices and all that.

So, these options you can just look into, we will just give a program example with 1 of the options, but of course, these are quite intuitive. So and for a specific sub device, you can create a separate command queue and you if you launch a command, it will get launched not into the entire device but into that sub device.

(Refer Slide Time: 12:47)

Creating OpenCL Sub-Devices Example

Creates an array of sub-devices, each referencing a non-intersecting set of compute units within given CPU device.

```
#define NUM_CUS 8
...
cl_device_partition_property subDeviceProperties[] = {
    CL_DEVICE_PARTITION_EQUALLY, NumOfCUPerSubdevice, 0 };
clCreateSubDevices(device_id, subDeviceProperties, NUM_CUS/NumOfCUPerSubdevice,&
    subDevices,NULL);
//cl_int clCreateSubDevices(cl_device_id in_device, const
//    cl_device_partition_property *properties, cl_uint num_devices,
//    cl_device_id *out_devices, cl_uint *num_devices_ret )
//Other partition properties are CL_DEVICE_PARTITION_BY_COUNTS and
//CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN
...
cl_command_queue commands[NUM_CUS/NumOfCUPerSubdevice];
for (int i = 0; i < NUM_CUS/NumOfCUPerSubdevice; i++)
    commands[i] = clCreateCommandQueue(context, subDevices[i], 0, &err)
...
```

OpenCL Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So let us have a look into this simple program. So, first thing, you have to set up a device property array using which you are trying to specify how the sub devices are defined . So, for this you have to launch you have to define an array let us say we give it the name sub device properties is type has to be CL device partition property, and you have to give the up to populate this area with the following elements. The first component has to be the primitive like what is your choice of partitioning scheme.

So let us say I choose CL device partition equally distribute the sub devices equally and you have to specify; what is the number of sub devices you want. So obviously, the first thing is CL device partition equally. So whatever is the number of units available, you should do at saying that will you divide them equally across the number of sub devices and the number of sub devices you want is specified by this flag.

So once this array set up, you just make a call CL creates sub devices, you give it the device ID and you pass your requirements to this array and you give this thing that is what is the total number

of compute units you have and divided by the number of compute units to one per sub device and this call will actually provide back the handles to each of the sub devices inside this array sub devices.

So, once this is done next you let us look into how you can actually create queues and engage the different sub devices. So, and also we like to say that here we are using the equal equally flag instead of that I can actually give explicit counts or I can specify an affinity domain . So, coming back to the command queues. So this is the array in which I am trying to define command queues for each of the devices.

So in the fall loop as you can see, we are firing the CL create command queue command and the number of times this loop is going to rotate is exactly equal to the number of sub devices, which is the total number of compute device a compute units divided by number of compute units 1 per sub device. So that is the number of sub devices you have created. And for each of them, you are creating a separate command queue .

(Refer Slide Time: 15:23)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title 'Concurrency and CUDA' is centered at the top. Below the title is a list of bullet points. At the bottom, there is footer information including the slide number 'Page 4 / 4', the name 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur', and the IIT Kharagpur logo.

- ▶ Applications must execute functions concurrently on multiple processors so that available processors in the system can be efficiently used for heterogeneous computing
- ▶ CUDA Applications manage concurrency by executing asynchronous commands in streams, sequences of commands that execute in order
- ▶ Different streams may execute their commands concurrently or out of order with respect to each other.

Now so, that is how we actually create the set of sub devices and you create separate command us for each of the sub devices. Now, what is the advantage of this like we have been discussing? Well, now you can issue commands in parallel across the different command queues for the sub devices,

and that will help you to achieve overlapped execution in case you want 1 sub devices to do something you want some other subdivides to do some other thing.

Let us say you want sub devices 1 to some data copy operation while sub device 2 you actually engaging to some executing some other kernel. So, like once the sub devices are created, they are really going to act like normal full-fledged OpenCL devices to the programmer is point of view . So with this discussion about device fission and properties of sub devices, we like to end this lecture. Thank you.

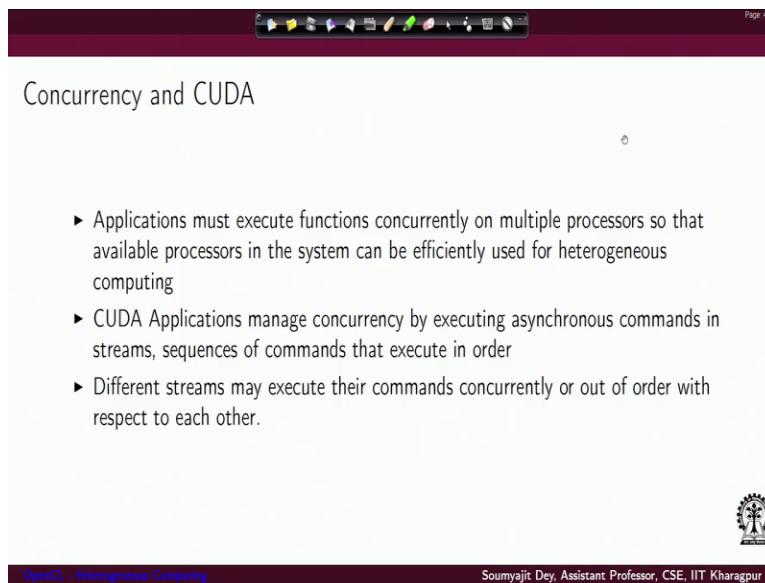
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 52
OpenCL – Heterogeneous Computing (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming. So, in the last lecture we have discussed how to create open CL sub devices how to perform DAC scheduling on architecture comprising multiple devices which support open CL and all that now, in this context while we have been discussing how I can execute kernels in parallel, or how I can break down a kernel and partition it and launch into multiple devices.

We like to go back to our original programming language which was CUDA and check whether in CUDA we have any support for such concurrent execution.

(Refer Slide Time: 01:05)



The screenshot shows a presentation slide with a dark header bar containing navigation icons. The main title is "Concurrency and CUDA". Below the title is a bullet-point list:

- ▶ Applications must execute functions concurrently on multiple processors so that available processors in the system can be efficiently used for heterogeneous computing
- ▶ CUDA Applications manage concurrency by executing asynchronous commands in streams, sequences of commands that execute in order
- ▶ Different streams may execute their commands concurrently or out of order with respect to each other.

In the bottom right corner of the slide area, there is a small circular logo of the Indian Institute of Technology Kharagpur. At the very bottom of the slide, there is a thin footer bar with the text "OpenCL - Heterogeneous Computing" on the left and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur" on the right.

If you remember, so, by the way, this is the point where we have we can officially say that we are done with our, discussions on open CL and heterogeneous computing using open CL. We are kind of going back to CUDA programming language and use usage for this purpose and also in the future lectures will be using CUDA for other purposes as I have been provided in the indexes. So, the issue is, if you look at our earlier lectures on CUDA programs.

CUDA runtime system and everything, our programming model has been used to launch a host code, the host code execute sequentially and it can launch a CUDA kernel and then it will wait till the kernel returns and then you can fire and the host code can go beyond and then it can launch another CUDA command. So although the host code can launch the CUDA kernel and go beyond to the next command to be executed for the GPU.

But the next command it will mean execute only when the kernels corners execution is done. So there is sequentiality in that way. But can I have a CUDA applications sharing, a GPU and walking together, because as we see, the issue is, if I look at a GPU is like an accelerator, it does not have any scheduling primitive from the software side is a hardware scheduler where is device. So, you cannot launch the way we have defined till now, we never had the facility that we will be launching multiple kernels together on a CPU on a GPU device at the same time to execute or something else.

So the issue is, can I have support for concurrent execution CUDA? The answer is yes. In currently available CUDA runtime systems, you have such supports. So and that is also desirable because in then you can also extract the maximum parallelism that you have ever in the underlying hardware. So, the way CUDA manages concurrency is by executing a synchronous commands in streams or CUDA streams and in each stream can execute a sequence of commands that are that have been that have been dispatched to the stream in that order.

So in CUDA the way we have been discussing till now we have when we are talking about a CUDA program, there is something called a default CUDA stream that stream is 0. In general, I can write a program which is engaging this CUDA, GPU by running multiple CUDA streams in parallel. So the runtime system will use this obstruction of streams to execute CUDA commands in parallel, while each of the streams while the host program will just issue commands as synchronously to each stream.

So we will see that why we call it synchronize and all that so, different streams where we are so, the host program as we are saying, instead of writing a normal CUDA program, which is a single stream of execution, the host program can be find multiple streams. And it can be it can dispatch

commands for the runtime system individually to each of the streams. And each of the streams will be executing those read/write or kernel commands.

In the order they have been dispatched, but in what order they do it among streams is completely defined by the runtime system. So they will actually execute out of order with respect to each other that is at the stream level.

(Refer Slide Time: 04:46)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title 'CUDA Streams' is centered above a list of bullet points. On the right side of the slide, there is a portrait of a man, Soumyajit Dey, wearing a light blue hoodie. At the bottom of the slide, there is a footer bar with the text 'OpenCL Heterogeneous Computing' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right. The slide content is as follows:

- ▶ A CUDA stream refers to a sequence of asynchronous CUDA operations that execute on a device in the order issued by the host code.
- ▶ These operations can include host-device data transfer, kernel launches, and most other commands that are issued by the host but handled by the device.
- ▶ The execution of an operation in a stream is always asynchronous with respect to the host.
- ▶ It is the programmer's responsibility to use CUDA APIs to ensure an asynchronous operation has completed before using the result.

So what is a CUDA stream? So it refers to a sequence of synchronous CUDA operations that execute in a device in the order issued by the host code. So suppose the host code is issuing a set of orders to put a stream 1 those will be executed by that stream in that order. The host code is also issues some other commands to a separate stream that stream will execute whatever commands has been issued to that stream in that order.

But as I was saying, in what order these streams interleave is dependent on the runtime and there is no finite control on that so, the typically when I talk about CUDA commands, we are thinking of basic operations like host to divide your transfer host and device or device to host your transfers, kernel launches and synchronization commands that may be issued by the host but they are handled by the device, those kinds of commands, and the execution of operations inside a stream that is synchronous with respect to the host.

Now, this is the key point. So the host program just dispatches commands, 2 different streams and moves forward. So this commands execute in a CUDA stream setup for the GPU device completely independent of what the host is doing the host has dispatched commands is very much like open CL command queue, you have open CL host program moving forward dispatching commands to command queues, you can have the CUDA program moving forward and dispatching operations to the stream.

So, once they have been dispatched, the host program does not have a control, it is just moving forward as synchronously. So, now it becomes the job of the programmers to actually use the CUDA APIs to ensure that these asynchronous operations are can preserve dependencies. So here also like open CL, we have been discussing here also will see the role of events through which synchronization can be attained across streams and all that.

(Refer Slide Time: 06:47)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'CUDA Streams'. The content discusses the execution of CUDA operations in streams, mentioning two types: implicitly declared (NULL stream) and explicitly declared (non-NULL stream). It notes that the NULL stream is the default. At the bottom, there is a photo of a man and text identifying him as Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur.

All CUDA operations (both kernels and data transfers) either explicitly or implicitly run in a stream. There are two types of streams:

- ▶ Implicitly declared stream (NULL stream)
- ▶ Explicitly declared stream (non-NULL stream)

The NULL stream is the default stream that kernel launches and data transfers use if you do not explicitly specify a stream. All CUDA examples discussed previously used the NULL or default stream.

OpenCL - Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So by speaking of CUDA streams, all CUDA operations, both kernels and data transfers, they either implicitly or explicitly running a stream. So when we do not specify the stream is that default, now stream. So these are the 2 defined 2 different types of streams the implicitly declared stream or the NULL stream. And the explicitly declared streams, which are the non-NULL stream. So, by default everything is stream 0. And otherwise, if you want to create more streams for attaining concurrent execution, you have to define them exclusively.

So the NULL stream is the default stream where kernels launch and data transfers, use in specify it and whatever, like we have been saying that whatever good examples we specified earlier, they were all by default getting launched into the string.

(Refer Slide Time: 07:36)

The screenshot shows a presentation slide with a dark header bar containing icons. The main content area has a title 'Asynchronous API' and a code block. The code is written in C and demonstrates basic stream operations, asynchronous memory transfers, and kernel launching. A video player window is overlaid on the slide, showing a man in a purple shirt speaking. The video player has a progress bar at the bottom. At the bottom of the slide, there is a footer bar with the text 'OpenCL Heterogeneous Computing' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right.

```
// Basic stream operations
cudaStream_t stream;
cudaError_t cudaStreamCreate(cudaStream_t stream);
cudaError_t cudaStreamDestroy(cudaStream_t stream);
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
//Blocks host until stream has completed all operations.
cudaError_t cudaStreamQuery(cudaStream_t stream);
// Queries an asynchronous stream for completion status.

// Asynchronous memory operations
cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count,
                           cudaMemcpyKind kind, cudaStream_t stream = 0);
// cudaMemcpyKind is an enum type with values: cudaMemcpyHostToHost,
// cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceTo

//Pinning memory on the host is required for asynchronous data transfer
cudaError_t cudaMallocHost(void **ptr, size_t size);
// Launching Kernel
kernel_name<<<grid, block, sharedMemSize, stream>>>(argument 1);
```

So, let us get our first test of this asynchronous API. So, first we are just going to give it we are just trying to give some basic examples of stream operations like so the type is CUDA stream underscore t. So you can define a stream you can declare a stream using this command, the first one who CUDA stream underscore t then stream. So this is a stream that you have defined. And then after declaring it you have to create the stream using a command like CUDA stream create.

If you want to destroy an already created stream, you have to fire the command CUDA stream destroyed. And if you want a stream to synchronize, and we will see that first is required, you have to use this command CUDA stream synchronize. Now so when you want to synchronize, you actually blocked the host until all the streams has completed operations. So we will see the different possible synchronizations. Maybe this is a good time to look at that, like just a minute.

(Refer Slide Time: 08:46)

The slide has a dark header bar with various icons. The title 'Synchronization' is centered at the top. Below the title is a bulleted list:

- ▶ Synchronizing the device `cudaDeviceSynchronize()` : wait until all streams finish
- ▶ Synchronizing a stream `cudaStreamSynchronize(stream)`
- ▶ Synchronizing an event in a stream `cudaStreamWaitEvent(stream, event)`
- ▶ Synchronizing across streams using an event `cudaEventSynchronize(event)`

At the bottom left is the OpenCourseWare logo, and at the bottom right is the name 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

So these are the different ways you have synchronization on the stream. Now we are just saying that let us say we just had this command CUDA stream synchronize, then you are asking that wait if this is they are in the host program in the program. We will wait here until all the streams finish. If this is there in the host program CUDA streams, sorry, the first one is CUDA device synchronous. So if you are firing CUDA device synchronize in the host program.

You want all that streams that are running in the CUDA device to finish, and then only the host program move beyond this point. If you want a specific stream to synchronize, then you just write CUDA streams synchronize with the stream that means the host program does not move forward until this specific stream will finish the other ones will see soon. So now, let us go back to our example that we have started. So this is CUDA streams synchronize.

So you have blocked the host program until unless the stream finishes, whatever operation it has been given here, we have not given to anything till now. So the other thing that you can do is you can query a stream. So, this gives you an error status. some status can say whether it is completed either executions were there, it is still in process and all that so, an important thing that we have to remember is when you want to perform data transfer to the CUDA streams.

They have to be a synchronous. Why? Because unlike a normal CUDA program, if you are using CUDA streams, then your host program is moving forward just by dispatching commands. So for

that the memory copy operation should be done in such a way that it is complete is it has to run in a separate thread with respect to synchronously with respect to the real host programs, thread of execution.

So for that, unlike normal CUDA mem copy, you have to use this mem copy command which is CUDA mem copy async. Because now you really want a synchronous memory operation that you do not wait at this point here until the mem copy is done. You just issue the command mem copy to a specific stream. Let us say in this case you stream is 0 and you just move forward. So there is a difference between a normal CUDA mem copy and mem copy assigned.

So, this means that transfer a synchronous so that the host program can move at its own speed and this transfer can happen at own speed as is supported by the device. Now in this command as you can see where you have a source, you have a destination you specify in which stream to copy, you give the size and you also have a flag CUDA mem copy the kind, which is an enumerated type. It has the possible values like we all know that earlier. these are also similar to the flags we have been using for CUDA mem copy normal mem copy.

So you can either do mem copy host to host does not make sense maybe and then you can do host to device, device to host and if you want to do mem copy across multiple devices, then device to device. Now there is another important thing when you are doing this work on across streams. And then the requirement would be that you want whatever memory has been allocated on the host side to be pinned. So let us understand what we say here like.

Let us say you have a host side memory and a buffer has been declared and you want to copy it to a device side memory. So, this is the host memory buffer and you were copying it to a device. Earlier it used to be a single 1 short copy command. But now, in the current form, since things are a synchronous, so the copy you are actually living it to the copying us from the device and the runtime system to manage a synchronously so the copy now can happen as and when the runtime system wants to do it.

While you were moving forward with the host program and the copy operations are happening as synchronously. The point is they may not happen in 1 shot, it is not a blocking operation. So, then thinking from the operating systems point of view, it may happen that the copy operation started some amount of data has been copied, and then when you want to restart by that time, the page in those squares host side is memory, where this buffer was sitting, it has been mapped out.

So, that will create unnecessary delay and it will be an issue. So, what you want is if you are allocating memory on the host side, this is the host side memory from which you are going to copy in synchronous streams. You want this memory to be what we call us pinned. You want this memory to pinned on the device so that you are essentially we are asking that, you do not pay it out because I am going to copy as and when I want in a lazy fashion.

So for ensuring that the memory is pinned on the host side, unlike normal CUDA_malloc, now you use the command CUDA_malloc_host. So, there are differences you can see CUDA mem copy async, CUDA malloc host. So, you use CUDA malloc host to have a pinned data offering the host side memory, and then you launch the kernel to a specific stream. So, this is a new thing earlier they have been launching kernels. So, you give a grid and block then as you can see, there can be a third parameter that will how much amount of shared memory in the GPUs will be allocated for this specific kernel.

That can also be specified by this shared memory size. Why will you are having multiple streams it is expected you can launch multiple kernels across streams. So you may want to specify well which stream working with which kernel should use how much amount of shared memory, so that is where you can specify the shared memory size result for this kernel. And in which stream, do you want the kernel to be launched?

(Refer Slide Time: 15:18)

The screenshot shows a presentation slide with the title "CUDA Streams: Basic Example". The slide contains the following C code:

```
__global__ void kernel(float *g_data, float value)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    g_data[idx] = g_data[idx] + value;
}
#define CHECK(call)
{
    const cudaError_t error = call;
    if (error != cudaSuccess)
    {
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);
        fprintf(stderr, "code: %d, reason: %s\n", error,
                cudaGetErrorString(error));
    }
}
```

At the bottom of the slide, there is a footer bar with the text "OpenCL High Performance Computing" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So this is how you can actually copy, you could declare host side data, data buffer, and then you can copy that so this is not just a little reminder, this is not a sequence of program statements. We have just written these different directives to let you understand what are the different specific operations that the synchronous API of CUDA is actually allowing you to do? So here is how you launch the kernel.

These are you launch the kernel to the host in a pinned fashion these are you perform a synchronous transfer. These are you actually synchronized on a specific stream and is how you create or destroy the stream. And you can also query a stream status. Coming to basic CUDA stream programming examples. So let us say you have a normal kernel. So all you are doing is you calculate the global thread id. And you just add that with a constant value, the simple kernel.

So what we will do is we will also define a check function here, so that whatever function call will make next in whatever programs we show you, the status that the function called returns can be, is actually handled nicely through this check function. So we are actually doing this so that we can avoid writing all these handler code everywhere. So for every function call, we can just give this call to check so that whatever type will be emitted. If in case it is not a success, it is gracefully handled with suitable file prints to their era standard.

(Refer Slide Time: 16:56)

```

int main(int argc, char *argv[])
{
    int devID = 0;
    cudaDeviceProp deviceProps;
    CHECK(cudaGetDeviceProperties(&deviceProps, devID));
    printf("> %s running on", argv[0]);
    printf(" CUDA device [%s]\n", deviceProps.name);
    int num = 1 << 24;
    int nbytes = num * sizeof(int);
    float value = 10.0f;
    // allocate host memory
    float *h_a = 0;
    CHECK(cudaMallocHost((void **)&h_a, nbytes));
    memset(h_a, 0, nbytes);
    // allocate device memory
    float *d_a = 0;
    CHECK(cudaMalloc((void **)&d_a, nbytes));
    CHECK(cudaMemset(d_a, 255, nbytes));
}

```

OpenCL High-Performance Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So now let us look into the host programs example for CUDA streams. So, inside your mean, you first declared this device properties, you have a variable for device properties. And you can check whether for this device properties for some specific CUDA device id, what are the device properties, and then it will be able to say that will what is running in that CUDA device and all that.

(Refer Slide Time: 17:35)

```

> ./simpleMultiAddBreadth Starting...
> Using Device 0: Tesla K40m
> Compute Capability 3.5 hardware with 15 multi-processors
> CUDA_DEVICE_MAX_CONNECTIONS = 1
> with streams = 4
> vector size = 262144
> grid (2048, 1) block (128, 1)

Measured timings (throughput):
Memcpy host to device : 0.383424 ms (2.734769 GB/s)
Memcpy device to host : 0.182272 ms (5.752809 GB/s)
Kernel : 1605.997192 ms (0.001306 GB/s)
Total : 1606.562866 ms (0.001305 GB/s)

Actual results from overlapped data transfers:
overlap with 4 streams : 402.014435 ms (0.005217 GB/s)
speedup : 74.976738

```

OpenCL High-Performance Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if you look at an example of the prints we have here, so, is just going to say that will, and this is the program that we are running, and it is running in that this specific device with disease capabilities and all that so, I believe, earlier we have already talked, about what our device properties what are the different CUDA device properties and all that essentially the information

of the runtime system So, once I make this call using this, I am actually extracting from the runtime system all the device properties and storing them into this.

And for the specific for some specific GPU device id in this case we are said they were just waiting for the ideal GPU device with id is 0, it can be whatever you want? So and then we have some standard prints just to tell what is the program name that is running because it is again, that is that is what should be the executable name in argv is 0 location. And you also print what is the name of the CUDA device what is the GPU device on which you are running the program.

So that would be in the device properties structure itself in the field device properties. So these just normal bookkeeping code like we write for you have the device property structure, when you define an order type for that you define a device property structure and you extract the device properties for the target device, you print all those status effects. And then let us get into the normal action. So, let us say you start with elevating host side memory using as we have discussed now, we will be using cooler CUDA malloc host command to get in host memory.

And then you can use normal CUDA malloc. For the device and memory why use normal CUDA malloc? Well, then in the in the device side memory, you do not have an issue with paging out of memory and all that so, as you can see that we are making all those function calls and under they got of the check statements, so that any kind of error is was fully handled. So once this is done, let us decide on some launch configuration of the kernel. So, you define a set block of size 512, you define a grid with this some specific number of blocks like this.

(Refer Slide Time: 20:15)

```

// set kernel launch configuration
dim3 block = dim3(512);
dim3 grid = dim3((num + block.x - 1) / block.x);
// create cuda event handles
cudaEvent_t stop;
CHECK(cudaEventCreate(&stop));
// asynchronously issue work to the GPU (all to stream 0)
CHECK(cudaMemcpyAsync(d_a, h_a, nbytes, cudaMemcpyHostToDevice));
kernel<<<grid, block>>>(d_a, value);
CHECK(cudaMemcpyAsync(h_a, d_a, nbytes, cudaMemcpyDeviceToHost));
CHECK(cudaEventRecord(stop));
// have CPU do some work while waiting for stage 1 to finish
unsigned long int counter = 0;
while (cudaEventQuery(stop) == cudaErrorNotReady) {
    counter++;
}

```

So this is my definition of the grid? what is the number of blocks and all that so once I have defined what is the total number of blocks? That would be in the grid, and what is the thread block size? Then let us use the CUDA API event to perform some bookkeeping of the execution times. So I believe in our earlier lectures, we have already discussed of the CUDA event API and how to use CUDA events to sample timings.

So at this point, you use the CUDA event create command to sample the current time in and storage in the event, object type event create and stop. And then let us say we get into the action with the different CUDA streams. So we will just issue as work as synchronously to the GPU, which is said to me, for GPU stream 0. So what we are doing is we are just using the CUDA mem copy as in commands to launch a copy of version from the host side array hA to the device side array.

Using CUDA mem copy host to device directive. And as you can see, that is an asynchronous copy. So we will just immediately go forward to launching the kernel. And since no stream is explicitly specified is always happening in stream 0. And then once that is done, you expect the result to be in the in the device side memory. So you get back that data from the device side to the host side memory, and then you again record you perform CUDA event record again on the stop event.

So, like, so, here, you create you actually declared the CUDA event object stop and you actually created you use this API function could even create to define and then here you actually sample it so, from this point, the event is ready and it has started sampling the timings. And you are actually measuring the time that has been sampled here in using this CUDA event record. Now after this, you want the CPU to do some work while waiting for stage one to finish that you are you are waiting for the stream to complete.

So what do we is you can keep on this job. And whether it is at a state that is, it is not ready, whether it is returning or not ready flag. And for the time being, you are just making the CPU some do some simple action like, like implementing a variable, because as you can see that all the work of the memory copy is going on a synchronously to the GPU device and coming back is just there to the host side here.

(Refer Slide Time: 23:28)

CUDA Streams: Basic Example

```
// print the cpu and gpu times
printf("CPU executed %lu iterations while waiting for GPU to finish\n",
counter);
// release resources
CHECK(cudaEventDestroy(stop));
CHECK(cudaFreeHost(h_a));
CHECK(cudaFree(d_a));
CHECK(cudaDeviceReset());
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So once we know that, well, from this event record, we get to know that, things are done, you move forward. So from the CUDA event enquiry, you get the stream status, that, these commands have finished their execution, then you actually print the CPU and GPU times. So you get out of this loop when you are actually querying this event and you are getting that the good it is not really good era not ready for this, this flag is no more true.

Then you just print what is the execution time of the CPU? what is the number of iterations that the CPU spent waiting for the GPU to finish, because that status will actually change when this all the commands that have been synchronously issued to the GPU, they are all completed. So this is us telling you how many iterations of the loop happen for the time in which the GPU got it has synchronously executed the kernel.

And then copied back to the last mem copy from device to side. So once everything is done, you just release the resources through a CUDA event destroy because now this event tag is no more required, the CUDA event variable is no more required. And of course, you can free the host side as well as a device that memory and perform CUDA events device reset. So these are a basic example all we are doing is we are doing the normal kernel operations. But just using a synchronous memory copy and a synchronous execution of the kernel.

(Refer Slide Time: 25:10)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Exploiting Concurrency'. Below the title, there is a text block and a bulleted list. On the right side of the slide, there is a small video window showing a man with glasses and a light-colored hoodie. At the bottom of the slide, there is a footer bar with text and logos.

Asynchronous, stream-based kernel launches and data transfers enable the following types of coarse-grain concurrency:

- ▶ Overlapped host computation and device computation
- ▶ Overlapped host computation and host-device data transfer
- ▶ Overlapped host-device data transfer and device computation
- ▶ Concurrent device computation

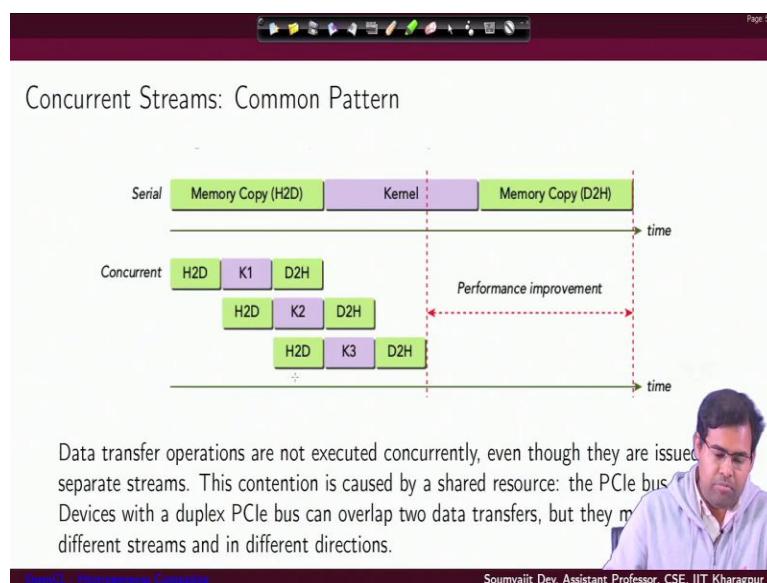
OpenCL: Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

But yet, we have not really exploited the concurrency that is available on the GPU side device by using multiple streams, it was just about exploiting this notion of asynchronous dispatch. So now, we will move forward with that and see how using multiple streams, we can have overlapped computation of host and device overlap computation in the host and also a data transfer happening between host and device.

So these are the different possibilities where I can have some action happening both in the host and the device. And in that way, I have overlapped executions with a total execution time reduces? So that is what we want to do. Now this is a listing like what are the things that are possible in parallel host is computing something while and the device also computing something host is computing something and you have some transfer of data from the host to device.

You have similar transfers and also the devices computing something or you have multiple streams that are executing in the device. So, now, the mean these are the possibilities that you have.

(Refer Slide Time: 26:20)



Now, the typically common pattern that we have with respect to concurrent execution, the advantage of concurrent execution would be like this for example, consider a normal execution of a GPU program or a CUDA kernel here so, you have a memory copy from host to device that will take some time is followed by the execution time of the kernel. And then you have time for memory copy from that device to the host side.

Now, suppose you break down these memory transfers to small chunks, and you create, different kernels. For kernel versions, I would say for an operating on these chunks. And then you launch them in multiple streams. So, as you can see that these are pipeline execution. So now, while kernel is executing for this first chunk of data that has been copied in parallel, you can actually have the GPU getting the data for the second instance of the data segment.

So, this small host to device computation is done, then the kernel is working on it in one stream. And in the other stream, you can have the risk the second segment of data chunk coming in from the host side, well in hardware, this should definitely be possible. So you need support like in your GPU side, you have a copy engine, which is copying the data from the host. And you have execution units, which are engaging the different threads and processing the data which have been already copied.

So in terms of hardware, this will be possible. That is why you have you can have this overlapped execution, since you have overlap this executions of copy and computation overall as you can see, you have a performance improvement and this is what we want to attain by launching things in parallel across trims. So, in general, if we assume that we have a PCI Express Bus through which data copy is happening, and of course, we cannot support multiple copies in a in a PCI Express Bus.

Then data transfers cannot execute concurrently. But they although they are issued in separate streams, but the data transfers can overlap with execution of the kernel as we can see in this example, however, there are devices with duplex PCI express buses, where I can have a support for 2 parallel transfers. And in case they are in different streams and in different directions they can have been in parallel.

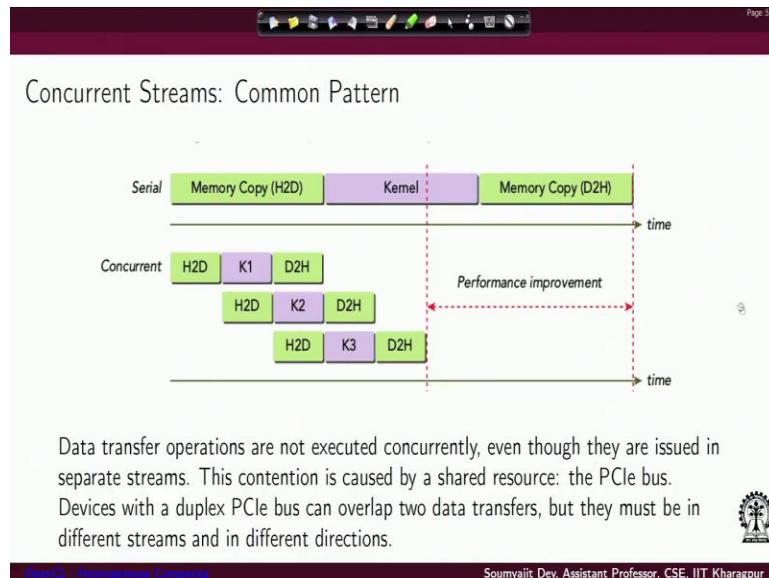
So, let us say in stream 1, I have a H2D type copy and in stream 2 I have a H2D type copy that is considered absolutely fine, so with this now maybe we will stop here, and we will see some program examples like how this concurrent streams are useful in the next lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 53
OpenCL – Heterogeneous Computing (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming. So, in the last lecture we have been discussing about CUDA streams and how overlapped execution can be performed across streams.

(Refer Slide Time: 00:36)



And in that regard, we have set up a nice example here of like how it is better to say overlapping the execution and data transfer. And we also mentioned that in case we have a device where we have a support for duplex PCI express bus then I can overlap today transfers happening in different devices and happening in different directions. So let us assume that we have a normal PCI express bus as well as is the case here so, we can overlap execution in the GPU device along with 1 sided a transfer.

(Refer Slide Time: 01:12)

The screenshot shows a presentation slide with a dark header bar containing various icons. The main title 'Synchronization' is centered at the top. Below the title, there is a list of four bullet points describing CUDA synchronization primitives. At the bottom of the slide, there is footer information including the OpenCourseWare logo, the name 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur', and the IIT Kharagpur logo.

- ▶ Synchronizing the device `cudaDeviceSynchronize()` : wait until all streams finish
- ▶ Synchronizing a stream `cudaStreamSynchronize(stream)`
- ▶ Synchronizing an event in a stream `cudaStreamWaitEvent(stream, event)`
- ▶ Synchronizing across streams using an event `cudaEventSynchronize(event)`

Now, if you are really engaging different streams to perform concurrent execution you may also require sometimes to synchronize the executions because in general streams are asynchronous. So for that CUDA will provide you several synchronization primitives. In case you want the host program to wait till all the CUDA devices all the streams executing on the CUDA device to finish, you can use the first option device synchronize.

If you want a single stream to you if you want the host program to wait a point where you want the single stream to synchronize you execute CUDA streams synchronize with the option that which is the which specific stream you want to synchronize at that point. If you want synchronized an event in a stream on a specific event in a stream, then you have to give this command CUDA stream wait event.

So you are saying that will this stream should wait for this event to happen. So there is again, anyone options, you want all the streams to synchronize at some point. So that is good our device synchronize some at some point in the host problem. If you want to synchronize a specific stream, you use CUDA stream synchronize. If you want CUDA stream wait event, you want that a stream will synchronize in an in a specific event to a specific event, then you use CUDA stream wait event.

And if you want synchronizing across streams using an event, then of course, you do not need to specify the stream you just specify the event. And you have the command CUDA event synchronize just with the event argument. So you synchronize all the streams with this event.

(Refer Slide Time: 02:58)

```
int main(int argc, char **argv)
{
    printf("> %s Starting...\n", argv[0]);
    int dev = 0; cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("> Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));
    if(deviceProp.major < 3||(deviceProp.major==3&&deviceProp.minor<5)){
        if (deviceProp.concurrentKernels == 0)
            printf("> Concurrent Execution not supported. CUDA kernel runs will be
                   serialized\n");
        else
            printf("> GPU does not support HyperQ. CUDA kernel runs will have limited
                   concurrency\n");
    }
    printf("> Compute Capability %d.%d hardware with %d multi-processors\n",
           deviceProp.major, deviceProp.minor, deviceProp.multiProcessorCount);
}
```

So now coming to our concurrency example. So we will try to see how overlap execution can be attained. So let us say you have a main were you in then this part of the code is again, quite simple, you get the device properties, you are saying that in which device what is executing and all that and then you can perform. This is a piece of code that is necessary, to figure out whether you have support for concurrent execution or not. So that is actually from the device property structure, you can figure out the CUDA runtime systems major and minor number.

And using these numbers, you can figure out how much is the execution support, so in case the measure is less than 3, and or the device measure is equal to 3 and less than 5? Well, in that case, in cases less than 3, you do not have any support for concurrent execution. So CUDA kernels will be serialized now and also there is this issue with major and minor values. So based on this choice, what is the major and minor value of the CUDA runtime system?

There would be your device property will contain, whether the value 0 or 1 for the concurrent kernels flag, and that would tell you whether there is support or not. And also, there are 2 possible checks here, this is what I am trying to say that you can have concurrent execution fully, or you

can have limited concurrent execution. in the other case, which is this else, like you satisfy the major and minor requirements like let us say you have major 3 or minor less than 5.

Then in case your CUDA kernels is that mean for those devices, where the major version is earlier to the earlier 2, 3, or it is equal to 3 but the minor version is the minor of number is earlier to 5. For those devices, if you have CUDA kernels command state to 1, for the earlier devices, the thing is, you do not have a support for something called hypercube. So you will have concurrency support, but not fully, we will see what is that and in case you are not getting inside this chip.

That is that means your CUDA runtime systems is saying that will for the GPU device we are talking about the major number is greater than 3 or the major number is equal to 3 but the minor is greater than 5. For that, you have compute capability and which is suitable for concurrent execution. So it will give you your code here, the example we are talking about here, we will just say that, you have concurrent execution capability, and it will also specify what exactly is the major and minor value

And what is so that would that would be a kind of indicator of what is the amount of compute capability supported. Now, something you have interesting here that what is this Hyper Queue? So, there are earlier GPU devices where you have support for concurrent execution, but there is limited. So, let us try and understand what is this limitation. So, consider the situation that will you have got streams, and you have got multiple kernels, which are executing on the streams.

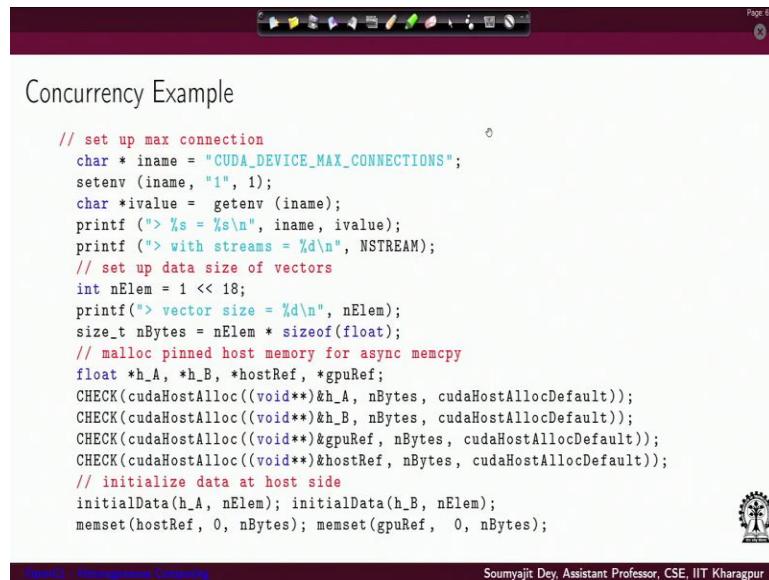
So you were you want to symbol execute a task graph where you have the kernels A, B and C queued up. Let is call their instances the first second. First all the first instances of A, B and C, you want a to finish then be to work and see what to work on the data that we will be produce things like that maybe you have other instances of these kernels sequenced in some other stream and you have the same corners third instances sequence to some other stream.

The issue is in CUDA older runtime systems, this streams will finally get into a job queue where things are pretty much serialized so you may have I am just trying an option here A1 followed by B1 followed by C1 followed by A2 followed by B2 followed by C2 so on so forth. And then the

system is trying to identify what is the parallel is in every level, it does not see any parallelism here because of these dependencies, does not see parallelism here because the dependencies, it only figures that these are the 2 that can be really executed in parallel.

Whereas we can see that will A1, A2 A3 can be executed in parallel B1, B2, B3 can be executed in parallel and C1, C2, C3 can be executed in parallel. So, this issue with me in the final hardware, queue has been resolved with the architectural support of some structure called hypercube, where through which it is actually able to see the dependency structure and see what is the full scope of concurrent execution? That is possible across the different streams. So, we are we are not going to more architectural details, you can read NVIDIA documents that are widely available.

(Refer Slide Time: 08:23)



The screenshot shows a presentation slide titled "Concurrency Example". The slide contains a block of CUDA C code. The code sets up maximum connections, initializes vectors, allocates pinned host memory for asynchronous memcpy, and initializes host side data. The code uses printf statements to print configuration values and stream counts. At the bottom of the slide, there is a footer bar with the text "Parallel Programming Computing" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

```
// set up max connection
char *iname = "CUDA_DEVICE_MAX_CONNECTIONS";
setenv (iname, "1", 1);
char *ivalue = getenv (iname);
printf ("> %s = %s\n", iname, ivalue);
printf ("> with streams = %d\n", NSTREAM);
// set up data size of vectors
int nElem = 1 << 18;
printf ("> vector size = %d\n", nElem);
size_t nbytes = nElem * sizeof(float);
// malloc pinned host memory for async memcpy
float *h_A, *h_B, *hostRef, *gpuRef;
CHECK(cudaHostAlloc((void**)&h_A, nbytes, cudaHostAllocDefault));
CHECK(cudaHostAlloc((void**)&h_B, nbytes, cudaHostAllocDefault));
CHECK(cudaHostAlloc((void**)&gpuRef, nbytes, cudaHostAllocDefault));
CHECK(cudaHostAlloc((void**)&hostRef, nbytes, cudaHostAllocDefault));
// initialize data at host side
initialData(h_A, nElem); initialData(h_B, nElem);
memset(hostRef, 0, nbytes); memset(gpuRef, 0, nbytes);
```

So, for this example, let us, first see that how things can be how we can have overlap execution. So, what will first do is we will figure out what is the maximum number of streams that we can have so, we set up this maximum number of streams, because we will be declaring that number of streams and all that so, let us say we first initialize the host side that is on which to work on so, let us say initialize the host side which are getting locked and pinched to the host side memories for that we have support for a synchronous memory copy. And also we perform allocation for this GPU difference and host reference areas here, and then we initialize all these data structures who say that is hA and hB.

(Refer Slide Time: 09:24)

```

// malloc device global memory
float *d_A, *d_B, *d_C;
CHECK(cudaMalloc((float**)&d_A, nBytes));
CHECK(cudaMalloc((float**)&d_B, nBytes));
CHECK(cudaMalloc((float**)&d_C, nBytes));
cudaEvent_t start, stop;
CHECK(cudaEventCreate(&start));
CHECK(cudaEventCreate(&stop));
// invoke kernel at host side
dim3 block (BDIM);
dim3 grid ((nElem + block.x - 1) / block.x);
printf("grid (%d, %d) block (%d, %d)\n", grid.x, grid.y, block.x, block.y);
// sequential operation
CHECK(cudaEventRecord(start, 0));
CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice));
CHECK(cudaEventRecord(stop, 0));
CHECK(cudaEventSynchronize(stop));
float memcpy_h2d_time;
CHECK(cudaEventElapsedTime(&memcpy_h2d_time, start, stop));

```

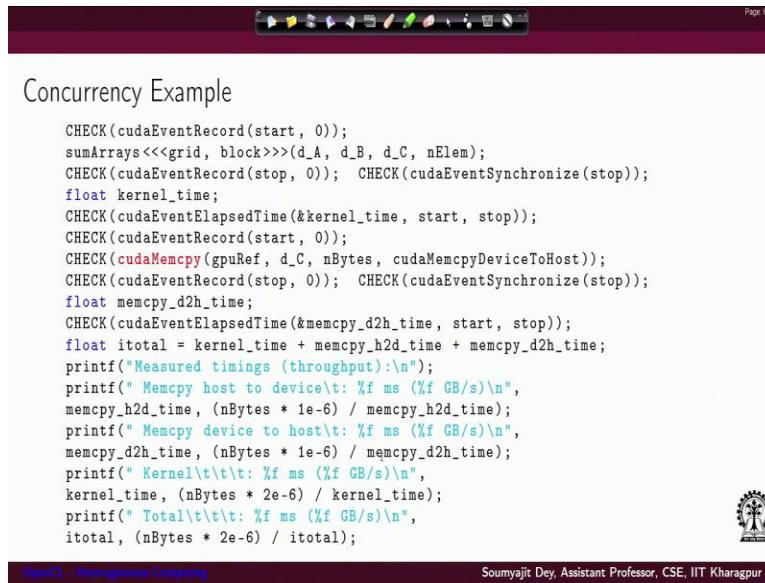
And then on the device side, we actually create the different device that is A, B and C for the individual memory in a device global memory. So, once this is performed, then for bookkeeping purposes we create we define 2 CUDA events start and stop and set them up here for recording. So, we just create the CUDA events and the moment we execute this function CUDA event create So, the timing of whatever is going on next will, they will start recording from this point in the start and stop data structures.

And then sorry so here we just create the events and later on when we will have to CUDA event record that is the point from which the start and stop events will actually start recording the timings. So here we are just creating and declaring the events and creating their corresponding objects. And then let us, we are just performing suitable declarations of the creed and blog dimensions, using code similar to whatever we have discussed earlier.

And we perform normal memory copy here just to show a normal execution. So you copy the whole side data to the device side data hA to hB. And you perform and before this, copy, you actually start logging the timing using a CUDA event record. In the timer, start in the event start, and here, you record the timing in the event stop. So you actually have the 2 timings before and after the synchronous the normal sequential data transfers.

So, for a normal sequential transport, what is the time elapsed that we will now we can check through a CUDA event elapsed time API by supplying it with this recorded times in this CUDA events at these points in the video will start and stop. So if you supply these 2 time point variables to this API, you get the value reported. And of course, after the CUDA event you will have a CUDA event synchronized call for the stop event.

(Refer Slide Time: 11:42)



```

Concurreny Example

CHECK(cudaEventRecord(start, 0));
sumArrays<<<grid, block>>>(d_A, d_B, d_C, nElem);
CHECK(cudaEventRecord(stop, 0));  CHECK(cudaEventSynchronize(stop));
float kernel_time;
CHECK(cudaEventElapsedTime(&kernel_time, start, stop));
CHECK(cudaEventRecord(start, 0));
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));
CHECK(cudaEventRecord(stop, 0));  CHECK(cudaEventSynchronize(stop));
float memcpy_d2h_time;
CHECK(cudaEventElapsedTime(&memcpy_d2h_time, start, stop));
float itotal = kernel_time + memcpy_h2d_time + memcpy_d2h_time;
printf("Measured timings (throughput):\n");
printf(" Memcpy host to device\t: %f ms (%f GB/s)\n",
memcpy_h2d_time, (nBytes * 1e-6) / memcpy_h2d_time);
printf(" Memcpy device to host\t: %f ms (%f GB/s)\n",
memcpy_d2h_time, (nBytes * 1e-6) / memcpy_d2h_time);
printf(" Kernel\t\t\t: %f ms (%f GB/s)\n",
kernel_time, (nBytes * 2e-6) / kernel_time);
printf(" Total\t\t\t: %f ms (%f GB/s)\n",
itotal, (nBytes * 2e-6) / itotal);

```

Source: OpenCL Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now in that way, we are just doing a bookkeeping we are just giving an example like how can you perform bookkeeping for all the normal calls? Again, we have a CUDA event record for start and you have a CUDA event record for stop and in between you make a normal kernel call for specific before let us do the sum of edits calculation for that kernel without using the synchronous API.

So, then we are having everything together in the same program. So, for that we are again performing a CUDA event synchronize on stop here just to ensure that the full stream the default stream here synchronizes on this event and then we can just change the elapsed time and all that now, in case I want to see that what do you what is the I would also like to see what is the time required for doing normal sequential copy from the device side to the GPU, CPU side.

We can make again and record a start event and stop event and in between I can copy back the data from the device to host and once that is done and I can compute the elapsed time here and I can print if this measure timings through the print API is here so I can just check what is the

normal execution time for this particular GPU device for a normal host to device copy for a normal device to a copy for a normal sequential kernel execution and what is the product time. The reason we are doing this and we are trying to show you usage of this API is soon we will be using this also for the synchronous case and compare the performance statistics.

(Refer Slide Time: 13:26)

```

Concurrency Example
[1 2 3]
// grid parallel operation
cudaStream_t stream[NSTREAM]; int iElem = nElem / NSTREAM;
size_t iBytes = iElem * sizeof(float);
grid.x = (iElem + block.x - 1) / block.x;
for (int i = 0; i < NSTREAM; ++i)
    CHECK(cudaStreamCreate(&stream[i]));
    CHECK(cudaEventRecord(start, 0));
// initiate all work on the device asynchronously in depth-first order
for (int i = 0; i < NSTREAM; ++i){
    int ioffset = i * iElem;
    CHECK(cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,
        cudaMemcpyHostToDevice, stream[i]));
    CHECK(cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,
        cudaMemcpyHostToDevice, stream[i]));
    sumArrays<<<grid, block, 0, stream[i]>>>(&d_A[ioffset], &d_B[ioffset],
        &d_C[ioffset], iElem);
    CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d_C[ioffset], iBytes,
        cudaMemcpyDeviceToHost, stream[i]));
}
    CHECK(cudaEventRecord(stop, 0));

```

So now, let us start with a synchronous case. So, we define this number of streams the maximum number of streams which are supported as was gathered from the system earlier. And then for each stream, we start dispatching a synchronous commands. So, we have a loop running from 0 up to the number of streams less than that so that it will run this many number of times. Every time you are creating 1 stream. By invoking the CUDA stream create command for the stream variable that is that has been declared for the stream.

You have a start of you start recording the timing for using the start event variable. And inside, then you mean inside this loop you will define another loop through each now you start dispatching data from the host side to the device side in chunks. So for so in this case, what we are doing is, first you have this loop. first we have a loop using which you create all the different streams. Then you launch this loop through each for each stream, you start performing a synchronous data transfer.

So for the first stream, you give you provide data from the host to the device with in some chunk. Now the chunk size has been decided by this i offset value, as you can see, so this is actually telling you that well, you want to actually execute. you want actually that will this stream will copy data of this memory chunk. So just if we want to illustrate to an example, so let us overall memory chunk, and that you want to copy the total size of the buffer.

And let us say that you have you want you want that in each memory copy comment, you copy, 1 chunk of the data, let us say the chunk size is this i element. And in each iteration, you said offset value. by 1 more successive among it just shifted by an amount that is equal to i element. So, you copied it from here to here, then from here, suddenly falling by i element size like this.

And also in each mean in each iteration that you have to specify the number of bytes. So, this is the chunk size in terms of number of elements, of course, you have to specify what is the total number of elements you want to actually what is the total number of bytes these elements occupy. So, that is expected in this I bytes point which we calculated earlier here as you can see, so, number of elements multiplied by the size of the type that is flawed, that is giving you the i bytes.

So, this is a chunk size so, you choose it here and then inside this loop for each of the streams, you are copying 1 chunk of data inside this loop for each of the streams, you are copying 1 chunk of data for input at hA to output a device dA idea input at hB to the device are dB , and so on so forth.. And after these copying from the input data to the device, you also, you know, you make the synchronous call to the launch the kernel on this on the stream.

So, you launch 1 kernel version of some areas, so, to the stream, of course, you want this kernel to work on that specific data chunk that has been copied. So, if we just draw an example, let us say in general is the whole side elements. So in the iteration, we have copied this data and this data to the device side. And so you have 2 of these segments and you want in the istream the kernel launch event will only work on this part in the next stream the kernel launches work on the next part so on so forth like that.

So, essentially you have launched in the full iterations of the loop, you have launched 1 kernel instance in each of the streams and in each launch, the kernel instance is working on that chunk of data that has been copied. So, in that way, you have all the streams engaged with executing some part of the original data or enough amount of data. So, inside this loop, in 1 full iteration of this loop, and that means you iterate through across all the streams.

And each stream is responsible for copying some, some specific chunk of input data. Since there are 2 input areas each stream in 1 iteration of the loop, you are launching 2 chunks of data from 2 input areas to the device, you are launching 1 kernel instance into the inside that stream to work on these 2 copied instances of chunks of data. And then you are launching another copy comment in the same stream to copy back the data law execute data that is produced by this kernel on this specific stream back to the host side.

So, all that you are doing is you are breaking the overall copy to the device, overall kernel execution and overall copy to the host operations into small chunks and each chunk is performed the each of the operations of each of our corresponding to each of the chunks are executed in different streams. What will have the advantage will be the pipeline execution that we have discussed earlier that different streams will have some amount of copy some amount of some amount of data to work on, and some amount of copy that.

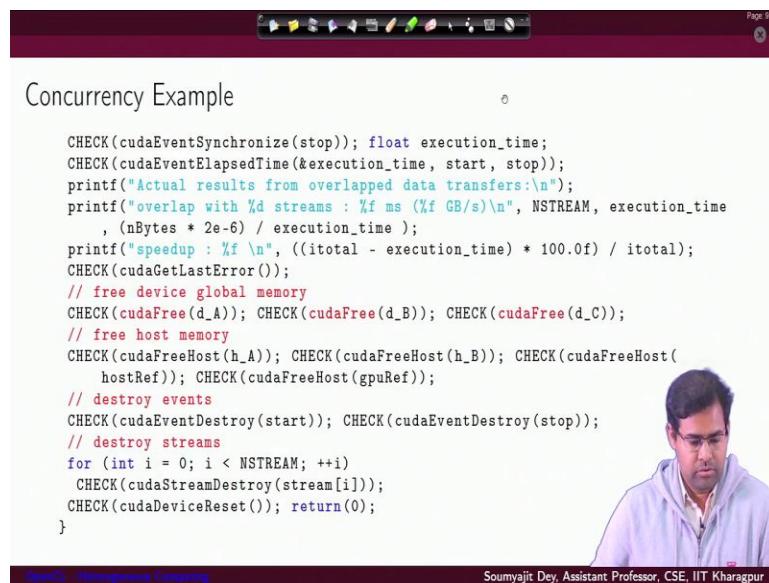
So they will be able to overlap and they will be able to make good use of the devices. So if I just take an example and draw this figure, so like in stream one, I have an H2D copy of a chunk for array A so let us say this is stream 1, you make a copy of array A some chunk, then an H2D copy of some array B. Then you execute the kernel, so this is the first chunk. And then you have device to his copy of the first chunk. Now, this can of course, overlap with certain operations, we are assuming that the copies cannot overlap.

So here when in stream 1, there is a here we have stream 1, we have the kernel operations, it can overlap with the H2D copy in off stream to have the second chance. So this this index is for the chunk in first stream, you are copying the first chunk. So a chunk would be the first chunk of

memory, the second chunk of memory, the third chunk of memory. In each chunk, I have this is a number of elements, the sizes, i bytes and all that so this will overlap.

And I can also have this is should be a second chance. I can have a H2D of array B second chunk, I can launch a second kernel instance here. So this will be the overlapping stream to and then I also have the device to host for the second chunk happening now in the stream 3, I can also have when this kernel execution is going on this is the point where I can start doing the H2D copy of the third chunk in the third stream for the array followed by A, B, and C, so on, so forth. So we have this kind of overlap execution. And that will actually help me to get the pipeline performance benefit like we have been discussing earlier.

(Refer Slide Time: 22:29)



The screenshot shows a presentation slide with a dark purple header bar containing icons for file operations, search, and other controls. The main title is "Concurrency Example". Below the title is a block of CUDA C code. On the right side of the slide, there is a video feed of a man with glasses and a light blue hoodie, identified as Soumyajit Dey. At the bottom of the slide, there is a footer bar with the text "Parallel Programming Computing" on the left and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur" on the right.

```
    CHECK(cudaEventSynchronize(stop)); float execution_time;
    CHECK(cudaEventElapsedTime(&execution_time, start, stop));
    printf("Actual results from overlapped data transfers:\n");
    printf("overlap with %d streams : %f ms (%f GB/s)\n", NSTREAM, execution_time
        , (nBytes * 2e-6) / execution_time);
    printf("speedup : %f \n", ((itotal - execution_time) * 100.0f) / itotal);
    CHECK(cudaGetLastError());
    // free device global memory
    CHECK(cudaFree(d_A)); CHECK(cudaFree(d_B)); CHECK(cudaFree(d_C));
    // free host memory
    CHECK(cudaFreeHost(h_A)); CHECK(cudaFreeHost(h_B)); CHECK(cudaFreeHost(
        hostRef));
    // destroy events
    CHECK(cudaEventDestroy(start)); CHECK(cudaEventDestroy(stop));
    // destroy streams
    for (int i = 0; i < NSTREAM; ++i)
        CHECK(cudaStreamDestroy(stream[i]));
    CHECK(cudaDeviceReset()); return(0);
}
```

So, we can, do it like as we have discussed, and then we can once we are out of the loop used to use the stop timer, and then we can actually measure what is execution time. And here we have just the normal freeing events and destroy events like we do for freeing and resetting the device now. So that is one way to perform the concurrent execution.

(Refer Slide Time: 23:03)

```

> ./simpleMultiAddDepth Starting...
> Using Device 0: Tesla K40m
> Compute Capability 3.5 hardware with 15 multi-processors
> CUDA_DEVICE_MAX_CONNECTIONS = 32
> with streams = 4
> vector size = 262144
> grid (2048, 1) block (128, 1)

Measured timings (throughput):
Memcpy host to device : 0.397920 ms (2.635143 GB/s)
Memcpy device to host : 0.180288 ms (5.816116 GB/s)
Kernel : 1595.653687 ms (0.001314 GB/s)
Total : 1596.231934 ms (0.001314 GB/s)

Actual results from overlapped data transfers:
overlap with 4 streams : 401.155762 ms (0.005228 GB/s)
speedup : 74.868576

```

And then with that we can have certain statistics. So we there is no point in reading out the statistics is for you to see that what are the execution times for in case you have it . How much time you have overlaps across streams, what is the effective speed up? How much of overlap as you can see that so, these are the measure timings you have that mem copy time from host to device mem copy time from device to host side for this specific vector size.

So, these are all specific to Tesla K40m GPU. So, you can set up maximum 32 streams here, were we are working with 4 streams here. And we are using these vector size we have launched the kernel with this gradient block setting and you have the mem copy times, for the host to device to it, the execution time and all that now, with the optimization of using concurrency by exploiting the streams, you have overlapped the 4 streams to get a reduced execution time of 401 millisecond. So that is the advantage we can see here from overlapped execution.

(Refer Slide Time: 24:17)

```

Breadth First Order

// initiate all asynchronous transfers to the device
for (int i = 0; i < NSTREAM; ++i){
    int ioffset = i * iElem;
    CHECK(cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,
                         cudaMemcpyHostToDevice, stream[i]));
    CHECK(cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,
                         cudaMemcpyHostToDevice, stream[i]));
}
// launch a kernel in each stream
for (int i = 0; i < NSTREAM; ++i){
    int ioffset = i * iElem;
    sumArrays<<<grid, block, 0, stream[i]>>>(&d_A[ioffset], &d_B[ioffset], &d_C
                                                [ioffset], iElem);
}
// enqueue asynchronous transfers from the device
for (int i = 0; i < NSTREAM; ++i){
    int ioffset = i * iElem;
    CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d_C[ioffset], iBytes,
                         cudaMemcpyDeviceToHost, stream[i]));
}

```

Now, just to highlight that this is not the only possibility, there will also another possibility, what we did where we are copying data in chunks and overlapping the data chunk copies with the kernel execution. But let us say we do it in an order that is for each stream instead of it running across streams and letting each stream copy that data and in chunk let us say, for each stream first, we copy all the data in then for each stream, we execute all the kernels in parallel on the data.

And then for each stream, we copy back all the data. So that can also be an option. So essentially, we are saying that so, here we are, we are also using the streams. But as you can see, we are not overlapping the host to device copies with the kernel execution, but rather we are dividing the copy operation across streams. So, let us say this is my input array and you have stream 1, stream 2, stream 3, you divide the stream into 3 parts.

And you ask stream 1 to do some copy then stream 2 to do some copy and stream 3 to do some copy and in effect you have the entire thing copied. So, in that way with respect to the streams, you do not have much at this stage, you do not have some advantage because, well, at this stage what is happening is you are asking stream 1 to perform a copy, then stream 2 to perform a copy and stream 3 is performing a copy. But if I look at the scenario that with respect to timing.

And the say now, once these things are done, so, you have the first stream, if you look into the program, so, the first stream has made a copy from the offset up to this size the second stream and

this first stream will next perform the copy up to this size let us and then after executing this loop, then it will execute the other loop through which you again have some synchronous dispatch of comments using which you are on each of the streams are launching their respective kernels.

And after that you have another third loop through each you are again executing a synchronous comment for copying bad the data where each of the streams are responsible for copying where certain chunks as per we as per we are directing here that which stream will copy where with chunk. So, overall, if we look at, well, what will be the overlap executions? So what do we really have? So, you have an H2D copy, let us say we are just trying to write the example.

So, you have an H2D copy of array followed by an H2D of array B and then so this is happening for some stream 1. And then for stream 2 while this is going on, nothing can happen. But now, when this stream 1 has finished the copying for the first and the second arrays, for whatever chunk size it is supposed to do, then the stream to can do its copy. So, it has the H2D copy of the chunk for which it is responsible in array A. And then each 2d copy for the chunk for which it is responsible in array B.

But while this is going on, we can have the kernel for the stream 1 execute somewhere here because there has been by this time this is going on. And this is an asynchronous API. So, you have just launched this comments, then you launch this comments, and then you will launch the copy back comments. So, the host, has just launched all these comments and moved out, this is where the action is happening.

And the CUDA runtime system is trying to manage the executions in the concurrent streams all for the GPU. So here we have sequentialization what so these are launched, but when they are executing, this will be sequentialization. But then when for the next stream, the copying is happening. You can have the kernel execution. And then for the third stream, let us say you have the copies that are starting for then it is chunk H2D for array A, then H2D for array B.

This is the time when the kernel can execute here for the second stream. And at some point of time, I will also have to copy back the data here. So you can go on and construct the rest of the picture

here. Like we are just trying to show that earlier, we have been getting overlap execution at our granularity. But here we will be getting overlapped execution at a different regularity. Because you can do the copies for A and for 1 stream entirely.

And then for the next stream when you do the copy. At that time, you can execute the first kernel. So it is all about how you are actually issuing the comments. And you can either issue them in a depth first order or you can issue them breadth first we are seeing breadth first because as you can see why this is first you break the execution of the original program into streams across the breadth of the program. So, you first kind of you try to submit all the copy comments for the host to device copies, then you try to submit all the kernel launch comments in another loop.

So, you go into streaming action and at the second level, once you have covered the breadth of all the streams at the second level, then you go to the third level in the third level, you know covered across all streams the breadth across all streams over a third level. So earlier was the order of streaming concurrency here we have earlier was the depth first order of streaming concurrency whereas here we have a breadth order of streaming concurrency and we can also have similar execution speed ups here.

So, it really is not the case that we get less speed up in one case or more speed up in the other is just like in what way you want to execute the comments and so that you can have overlapped execution.

(Refer Slide Time: 31:19)

The screenshot shows a presentation slide with a dark header bar containing various icons. The main title 'References' is at the top left. Below it is a list of three items:

- ▶ Khronos OpenCL Working Group. *The OpenCL Specification Version-2.1*. 2018 February 13,
- ▶ Kaeli DR, Mistry P, Schaa D, Zhang DP. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann; 2015 Jun 18.
- ▶ John Cheng, Max Grossman, Ty Mckercher *Professional CUDA C Programming*

At the bottom of the slide, there is a footer bar with the text 'OpenCL - Heterogeneous Computing' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right, along with the IIT Kharagpur logo.

And this will be some of the nice references from which we actually figured out what will be the important text that go into our presentation. Now some we have borrowed pictures as well as a text items from these references. And so these are the open CL and this is CUDA reference is a very nice book on professional CUDA programming. So with this will like to end our lectures for this week. And thank you.

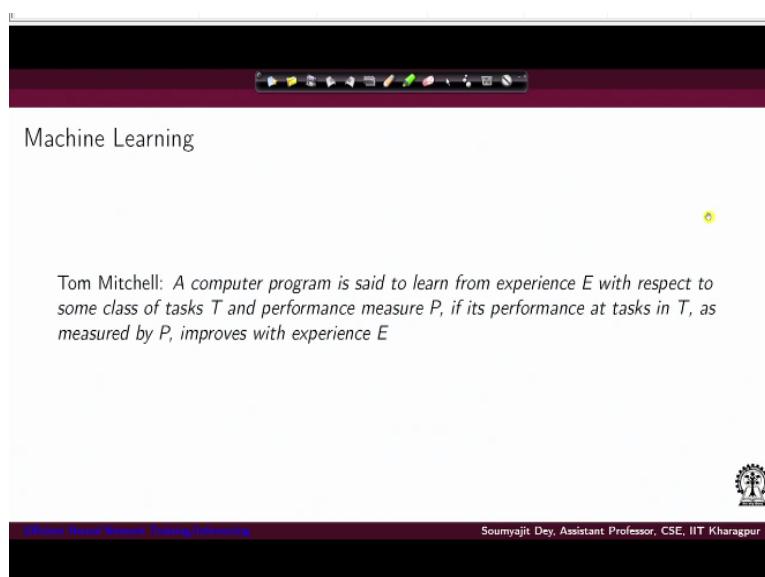
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-54
Efficient Neural Network Training/Inferencing

Hi, welcome back to the lecture series on these GPU architectures and programming. So we will start with one very relevant topic here, which is like how this paradigm of GPU programming or better to say in general assembly style programming, how does that really help in accelerating in ML workloads and to be more specific, we will focus on how this kind of architectures help to accelerate the execution of several I mean, both neural networks training as well as inferencing passes.

Because as we all know these are the most relevant workloads, at least in the ML and CS community and also CS community in general. And also it is finding a lot of applications in several cross disciplinary areas. And in terms of system execution, we can always say that this is the basic reason why these kinds of neural network based systems are finding popularity is that point 1, there is a lot of data available nowadays. And point 2 is of course, we have got this kind of large scale systems where we have the facility of GPUs to accelerate parallel programs run times by several orders of magnitude itself and that really helps in realizing this kind of large scale deep neural networks and they provide us significant and more and better to say usable accuracy in many application domains.

(Refer Slide Time: 02:06)



Machine Learning

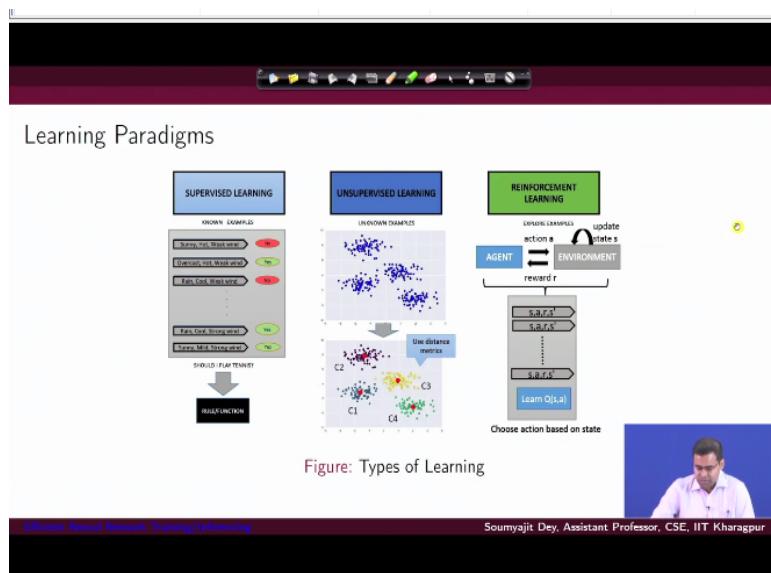
Tom Mitchell: *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E*

Efficient Neural Network Training/Inferencing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, with that background, I mean, we like to quote the famous researcher Tom Mitchell here where he says that a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P. If the performance it provides that tasks in T as it is and the performance measure of course is P, it improves with the experience. So the more experience you provide it is learning better how to execute this class of tasks with respect to a performance measured that is defined by P.

So we can say that it is learning from experience, provided it excels in future executions of such stacking task instances or maybe newer instances of data. I will just like to notify one very important thing that preparation of this set of slides have been significantly helped by my wonderful teaching assistant in course who really has done a great job in helping me prepare these slide sets. I mean, of course, the others were also there. But in these slides, he has got a huge contribution, fine.

(Refer Slide Time: 03:14)



So just the way we will approach it is that we will just give a brief overview of the basic computation that is performed while training a newer network for some tasks. So I will just put in a note here, that this is not going to be a ML kind of lecture, where we go into the intricacies of what is learning how it is done and all that, but rather, we will just take a specific workload, which is neural network training and inferencing process.

We will just identify what are the computational parts of that. And then we will map those computational parts to GPU architectures and see how they can be accelerated. That would be our approach here to be very clear. So doing a quick brush up of the learning techniques as we

all know that, these learning paradigms can be divided roughly into 3 aspects like you have supervised learning, unsupervised learning. And also there is this other paradigm of reinforcement based learning.

In strictly speaking if we just consider supervised and unsupervised learning, in the first case, we consider a set of label data that is already provided we there is a label like you have a data set and you have tags which actually said whether which are the positive as well as the negative or some other kind of classification tags are also there, which give you some information about the dataset.

In the other case, which is unsupervised, there is no such better supervision available that is that for the data, you do not have any such tags that are available. So, all ML techniques which work on the first kind of data are known or kind of classified under supervised learning paradigm, whereas the other case we call it as unsupervised learning.

(Refer Slide Time: 05:00)

What is Performance P?

- ▶ Performance P refers to some metric for assessing the quality of the rule/function learned.
- ▶ We restrict our discussion to Supervised Learning Problems.
- ▶ Supervised learning problems can be Classification (The target labels are discrete or categorical) or Regression (The target labels are continuous values).

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, in general, when we are talking about a machine learning system, so, it has a got a set of tasks, where you are trying to learn rules or functions based on such examples. So the examples can be characterized by a vector of real numbers or Boolean variables, depending on what kind of tasks we are talking about. Each example may have a target level like we discussed supervised learning in that case, there is a target level.

In case it does not have any kind of such levelling, we call it an unsupervised learning system. Examples are not available explicitly and some agent environment interaction must be

envisaged to extract meaningful examples, which is the case for reinforcement learning, which is gaining a lot of attention nowadays. So like we say here that the examples are not readily available.

But you set up an agent environment interactions, so you kind of model the environment or you kind of put in a learning agent which is trying different possible moves in the environment and finding out what I mean how good is that move based on some runtime evaluation. And that data is kind of used to figure out well, what is a meaningful example and then make good use of that example to learn a classifier or a similar kind of system.

And then we have this issue of experience. So of course, more experience means you have a bigger data set with reach examples. Now, of course, this is something which will be well known in the ML community. Of course, you can have a big data set, but that does not mean that there is enough information about the state space that you want to learn.

So the amount of data in that dataset may not be that important, as like the richness of the data set in terms of information that can be mined from that data set about the characteristics of the status.

(Refer Slide Time: 06:59)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "What is Performance P?". Below the title is a list of bullet points:

- ▶ Performance P refers to some metric for assessing the quality of the rule/function learned.
- ▶ We restrict our discussion to Supervised Learning Problems.
- ▶ Supervised learning problems can be Classification (The target labels are discrete or categorical) or Regression (The target labels are continuous values).

At the bottom of the slide, there is a video player interface showing a video of a man speaking. The video player has a progress bar and a control bar with icons. The footer of the slide includes the text "Efficient Neural Network Training/Inference" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

Then well, what is really performance, performance refers to some metric for assessing the quality of the function that has been learned. So well, at the end of doing all that training and everything based on the data that is provided you learn a function, but you have to calculate whether that function is good enough as for some deployment scenario or not. So do you need

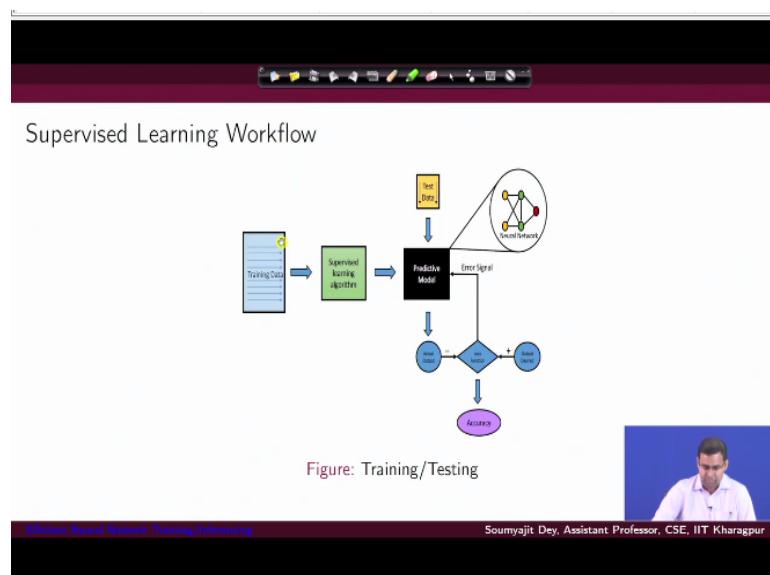
a measure, usually, we call it a cost function, which evaluates the performance of whatever function or rule you have learned.

So in our lecture, we will discuss only supervised learning kind of problems and this kind of problem. So that means we are assuming there is a training data set with proper levelling that is available and that will be used to learn proper rules or functions. And for this kind of problems, the primary classification that is done is like, well, you can have a classification task or you can have a regression task.

These are the 2 kinds of systems that you can really learn. So, when you learn a classifier that means, given a new data input, you can infer the class of the data. So, basically you can kind of classify all the inputs to certain categories and you are trying for a new input and you are trying to find well which category it best fits in. So, that is typically a classification task.

In the other case, where we have a regression class, regression would mean that you have this input data and you are trying to find out a fitness function like what function would fit that data pattern nicely. So then again, I would say you are learning a continuous function, so that given any new data point, this continuous function will give you a possible value for as an output, taking the new data point as an input.

(Refer Slide Time: 08:54)



So in general, when we are speaking of supervised learning tasks, we already have a generic workflow like this, that means you have a training data set with labeled data, you have this supervised learning algorithm, which will work on this data. And at the end of the day what it

will come up with is a predictive model. That model can either be a classifier or it can be a regression model.

Now, well how do I test the goodness of this model. I have trained it using the training data, I have figured out using some algorithm which ran on the training data that this is the model I am getting. Now to test the fitness of this model, I give you some other data as input called the test data set. And using this test data, I check with what is the inference that is done by the model, what is the classification of the new test data or what is the regression value that this predictive model computes.

Well, but this is test data that means for the input values, I also have the suitable output candidates available to me. So, based on whatever the model is predicting, and whatever is the real output value that was supposed to be in the test data, I calculate some error and that is measured by the loss function. So, I have this concept of loss function, I mean why do I have this function.

There is an error. I have calculated something and used my predictor and there is something that test data says that this should have been the real output. So, there is a difference. How do I really characterize this difference? Because we are speaking of data in a high dimensional space, it is not like a single tuple . The value is 1 which is predicted, whereas the actual value must be 1.1.

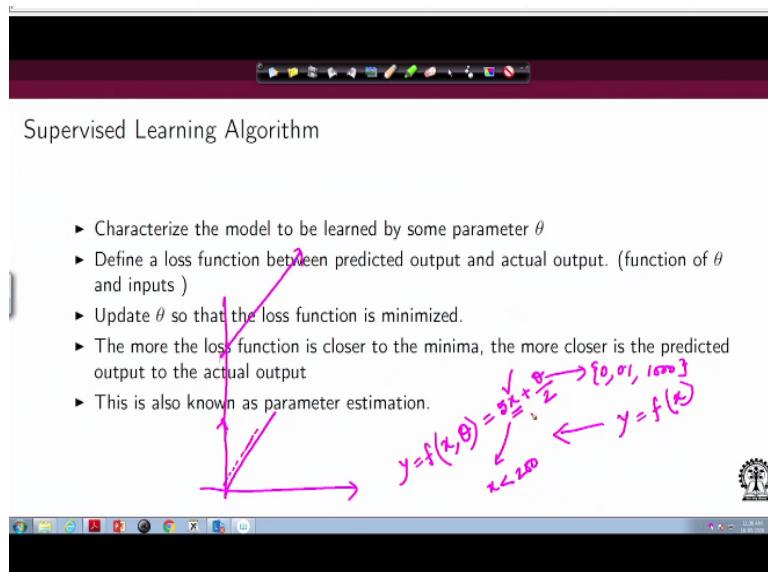
It is not really like that, in general, we are talking about high dimensional data borrowing from linear algebra, we are supposed to bring in a possible fitness function, which characterizes this distance metric of in the high dimensional space that will, I am predicting this point in an N dimensional space, whereas the actual output is this point in that same N dimensional space.

So I need a suitable distance metric, which I characterized by choosing a loss function. And then, using this loss function, I generate something called an error signal, which I can typically use to refine this predictive model. And that is how the loop will go on and finally, if I see that, whether I am happy with the accuracy that I am getting. So, if I see that well, the accuracy is not good enough.

Then I said that, well, this system is not learning enough from the training data, I may read some more data, or I may need to change the supervised learning algorithm itself or the other alternative can be that I need to change the parameters of the algorithm. So this is the generic flow that I have for any supervised learning system. So you build a model, you evaluate the quality of the model. You accordingly tune parameters of the model to figure out whether you are getting a better model or not. And this step is complex as you can see that this is an iterative step- you are building the model, you are checking the error that is generated ,based on that you may be tuning parameters of the model here. But finally, by exploring all the options in the parameter space, you can report a maximum accuracy.

If that is good enough you are done. If that is not good enough, you change things here or you may say that well even changing here is not helping me, you increase the input data set by some way. So, even if changing algorithms is not really helping that means, this training data set is not really rich, you are not covering enough information theoretic aspects of the input state space of the model fine.

(Refer Slide Time: 12:58)



So, now if we just enlist things step by step, so, any standard supervised learning algorithm would have the following characteristics, it will characterize the model to be learned by some parameter theta, it will define the loss function between the predicted output and the actual output, it will update this parameter theta that is, this kind of parameter tuning, as long as the loss function the value is not minimized.

So, of course, as you can see that the loss function's value depends on the model, that I have trained. And now inside the model I have put in this unknown variable which I am calling a parameter of the model. So, typically what is a parameter. So, let us say I am defining a function $f(x)$ for y , instead of that I write well let the function be $f(x)$ theta of y . So this is a normal mathematical function.

We are used to x is the input that generates the output. Instead of that all that we are saying is the output is y , this function is not only characterized by the input, there is a theta. Well, what can that mean. Let me just take an example here. So let us say I am just trying to write a closed form expression to motivate the problem, do not think this will be the actual thing in general. So let us say the function is $(5x + \theta)/2$.

I am just writing one function here. All I am trying to say here is that you can see that the value of the function for a certain input x , not only depends on the x itself, but also on what value of theta I am choosing. It can be 0 to say that it is absent. It is not really influencing the function, $5x$, it can have a value 0.1 to say that well. I am not biasing the output of $5x$ by much, it can have a value 1000, which can say that well I am playing, I am giving a big shift, I am giving a big positive shift.

In those cases, when the value of x is let us say, I mean less than some value like 200. I mean, I am just trying to motivate again, I have said there is nothing there is not a strict mathematical characterization, I am just trying to say that if you draw this our favourite coordinate system, and then if theta is really not there, you have a line like this slope. If you put a small theta, you are just giving a small constant value to that slope.

And then again, if you have a big value already here, then initially you will be rising from here, but this there is already a significant value. But again, this is a very nice day example I would say, in general, theta having a very complex relationship, because it is maybe a high dimension in the high dimensional space and all that. So, where all that we are doing is we are saying that width in that model, there is a lot of information, I do not know. And that is what I am characterizing by the theta.

And I am asking the learning algorithm to figure out what would be a good value of theta, so that the loss function that is the error in my estimate is getting minimized. So, the more the loss

function is closer to the minima, the more closer is the predicted output to the actual output. So in that case, I would say that well, in that case, I have chosen the theta in such a way that it approximates the actual functional relationship very nicely. So essentially, this is nothing but a kind of estimation of the parameter theta.

(Refer Slide Time: 16:53)

Linear Regression

- ▶ Given a dataset $\mathcal{D} = \{(x_i, y_i), x_i, y_i \in \mathbb{R}, i \in [1, n]\}$, learn a function $f_\theta(x) = y$ that can predict any unknown x_j not in the dataset, with a label y_j with reasonable accuracy.
- ▶ The function f_θ is of the form $f(x) = wx + b$.
- ▶ The parameters to be tuned are the slope w and the intercept b .



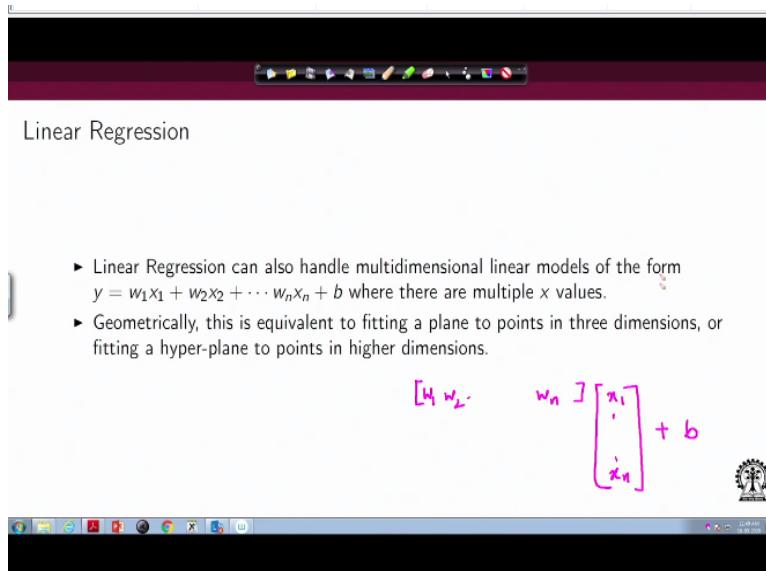
So we will start with regression here, like as we discussed that regression is nothing but learning a continuous function. So you have a data set, a label data set, given inputs y , you have outputs x . So you have this kind of value like that, and you are trying to figure out what kind of curve would approximate it nicely. As you can see, there is not going to be a good approximation because there are several values, which are lying at quite a distance.

So in general, that is what I want to learn, I want to learn a function f , which is parameterized by x as well as theta. I mean, it takes as input x , it is parameterized by theta and is going to output y . And I want this function to fit the test data as nicely as possible. So I want to learn this function, which can predict any new value of x , what is the output y . And I want that to happen with reasonable accuracy.

So what we can do is let us say we make a linear approximation here. So let fx be some wx plus b , and I want to parameter. So essentially, this w and b represents it is like a form by which I am saying that the parameter theta influences x , so in this case, I am saying that let theta be such that it essentially appears w and b . The first item w is just killing x and the b is giving a constant shift, right. So I have a function of this form.

And I want to tune the slope and the intercept. So that I get a functional representation, a linear representation, which fits these input values as nicely as possible.

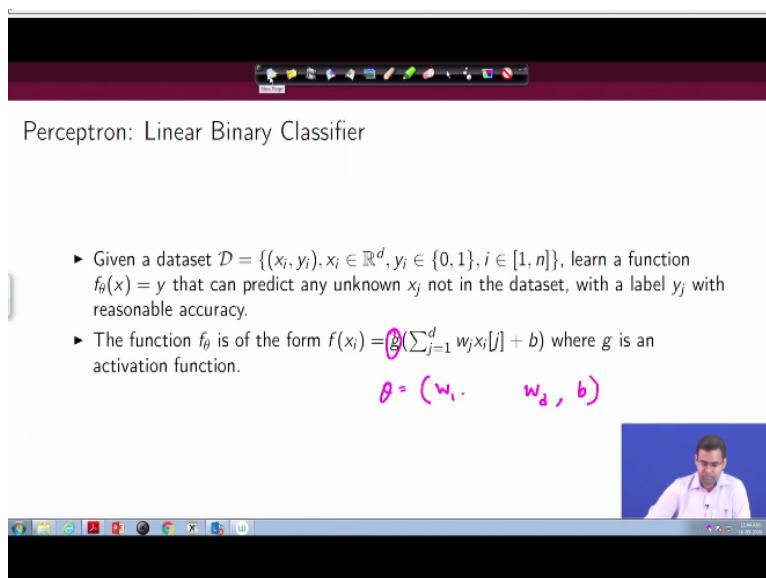
(Refer Slide Time: 19:02)



Now, of course, in general these input data can be high dimensional like we have been saying. So, x can be a vector comprising $x_1 \times x_2 \times x_3$ up to x_n like that. So, then I am not even I am also not saying that w is a scalar, but rather w is also a row vector. So, that essentially we are speaking like this that this is so, theta contains values w_1, w_2 like that up to w_n , which gets multiplied with x_1 and up to x_n .

And then I have the shift b , which is the value here. So, essentially in this case, I am saying that we are trying to figure out a function which fits in a multidimensional x .

(Refer Slide Time: 20:00)



So, this kind of linear binary classifiers or what we have been popularly knowing them in the literature perceptron. So, consider a data set like this x_i, y_i these are the training data that are there for given x_i input the output level should be y_i . Now, we are saying that there are all values where the dimension D . So, there is y_i belongs to \mathbb{R} to the power d . And in this case, we are considering a binary classification that means, the outputs are either 0 or 1.

Perceptron: Linear Binary Classifier

- ▶ Given a dataset $\mathcal{D} = \{(x_i, y_i), x_i \in \mathbb{R}^d, y_i \in \{0, 1\}, i \in [1, n]\}$, learn a function $f_\theta(x) = y$ that can predict any unknown x_j not in the dataset, with a label y_j with reasonable accuracy.
- ▶ The function f_θ is of the form $f(x_i) = g(\sum_{j=1}^d w_j x_i[j] + b)$ where g is an activation function.

Efficient Neural Networks Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, as you can see we are making a jump here. This is our model for regression which we have spoken about. So, in that case we come trying to approximate the data by a continuous function, the other problem as we discussed earlier, this classification, classification means given some input, I want to see the membership of the resulting output to a class. Now, in the simplest possible case, the class can be binary.

So in that case, I will say that we will have class 0 and class 1. So for a given input, just see whether it goes maps to a 0 or a 1. So that is why this is a simple linear binary classifier or in other words, we call it a perceptron. So simple perceptron, so we will learn a function for this problem, where we are trying to figure out what are the suitable settings of theta such that I have, I mean the most accurate possible binary classification.

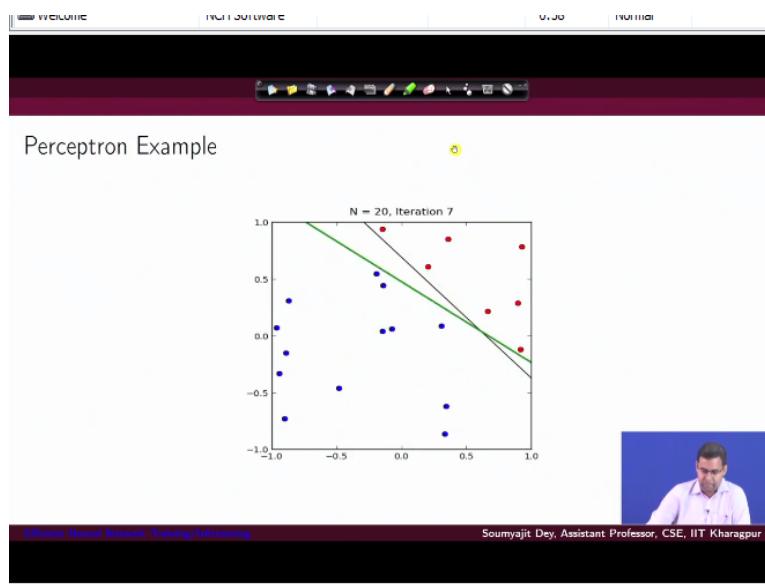
So we are trying to figure out $f_\theta(x)$ equal to y , that can predict any unknown x_j , which is not in the data set with a label y_j , which with reasonable accuracy, of course, in this case also we consider this f_θ to be of this form. Given an x_i I am trying to figure out first, what is going to be the linear scaling here. That means, in this case, we are saying that well you have this set of widths, w_1 to w_d .

I am considering that x is a vector of size D , and they are all getting multiplied by the x_1 to x_d . And so, kind of I am trying to see what is theta here. And then also we have another parameter that is b . But wait there is something more g . Now, what is that? So, in this case, what we are doing is, we are saying well, let us first apply our linear scaling idea which we discussed for the regression problem.

So essentially, we have a linear function map which is taking the input vector x generating a single value by multiplying it with a row vector w_1 to w_d , and then adding a constant t . But does that solve our requirement? No, what is the requirement, that requirement is not to figure out a real value y , but rather to figure out a Boolean value y which is taking either a classification 0 or 1.

So, what we do here is, whatever we get out of this calculation, we map it to something called an activation function, which will perform the final classification. So, it will have a thresholding nature, so that for up to some values, it may map it to 0 up to some values beyond that to 1, it will have some kind of thresholding feature, which will be useful to get the initial computation of the linear function map to a corresponding 0 or 1 partition class.

(Refer Slide Time: 24:04)



(Refer Slide Time: 24:09)

Let $x = [x_1, x_2, \dots, x_n, 1]$ and $w = [w_1, w_2, \dots, w_n, b]$
The value of the dot product $w.x$ is the same as $\sum_{j=1}^n w_j * x_j$.

```

while convergence is not reached
    for i in range(len(y)):
        y_pred = heaviside(w.x)
        dw = (y[i] - y_pred) * eta * x[i]
        w = w + dw

```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if we try to use this idea and design an algorithm for performing such classification task one can be as follows. So, you have this kind of set of inputs x , and you have for them, the actual outputs known for them, the y_i 's and you are trying to estimate what should be these parameters w 's and b 's. So, what you do is, you run an algorithm through which what you are trying to see you are trying to have some initial values for these w 's.

You use them to predict the y 's. And then you figure out what is the difference between the actual y and the predicted y . So, that is your error. You multiply that with a constant η called the learning rate. You multiply that with that position's x value. So, I mean, we will come and see why we are designing this like this that will come later on. Let us just try to assume that here you are scaling this error by these values.

And trying to figure out what can be an incrementally better estimate of w . So, this calculation gives you a modification in w , you add that modification here. So, this can be a positive or a negative quantity, you get a refined parameter w , which will again push into that in this loop again, let us understand all that we are doing, we are using a set of x , we are assuming a set of initial w 's.

We are using them to estimate a predicted value of y . Then we are calculating what is the difference between the predicted value and the original value. Again, here we are making use of `heaviside` as a threshold in function like g and then we are getting an error, that error we are multiplying by a constant as well as the current x value for that component. And then what we are doing is we are modifying the w .

Let us try and understand again. So this is a simplistic case, all we had done is we have removed the b here as it needs to be 0, we are just playing with w simplistically, we are using one special function called heaviside. Check what it is, it is a simple thresholding function. And using heaviside we are thresholding it to get a predicted value of y . We calculate the error by learning, read the value of the chosen x to get a possible shift in w and accordingly tuned w .

We keep on executing this loop as long as we are not really satisfied with the values that we are coming. That is what we will define as convergence later on. Now, we are just trying to give an example here of what really that looks like. Consider a 2 dimensional space, you have x 's and y 's and we are trying to learn a classifier or linear binary classifier or a perceptron here.

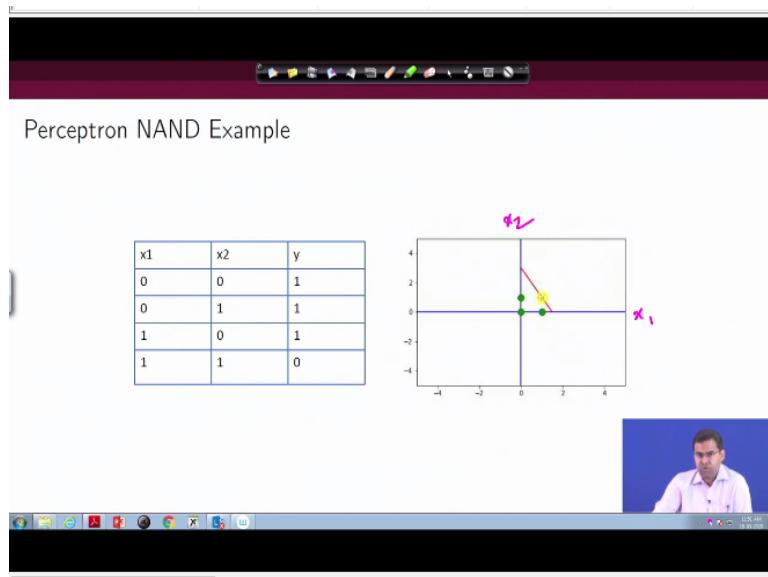
So, let us say these are your data points and in an ideal system, this is your classification line. So, these are the data points that should be in one class, these are the data points in blue that should be in the other class. If you start executing our algorithm that we just discussed, what may happen is you start with a test classifier like this. So this is the line that you are getting by assuming some initial values of w 's.

Because what really the initial values of w gives you. It gives you a line. Of course, here you also have a constant value b which is not 0. Now, as you keep on executing the algorithm, those values of w will keep on getting tune it will keep on changing and what you expect is that with certain time you are going to get possibly a good approximation. Now, why is this a good approximation for this classifier that will be hypothesized.

Because as you can see, this rightly classifies the data points of one class, which is marked in red from the data points in other classes, which are marked in blue. So that is a good classifier, then, as you can see, they are not identical, but this serves the same purpose at least in this case. So, with this, I will have great convergence, because this is also correctly classifying all the data points just like the original hypothesis classifier function.

So, I do not have anything new to learn from this data set. So that I will receive convergence and I must have stopped here.

(Refer Slide Time: 28:48)



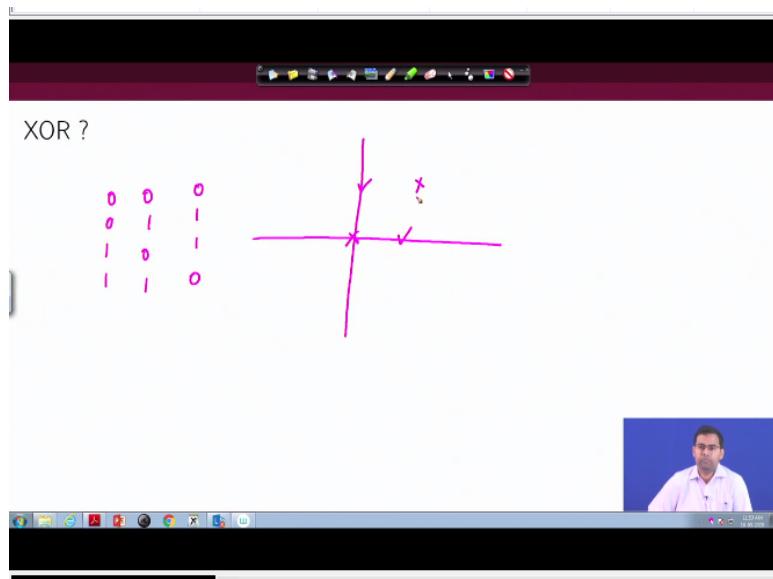
Consider certain examples here. For example, suppose I am trying to learn a NAND relationship. So what is NAND? So, this is the truth table of NAND here. Consider that you have inputs x_1 and x_2 . So you have an input x , a 2 dimensional vector a 2D represent here you have input x , which is a row matrix, a column matrix of size 1×2 , and you are trying to use a binary classifier to a 0 or 1.

So, that is like learning any 2 input Boolean function, in this case, the choice is NAND, we have the truth table of NAND here, both 0s, you get a 1 0 1 you get a 1 1 0 you get a 1 only when you have both 1s. So, that will give you an end which is 1 and now you negate the end to get 0 which is an end value. If you plot these here in your coordinate system, so the positive cases are the green points.

So and for all those cases, essentially we are just plotting here, the x 's - x_1 and x_2 . So in this direction, you have x_1 , in this direction x_2 . And here, we are just trying to see what are my positive cases. So 0 0 is a positive case, because output classification is 1, I am just saying positive instances of point 0 and all that. And you have 1 0, and 0 1 1 0 and 0 1, both also as positive cases.

Now for 1 1, that is a negative or 1 class classification, which is marked in red. If you apply the previous algorithm here, you will learn to slide, so as you can see, the classifier does its job because it is classifying everything that is positive on one side, keeping the 1 value here on it with suitable tuning or redefinition, I can put it I mean like this, as the classifier is doing his job.

(Refer Slide Time: 30:56)



And interesting question in this case can be what happens with XOR. First of all, let us understand that we are learning a linear classifier here. That means we are considering that these points are linearly separable. But in general, it may not be so. So let us see why. I mean, this method may not be working. So well we saw, what is the truth table absorbed by the way.

So the next task is to figure out whether this is linearly separable or not. So if you use the previous method, then you have 0 0. So that is a 0 class classification, 1 0, and 0 1 as 1 class or positive classification, and this is again, 0 class or negative classification. As you can see, you cannot draw a line, which will put the 0's on 1's side and 1's on the other side, you cannot have such a relationship here.

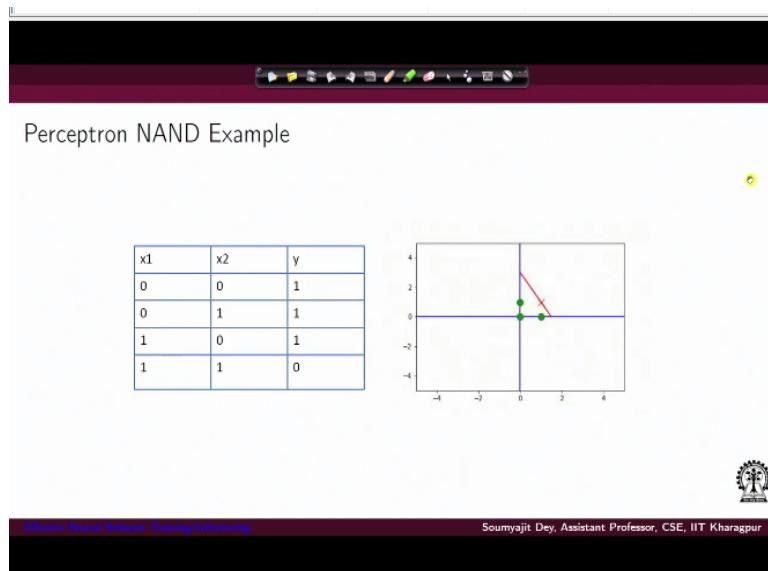
That is why the XOR relationship will not be linearly separable. And this algorithm where I am assuming that the function approximation is going to be a linear function is not going to work. So we have to see how things can be managed at this point. With this, let us end this lecture. And we will take it further from here in the next lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-55
Efficient Neural Network Training/Inferencing (Contd.)

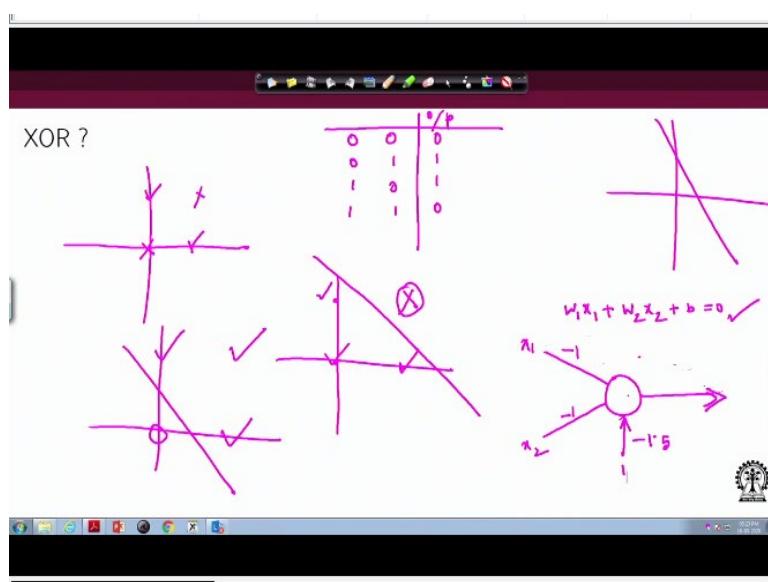
Hi, welcome to the lecture series on GPU architectures and programming.

(Refer Slide Time: 00:32)



So, in the last lecture we have discussed this perceptron example for the NAND function where we found that will, it is separable.

(Refer Slide Time: 00:37)



And then the issue was that we are trying to identify what happens if we consider XOR as the function. And so let us start from that example. So if I consider XOR so considering the truth table of XOR, so we have a positive output here, here. And this is the output corresponding to the other class, as we can see that we cannot have a linear classifier for the XOR function because fundamentally the function is not linearly separable.

So now this brings us the problem that how really are we going to tackle this issue. So getting into the root cause of the problem. Let us first identify what really does this, I mean binary classifiers where the functional form is linear, how really does it work. So in general, if I consider any linear classifier is basically the equation of a line . So I can simply write, it is an equation of this kind of a form.

So essentially this is the way we can capture any such linear classifier which is separating the function and separating the values. But as we see for XOR, this is not possible. However, what about the other situations as we have seen that for NAND is definitely possible. So if we just recollect what happens for NAND. So we will have positive outputs at all these points essentially 0 0 0 1 1 0.

And we will have the NAND function, will evaluate to 0 only for the point 1 1 where AND evaluates to 1. So that definitely can be classified by a linear classifier. So this is linearly separable. Well, what about OR we can just do a quick check and find out what happens with OR so far, OR I have positive outputs here, here and here. That is 0 1 1 0 1 1. And here I have the negative output.

So this is also linearly separable . Now, in a similar way, I can figure out what happens with AND also and we can see that for each of these cases we can actually have a linear classifier . So the next thing that we will try to do is we will try to figure out whether these kinds of linear classifiers can help us to figure out how to handle the XOR function, which fundamentally is not linearly separable .

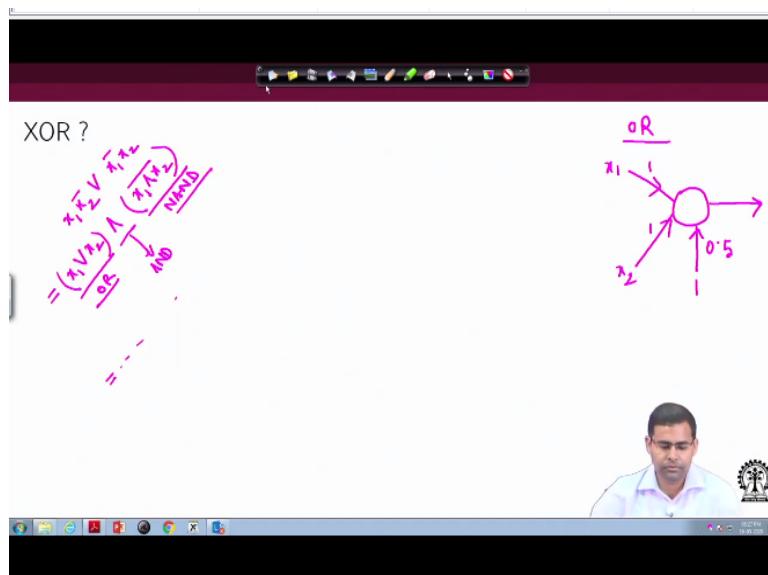
So for that, let us see first whatever it has been our linear classifiers if I can express them in the form of perceptrons, since I mean and that should be technically possible, because they are linearly separable. So, this was about NAND function . So, since I have a line like this, I can

check the corresponding since I will get a linear classifier, I can see that this will give me a line like this, which has treatable values of w_1 w_2 and b .

And what will happen is we will have weight w_1 and w_2 as minus 1, here, here and here. I can get the output where I give input of 1 and I have a multiplier factor of - 1.5. So, what we are trying to do here is creating a small perceptron corresponding to this NAND function . So, we are trying to see that, since this NAND function has linear separability I can express this in terms of this kind of an equation.

And if I can express it in terms of the equation, I will have the value of w_1 and w_2 and b as like this - 1 - 1 - 1.5 and then I can create a small perceptron like this where x_1 gets multiplied by the weight - 1, x_2 gets multiplied by the weight - 1 and then I add the bias term which is this constant 1 multiplied by the weight of - 1.5. And then, if I am considering the output, I get the corresponding line which will have this kind of feature .

(Refer Slide Time: 05:56)



Now, similarly, I can just stroke out well for all I can create a similar perceptron. So let me just write it for OR, I will have a similar behaviour with x_1 , x_2 , getting multiplied by these weights 1 and 1. And here we will add the bias, constraint 1 multiplied by the scaling factor 0.5. And we can easily check that this is going to act as our OR. Now when we're talking about XOR well, let us figure out first whether I can actually express it using this OR and NANDs.

Actually, it should be possible. So for XOR as we know I can actually write it as x_1 or x_2 and x_1 and x_2 bar. So, this is my OR, this is my NAND and here I have AND. So all these

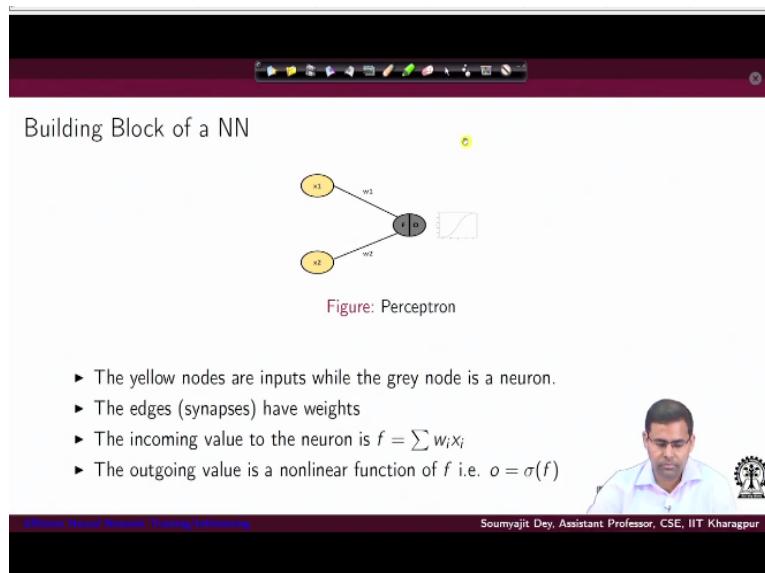
are kind of linearly separable . So, if we just break this down, I can have a representation like of course, I mean, you can just check that, in this I can have this thing expressed like this. So, what it does is it gives me an idea that it may be possible for something which is not linearly separable to consider that, well.

I have some intermediate neurons with whom I can actually create these values. But what I mean, well, in this case for XOR I am able to create an I mean, I am able to check that I can create an ended version of XOR using the ending of OR and NAND's where each of these OR and NAND's are linearly separable. But how does this work out in general, can I say that for any such complex function, which is not linearly separable, I can have linear combinations combining and creating such a value.

Well, that is definitely not going to be true. Because following our linear superposition principle, that is really not going to be possible . So, in general, I need a bit of non linearity in case I am trying to model sufficiently complex functions which are highly nonlinear. It has to be a combination of individual linear transformations and they need to be combined with non linear functionals .

So, we can say that well in I am creating a version where I am considering 2 linearly separable values OR and NAND and combining them with an AND, but in general how does that work.

(Refer Slide Time: 09:30)



So, the way we will look at it is as we have been discussing that we need a simple some kind of nonlinear function to be combined with linear components. And this is exactly what is done

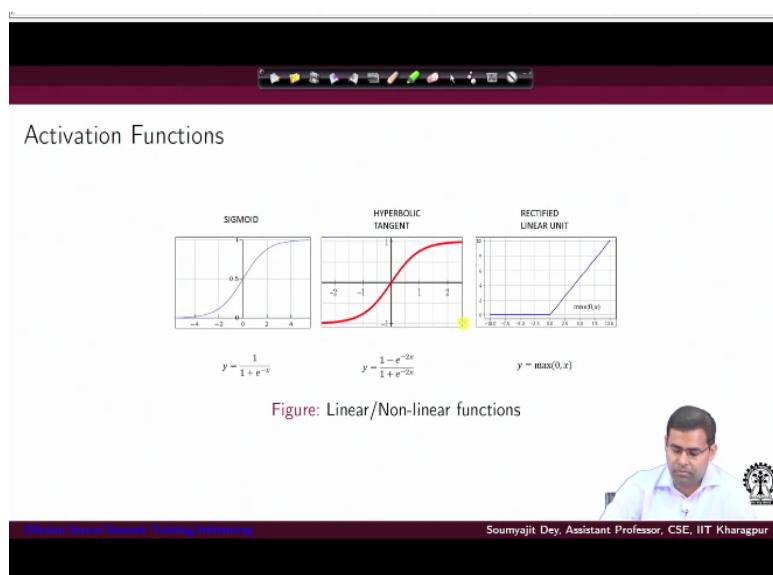
in the real perceptron which forms the building block of a normal neuron network. So, the way we do it is like this, you consider these kinds of inputs x_1 's and x_2 's, which are marked here by these yellow nodes.

And these inputs you are multiplying by weights w_1 and w_2 . So, that gives you linear transformations. You can add the weights, I mean the bias here the b values, I mean, so, you put a constant and multiply by a scaling factor like we have been doing for modeling NAND's and OR's, but then again, what you have is those kind of linear expressions, $w_1 \cdot x_1 + w_2 \cdot x_2$, and similarly, many more, and you can have some constants added through a proper bias, but that is not enough.

We do not have any kind of nonlinear component here. So, fundamentally, what we will do is well, will add our nonlinear transformation here, which is known as the activation function. So what we do is we will have edges or synapses like this, so these are my edges or synapses modeling, the neurons synapses they are kind of modeling the linear transformations, and then add these nodes, we have neurons enabled with their corresponding activation functions.

For example, here we have a picture of one activation function. And what we can observe here is, these are nonlinear functions, all it is doing is taking the input and mapping the output corresponding to this kind of a functional map that is provided here. So, in this case, you can see that this is not a line, so it is a nonlinear function. So, what we get here is let us call it a sigma and the final output is a sigma composed with f and f is the linear transformation.

(Refer Slide Time: 11:51)



Now of course, this the function we have here is not the only possible nonlinear activation function there can be many others. So, when we say that this neuron fires that means this function will have the input to the output. So, this example that we have here is for sigmoid for which essentially the closed form expression is like this y equal to 1 by 1 plus e to the power - x .

So, as you can see it has a 0 crossing at 0.5 for this input 0 you get the output as 0.5 beyond that the function saturates to 1 and -1 as you go forward. So, 1 and 0 as you go forward. So, of course, if you are putting the x value is very high, essentially this exponential term will decay and you get a 1 . You put the x value as very low, as very high but negative, then again this will become very positive e to the power - x .

So, then again what you get is that they will come down to 0 so, the highest possible value is approximately 1 asymptotically, the lowest possible value is as intrinsically reaching 0 . And at 0 at x equal to 0 , you have 0.5 . So there is a sigmoid function. And in general, you can have other kinds of nonlinear functions. For example, you can have dysfunction, which is the tan hyperbolic function.

As you can see, it is a different functional with respect to sigmoid, here we have the closed form functional representation. And also as you can see that the value you are essentially having z I mean as central 0 and asymptotically with positive and negative values you are approaching 1 and -1 respectively. The other very commonly used activation function is value or rectified linear unit.

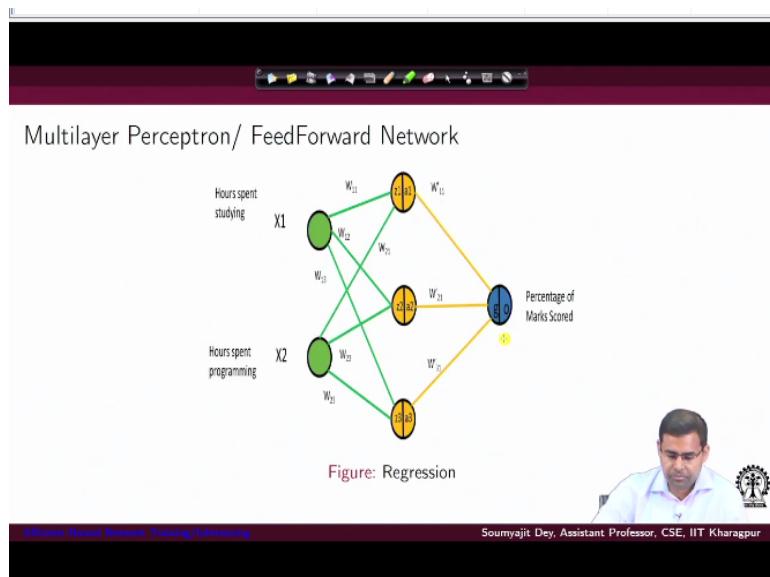
So essentially is just a max between 0 and the input value. So if your input value is positive, that is what you get as output. So that is approximate that is kind of modelled by this linear slope , and if your input value is negative when you get a 0 , so that is why you have for all negative values you have essentially on the x axis. So, these are the commonly used activation functions in neurons.

Again, what is the objective here, we are trying to model highly complex nonlinear relationships, the way we are doing it is we will create a neuron network built using these kind of perceptron blocks, where the neuron is essentially modeling a nonlinear activation function the inputs are modeling linear combinations with bias and we are assuming that with this kind

of combinations, we should be able to come suitably approximate any complex nonlinear function.

And also as you can see, these functions satisfy some nice properties like differentiability, so that it helps you in computing the weight, something we will see how this kind of specific functional forms really help you.

(Refer Slide Time: 15:00)



So, with this background, let us come to the genetic structure of a multi layer perceptron or a feed forward network . So, what is the motivation behind this kind of multilayer perceptron as we have seen that their OR functions which are definitely not linearly separable, we will like to combine them with suitable linear maps along with non linearity and eventually we like to get outputs .

So, in that way for modeling complex behaviours, we will have inputs going inside to neurons which are there in the hidden layer and then they map the outputs to the output layer. And inside each layer you have neurons which are connecting with the inputs or the previous stage neurons. And whatever set of layers you have in the intermediate between the input and output layer.

They are the hidden layers and in deep neural networks, I mean all that difference is you have significant depth in the hidden layers, how that helps is you are able to combine a lot of linear and nonlinear activation functions in different possible ways. So, that gives you the flexibility

to capture many possible complex functional relationships. So, these are examples taken from a source called Welch labs, which is on YouTube.

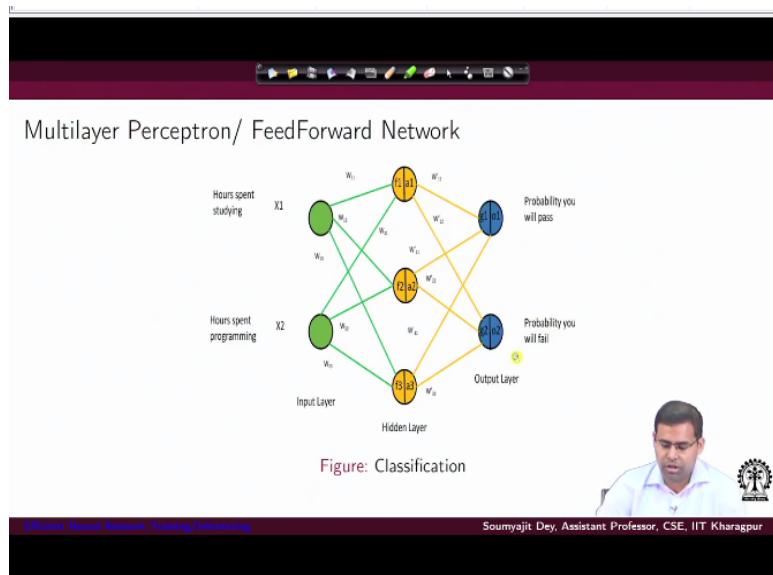
And it provides a very nice example, that suppose, you are trying to create a correlation between hours spent studying an hour spent programming along with what is the probability that you are going to pass or fail if you are going to model this kind of relationship. Well, this is an example there could have been many other examples. So, we are trying to propose a simple neural network architecture here.

So, from the inputs, you have these weights. So, since from a 2 input system, you are mapping to a 3 neuron hidden layer and as you can see, this is like a fully connected system here that means every hidden layer neuron has its connection to all the inputs in the input layer . So, you just denote these connections by the weights w_{11} from x_1 to f_1 . Similarly, w_{21} from x_2 to f_1 like this.

Then w_2 from x_2 to f_2 w_{23} from x_2 to f_3 like this . So, here, what you have is after you get the values here from the synapses where you get x_1 multiplied by w_{11} , you have the activation functions $a_1 a_2 a_3$ like this, through which the outputs will pass once that neurons fire and then you are again multiplying them by these weights, the output layer weights w_{11} prime w_{12} prime w_{21} prime w_{22} prime, so and so forth.

To get into the output layer, where again you have this node denoted by $g_1 g_2$ like that and finally, you have the output activation layer with I mean nodes denoted by $o_1 o_2$ like that.

(Refer Slide Time: 18:16)



So, this is a sample architecture for a classification system. So, essentially you are doing a_2 class classification for whether you are going to pass or whether you are going to fail like that. And then, what about regression. Well, like we discussed earlier regression is basically about computing the continuous function which is the approximation idea we have been talking about.

So, that would mean at the output you have only one node in the output layer and that gives you a value in a continuous range well. Whereas, in classification, you get values in all of this and whatever is the maximum you will like to come to consider the classification well, the definitions can vary as well your architectural feature . But overall we can say that these are generic structures for classification.

We have multiple outputs, and you are computing of real value here and here you have only one output for that because you want a functional output.

(Refer Slide Time: 19:15)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Note'. Below it is a bulleted list of course objectives:

- ▶ The objective of this course is to get you acquainted with the computation involved while training and testing a neural network.
- ▶ We shall not discuss core ML principles which should be followed while designing neural networks for various problem domains.

Below the list is a small yellow circular icon with a question mark. To the right is a portrait of a man, Soumyajit Dey, wearing a light blue shirt. The background behind him features the IIT Kharagpur logo. At the bottom of the slide, there is a dark footer bar with the text 'Efficient Neural Network Training/Inference' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right.

Now, what is the objective in our case, again, we will try to focus on that we are not going into that deep theory of neural networks and all that, rather than that, we are trying to focus on what is the exact computation involved while training and testing a neural network. So of course, we are not going to discuss core ML principles that need to be followed and all that we are trying to consider is that suppose I am given this network.

My objective is to train the network figure out by training we mean figuring out these values of these weights, given a test data set, so that we get a set of values which best approximates the functional relationship underlying the test data set, and, and how that computation can be accelerated by the GPU architectures and all that.

(Refer Slide Time: 20:03)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Neural Networks'. Below it is a bulleted list of steps in the computation process:

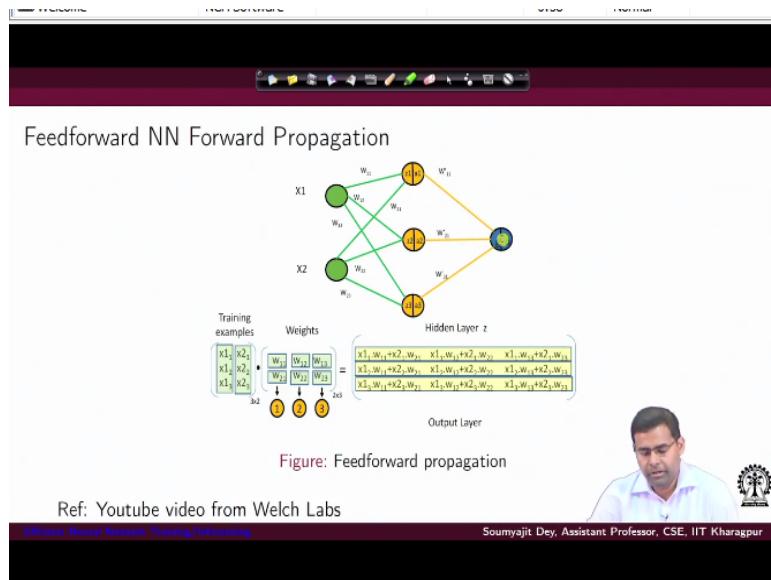
- ▶ Each neuron of a layer accumulates a weighted sum of the inputs from the previous layer.
- ▶ Each neuron applies an activation function to its input and propagates the output to a neuron of the next layer..
- ▶ This results in a series of linear and non-linear transformations from the input layer to the output layer.
- ▶ The predicted output value for every input example is a function of the input feature values and the weights and activation in the network

To the right is a portrait of the same man, Soumyajit Dey, wearing a light blue shirt. The background behind him features the IIT Kharagpur logo. At the bottom of the slide, there is a dark footer bar with the text 'Efficient Neural Network Training/Inference' on the left and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur' on the right.

So, just doing a short summary here. So, for neural networks, what we have is each network has I mean, you have a set of neurons, and each of these neurons have a layer they accumulated the weighted sum of inputs from the previous layer and then a neuron fires that means, it applies its activation function and it will simply propagate the output of the activation function to the neurons of the next layer.

So, essentially that we in that way, we have a series of linear followed by nonlinear transformations going on the inputs and propagating finally, to the output layer. Why is this necessary, because, as we discussed earlier, they are in real life you have many complex functions which are not linearly separable. So, the way we like to approximate them is through a combination of linear active linear maps, followed by nonlinear transformations and again linear maps like that.

(Refer Slide Time: 10:56)



So let us now come to the underlying computational parts here. So let us consider this simple system of feed forward neural network forward propagation. Okay, so let us see how things are going on here. So we have this training example where the input is like this. So you have a set of inputs, like for x_1 , you have these inputs, like x_{11}, x_{21} . So that is the first input pair.

Again, the second input, I mean, the second one you have is x_{12} and x_{22} , and then the third one is x_{13} , and x_{23} . So, for these training examples, they are going to get multiplied by the weights. Now, as you can see, this is the weight collection. Essentially, you have a 3×2 matrix, because you have weights, w_{11} , I mean, because each of them are mapped to this like this .

So, as you can see, you have got 6 weight components, but how really are they arranged. So, for w_1 you have 3 components because w_1 maps to the first neuron in the hidden layer second and w_{12} and w_1 also has a component mapping to the second neuron in the hidden layer and similarly, w_1 also has a component mapping to the third neuron in the hidden layer .

So, these are the essentially the components of w_1 . Similarly, you have 3 components for w_2 and then when you are considering the input, so, for this x_1 and x_2 , you have these training examples now, when you are creating these multipliers you want the final value to be propagating to the hidden layer . So, like since x_1 has these components , you have you must mean the way this is arranged that means, for each of the values you are going to for each of the training examples.

You are going to have values for these kinds of couples and they are going to get multiplied with the corresponding components for the w 's . So, this gets multiplied with this column and similarly for the next row, you have it multiplied with the second column like that, and in that way, you are going to get the corresponding entries. So, the first row and the first column multiply to give you the 1 1 at entry here.

So, that is what you get so x_{11} multiplied by w_{11} plus x_{21} multiplied by w_{21} , and then you have the second x_{12} , multiplied by w_{11} . I mean, if I consider the next multiplication like the one I highlighted here, this is going to give me the second row and second column, that is essentially going to give me this entry . So x_{12} , multiplied by w_{12} plus x_{22} multiplied by w_{22} .

And in that way, I can easily figure out all these values that are slowly getting computed here. So using the training examples and the weights, I can have these values propagating to the net towards the output layer. Of course, these values now, we will have to pass through the activation functions before they get mapped to the before they get further multiplied by the weights and propagate to the output layer .

(Refer Slide Time: 25:06)

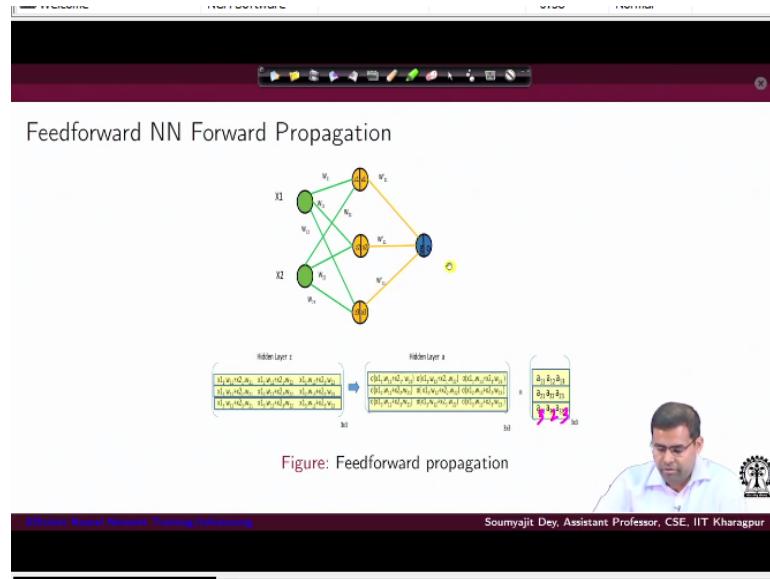


Figure: Feedforward propagation

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, let us see how that happens. So, here you have the inputs for the hidden layer . So, these are the inputs for the hidden layer and here they get they need to be I mean computed I mean, so, you have this is the output here for the hidden layer and then this hidden layer output gets mapped to the activation function some sigma and when this activation function applies out of that, you are going to get a set of activations which we have noted here like this a 1 1 2 like that.

This is what we're trying to show is what happens after the activation functions get applied here. So essentially, as you can see that for a 1, you will have next this like that, and so forth.

(Refer Slide Time: 25:52)

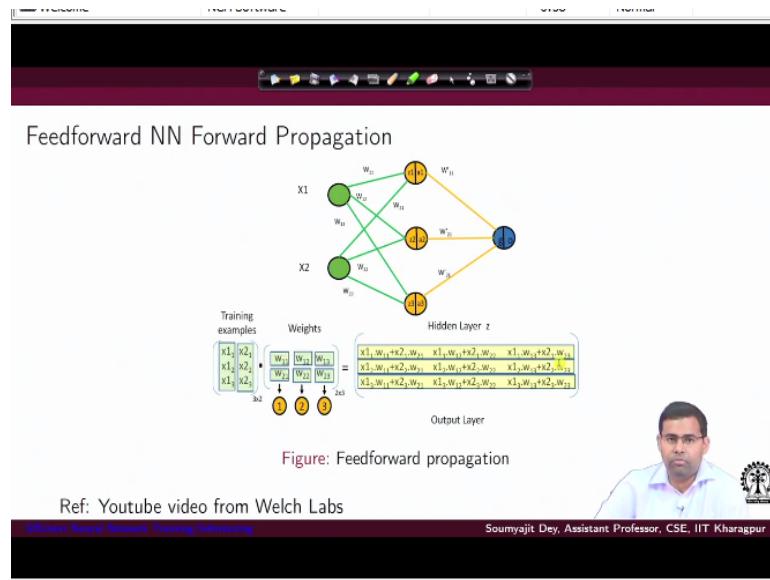


Figure: Feedforward propagation

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then at the output layer, you have these inputs coming out of the activation functions of the hidden layer. So, again let me repeat what we just did. So, we have these training examples

getting multiplied by the weights . So, as you can see, since we are considering this 2 input system x_1 and x_2 and we have weights average like this, so, this is how one instance of the training examples will get propagated.

Since the weights functionally form here a 3×2 matrix you multiply it like this to get components flowing through the interaction . So, since you are going to have this I mean since you are going to really have this kind of a setup, let you have these many values propagating through the layers. So, at each point here, you are getting values like this.

So, let us say for x_1 it gets multiplied by w_1 and then you have x_{21} getting multiplied by w_{21} , and they are getting summed up here. And then after activation is going to move on here. Similarly, if I do get the other points like x_{11} , getting multiplied by w_{12} , and, I consider the other combination of x_{21} , getting multiplied by w_{22} , and it is getting received here.

So, in that way I can say that each of these neurons are receiving the different possible combinations. So this guy is going to receive a combination of x_1 and x_2 multiplied by w_1 and w_{21} . This is going to receive a combination from x_1 and x_2 multiplied by w_1 and w_2 . And similarly, let us just take another example. This one is going to receive from x_1 and x_2 and it is getting multiplied by w_{13} .

And the other guy is getting multiplied by w_{23} . So similarly, let us just consider one such example again, so for x_1 of course that is going to happen for all the different components of the values. That is also going to happen for all the different components of the values. And the components are like as we have for x_1 , we have 3 components. For x_2 we have 3 components, and so on, so forth .

So for each of these components, I will have this kind of combination. So for the first set of components, I have the combination w_{11}, w_{21} , then w_{12}, w_{22}, w_{12} to w_{23} . And this is like what gets propagated in each of them, so for the first activation function, this is the part that is getting propagated for the second activation function here. This is the part that is getting propagated.

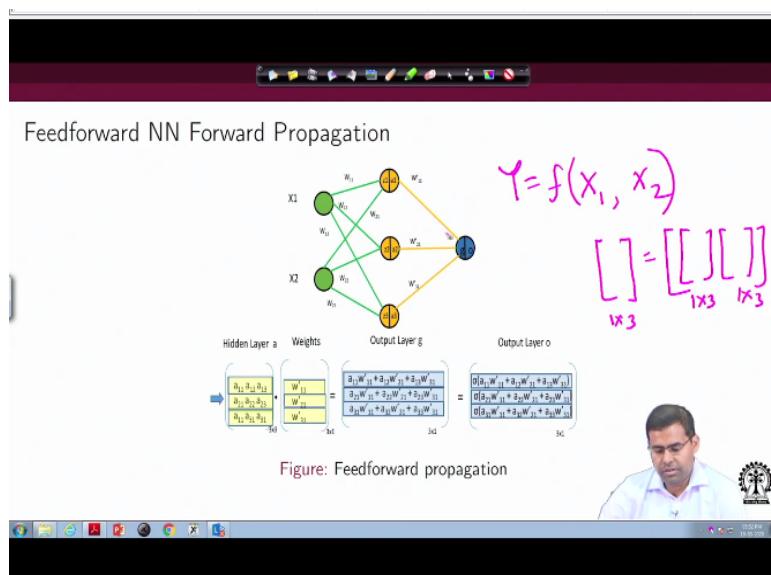
And for the third activation function here, this is the part that is really getting propagated. I hope that is clear from this. And that is really happening because we are also considering that

the inputs they are in this kind of a vector form here means something different, I mean for example, if the inputs are scalar that would be a different kind of computation. So, you have to really take that into account for x_1 and x_2 both of which we are considering because the inputs are arranged like this in size 3 vectors .

X_1 is the size 3 vector x_2 is a size 3 vector and then for them I have this kind of values that are getting propagated. And after that when these values are propagated, and then I have these entire things flowing through the network and then are reaching these activation functions and then the activation functions modify I mean they do suitable thresholding and they will create outputs.

For example, for the first one I will get this a_{11}, a_{21}, a_{31} it should be like that, a_{12}, a_{22} so, this one let me just correct it here. So, this should be a_{31}, a_{32} and a_{33} like that .

(Refer Slide Time: 30:21)



So, once this activation value reaches then what do we really have. So, these are the values that are flowing out of the network and now, they will get multiplied. So, this is what is coming and they now get multiplied by the output weights here . So, what are the output weights here, so, as you can see now, since I mean here we are concentrating the regression example, so, everything is going to be marginal .

So, from the hidden layer to the output layer we have single connections and they are represented by w_{11} prime w_{21} prime w_{31} prime, 1 represents a single neuron and output and 1 2 3 the first index denotes different elements in the hidden layer. So here we have the

hidden layer outputs, they are going to get multiplied by these weights of the synapses that connect the hidden layer to the output neuron here.

And then what we have is these vectors of a_{11}, a_{12}, a_{13} , I mean they are I mean and a_{21}, a_{22}, a_{23} and so, like these, this entire 3×3 thing is getting multiplied with these corresponding weights here. So essentially here, what you have is a combination like this. So what is very propagating. So, you have a_{11} multiplied by w_{11} prime plus a_{12} multiplied by w_{21} prime, a_{13} , multiplied by w_{31} prime these values.

So considering the first components, considering the first components that are coming out, so, that is what is coming out here. So, this is the first component, this is the set of first components that are coming out for each of them and so here what we have is all the components now, from each of them whatever is the first component that would get to propagate through this network.

And then you have at the output these kinds of combinations. For example, what is the second element in the output layer, it should be a_{21} multiplied by w_{11} prime plus a_{22} multiplied by w_{22} primes so on and so forth. So, if we just again mark it out, so, this and this will give you the first element and similarly for the others. So, with this we have all outputs flowing to the output neuron where the corresponding activity function is now getting I mean, that is indicated here by the sigma and you apply the sigma here.

So, then you get the final output from the output layer, which will be again a vector of size 3. So, here again just to summarize, we are considering that we have inputs which are kind of 3 size vectors. So essentially we are trying to learn a relationship like this. So x_1, x_2 . So we are trying to learn it like this where this output is a vector of size 1×3 and the inputs are all 1×3 s.

So that is the kind of relationship we are trying to learn here. And accordingly, the system is tuned, as you can see for the input weights, they are all having 3 components here. And for the output words, they are all having since I am finally mapping to a single neuron, they are all having single weights. So, I mean, we are not getting into further details of I mean, what is the motivation and all that.

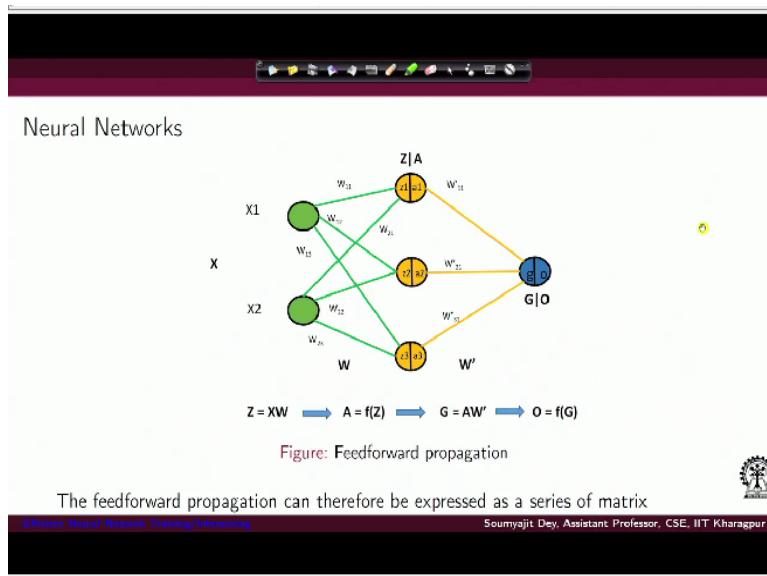
We are trying to cover it in simplistic terms more in terms of what is the underlying matrix multiplication relationships that are going to happen . So with this, we will end this lecture. And in the next lecture, we will see how this really lends to the neural network training part and how I mean what are the underlying computations that need to be done for the training part. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-56
Efficient Neural Network Training/Inferencing (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming. So, in the previous lecture, we just took an example of a simple neural network.

(Refer Slide Time: 00:30)



And we are trying to show how the inputs flow through these linear maps and then nonlinear activation functions and finally reach the output layer. Next, we will try to set up the loss function for such a neural network and see how the weights are being trained in a typical neural network. So, these are the basic architecture that we have. As you can see that if we start now using vector notation.

And try to understand what the mathematical relations that are holding up here. So let us say this is my input X . And if I consider a matrix representation of all these weights let that be W . So, what we have here, reaching the hidden layer is some Z , which is nothing but X multiplied by W . So you have X multiplied by W and that is the Z which is reaching this and then if denoted using f , the activations that are happening here.

Then after the activation they get output matrix A, which is f of Z f represents the activation function. So we have A that is the activation flowing through this and then this activation captured by this vector A gets multiplied by this, again, a set of linear maps denoted by these W primes. So let us capture it by the capital matrix W prime. So you have A W prime giving us the input matrix flowing into this which is some G.

And then on this G we again have the activation function f applying again by f we mean a set of activation functions applying component wise and that gives me the final output . So, we have if we just summarize we have some input X with a linear transformation W followed by f and then again the linear transformation W prime and then again we have the f that gives me the output .

(Refer Slide Time: 02:35)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title 'Neural Networks' is centered at the top. Below the title is a bulleted list of five points:

- ▶ Given the structure and weights of a neural network, we now know how to compute the predicted output value for an input example.
- ▶ The structure of the network i.e. the number of layers and number of neurons per layer are referred as hyperparameters (to be decided by the user).
- ▶ The weights are the actual parameters (θ) which will be learned during the course of training.
- ▶ Training involves the minimization of a cost/loss function.

In the bottom right corner of the slide, there is a portrait of a man and text identifying him as 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'. The footer of the slide includes the text 'Efficient Neural Network Training/Inference' and the date '10/10/2023'.

So, what do we really want to do here. So, we are given the structure of the neural network, we are trying to find out what we are trying and suppose we are given the weights here, we now know how to compute the predicted output from the input example. Now, of course, the overall objective is I am given the structure here and I am trying to figure out what are the suitable weight values.

So that the underlying overall function, which as we can see is basically a sequence of compositions here functional compositions. How does it best approximate the overall input output relationship . So that is the training job. Now, the structure of the network, there is a number of

layers, the number of neurons per layer and all these. This entire architecture, that is what we call we denote by mathematically as the hyper parameter.

And the weights represent the actual parameters, which you want to learn here. So, essentially, we will do the same thing like we discussed for the simple perceptron example of a binary linear classifier that will set up a loss function for this kind of a vector matrix system. And we will see how on that loss function we can perform suitable tuning of weight parameters and minimize the loss function.

(Refer Slide Time: 03:59)

Loss Function

$$\begin{aligned} o &= f(g) \\ &= f(AW) \\ &= f(f(Z)W') \\ &= f(f(XW) \cdot W') \end{aligned}$$

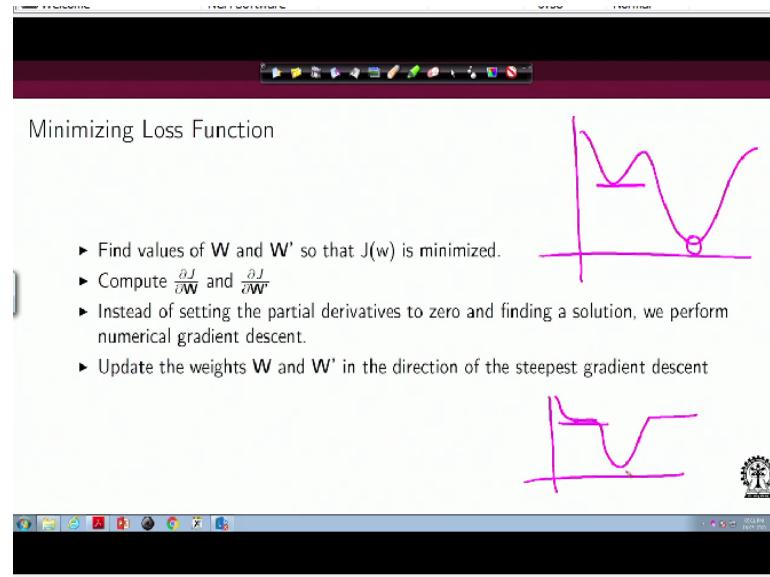
- ▶ Define a loss function $J(w) = \frac{1}{2} \sum_{i=1}^n (y_i - o_i)^2$ where y_i and o_i are the actual outputs and predicted outputs of input example i respectively.
- ▶ Recall the feedforward propagation equations.
 $Z = XW, A = f(Z), G = AW', O = f(G)$
- ▶ Therefore, $J = \frac{1}{2} (Y - f(f(XW)W'))^2$ where the summation operation is over the elements of the column vector obtained from the loss function.

So, we consider a simple mean square loss function. So, that is essentially you consider the overall errors, you have a test data set where you have the actual y 's and these o 's are the computed outputs . So, this is your mean squared error loss function. So, y_i 's and o_i 's are the actual outputs and predicted outputs for a given input example x_i like that. So, you do the square sum it up and consider n number of values.

So, you will have this mean squared error like this. And then let us understand how this get represented by the overall matrix system that we set up earlier. So, as we can see, output f output is like a function of, so if we write it down here f of G . So that is f of AW' . So that is $f(f(Z))$, W' . So that is $f(f(XW))$, and then W' . And so if I use the vector notations to read the loss function, we have something like this.

So we will have the summation outside and then y minus (y is the output). And this is what the function relationship gives me $f(XW)$, multiplied by W prime square, W prime and finally at the output, I have this square, so that would mean so, this is what is W prime and then this there the square is about .

(Refer Slide Time: 06:15)



So, overall what is the requirement, these W and W primes are my unknowns and I want this loss function to be minimized . Now, as we know the way to do a function minimization is that you do a derivative calculation with respect to the variable which is W and W prime and figure out what are the minimized and put the values back and then you get the minimum value for the function .

But observe that here we have this W and W prime as vectors . So, we need to compute the partial differentials which are $\text{del } J \text{ del } W$ and as well as $\text{del } J$ and $\text{del } W$ prime. And also we have several problems here because even if I compute, what are the points where this partial derivative set to 0, and we find a solution, we need not be sure that those are exactly the only minima.

Because for a complex function, for example, let us take a simple example here, this may be a complex function you can have a local minima and being a global minima. So, you consider one solution you may get you get stuck in this kind of a local minima. And you can miss the global

minima, which one is this, or there can be several options, maybe you can get stuck at this kind of foot hill, which is kind of flat.

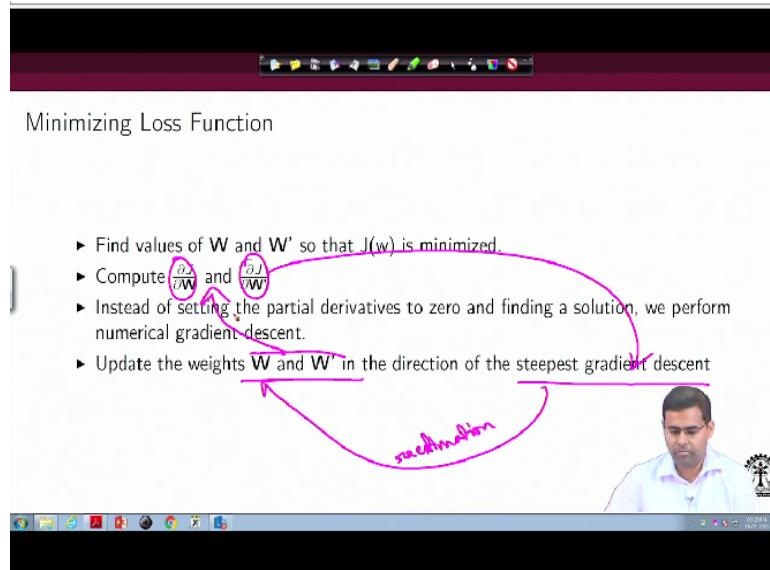
And you think that you have got the solution without sampling the other territories and you do not really get here. So, overall, it does not really make sense for a sufficiently complex system where these W 's and W primes are significantly big, it may not really make sense for such complex functional relationships that you really try to set the partial derivative to 0 and get a minima.

Rather, it makes sense to actually do some optimization here in the way that you perform what we call as numerical gradient descent. That means, you try to figure it out at different points, you compute this ΔJ , ΔW and $\Delta J \Delta W$ prime, and you try to figure out what are the directions in which these values, the value of the loss function decreases with the highest possible gradient.

Because then that is the maximum slope. So you should really focus your search on those reductions. I hope I am making myself clear, let me just repeat again, since J will be in a sort of sufficiently complex situation, this J has J is, these W 's and W primes of a very high dimension and J will have a very, very complex functional form. So, rather than really trying to set the partial derivatives to 0.

And compute all such minimas, which may be quite intricate, it makes sense using exact closed form expressions, it makes sense that you try to evaluate, you first find out how does these things look like .

(Refer Slide Time: 09:31)



So, you will get some expressions, then, on those expressions, if you figure out what the value of those expressions are for your current estimates of both W and W' . Then you get that well, for my given value of W and W' at those values, the rate of change of the loss function is something like this. From those let us change, you get that well, in which is the direction in which the loss function is decreasing maximally .

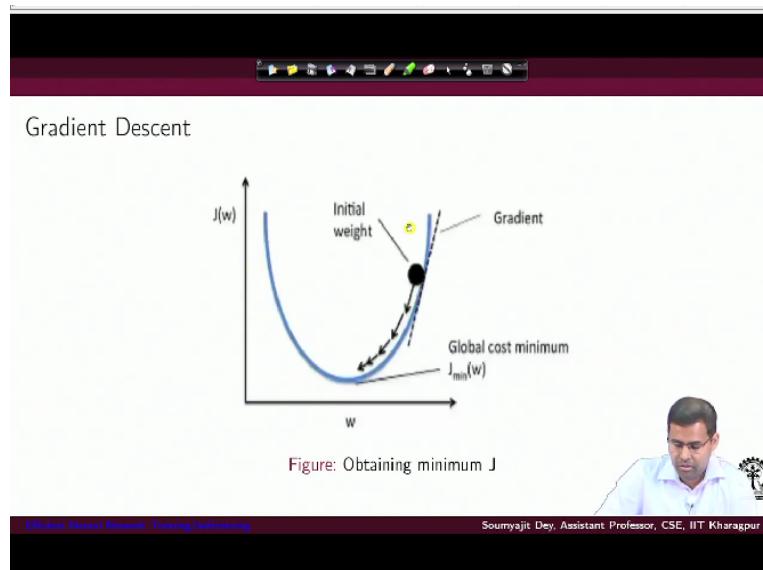
And then you start taking a different estimate of the value of the loss function. What you do is when you are trying to figure out what are the directions in the space of W and W' on which the directions in which the loss function is minimized maximally. So that is giving you the steepest gradient descent and then in these directions, you do an re-estimate of both W and W' . And accordingly, you again re-evaluate these values .

So, you use these parameter values to figure out what is the direction of the steepest gradient descent in that direction, you do a re-estimation. After that estimation, for those modified values of W and W' , you again compute these values. You carry on doing this until or unless you get convergence or you see that they are not really descending . Now, again, just to let you know and of course, most of you are aware of this already for sufficiently complex systems, even this is going to be difficult to do.

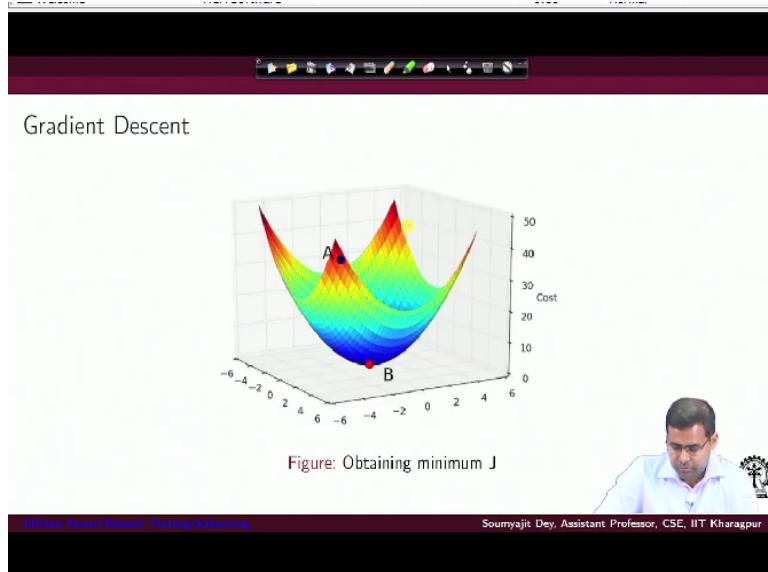
And what people really do is a very smart algorithm called stochastic gradient descent . We are not going to that detail for the time being rather, we will just see that what are the underlying computations, which are involved for finding out these values, because as you can see, whatever will be your heuristic for computing the gradient descent and finally, figuring out what should be the values of W and W prime I am going to be happy with every time you need to compute this del J, del W and del J del W primes.

So, what we are really interested in is the mathematical as well as the computational perspective of complexity of these values .

(Refer Slide Time: 11:54)

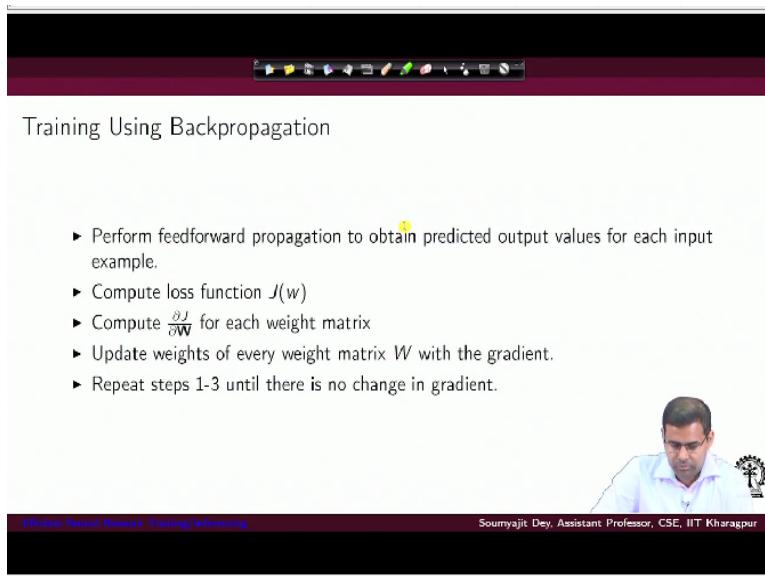


(Refer Slide Time: 11:58)



Yeah.

(Refer Slide Time: 12:00)



So that is essentially what we call training using backpropagation. So, the first step is you perform feedforward propagation to obtain the predicted values, you obtain the predicted values using those predicted values, you compute the loss function, then you compute the value of, of course, let us understand what is the relation, given the X, you need to compute the Y. That is the forward pass. Why do you really want to do that?

Unless you get the Y's that your network estimates you cannot really figure out what is the error that means what is the difference between your estimate and the actual values that are there for the

outputs in that data set. That is why you need the feedforward propagation only then you can do the loss function value estimation. And then you compute what is the value of $\frac{\partial J}{\partial W}$ for the current weight values.

Once you have computed this, you tried for each of the weight matrices, you have to do the reestimate. So that is the face of updating the weights for every weight matrix with the gradients. Once that is done for these modified values, you again do the entire thing. That means computing the loss function and all that.

(Refer Slide Time: 13:16)

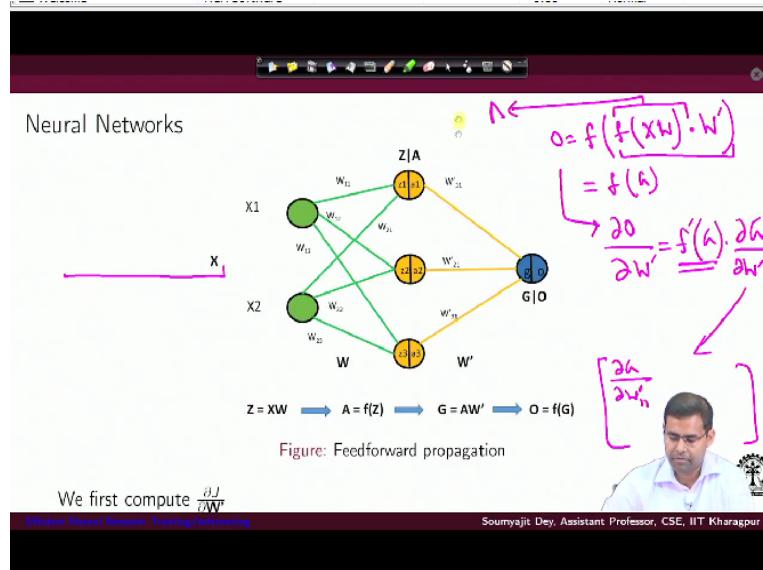
$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial w_{11}} & \frac{\partial J}{\partial w_{12}} & \frac{\partial J}{\partial w_{13}} & \cdots & \frac{\partial J}{\partial w_{1n}} \\ \frac{\partial J}{\partial w_{21}} & \frac{\partial J}{\partial w_{22}} & \frac{\partial J}{\partial w_{23}} & \cdots & \frac{\partial J}{\partial w_{2n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial w_{d1}} & \frac{\partial J}{\partial w_{d2}} & \frac{\partial J}{\partial w_{d3}} & \cdots & \frac{\partial J}{\partial w_{dn}} \end{bmatrix}$$

The dimensions of the weight matrix and its gradients will be the same.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So let us see, what is the underlying matrix operations that are going on. So what is $\frac{\partial J}{\partial W}$, when you express this in terms of components, as we have seen for the capital W, it is essentially having all these components . So let us go back to our nice small neural network to see the components of W.

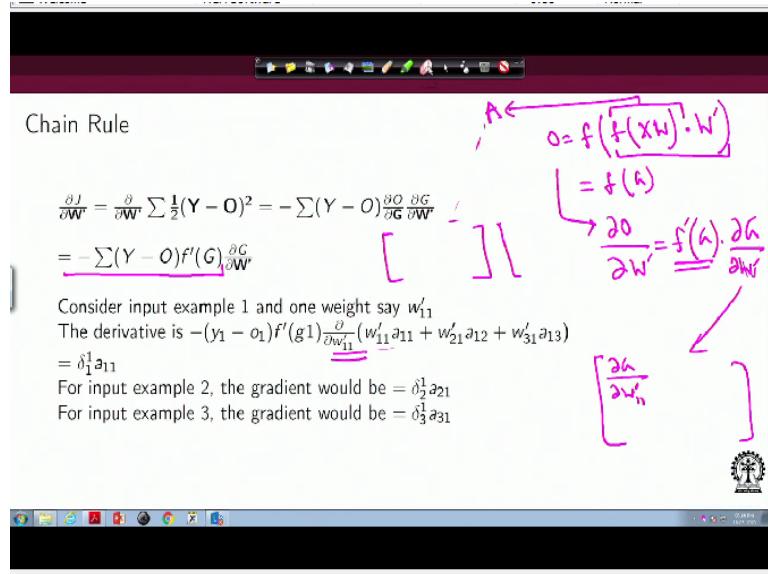
(Refer Slide Time: 13:34)



So, these were my components for a simple 3 cross 2 system . So, you have this W_{11} , W_{12} W_{13} , again W_{21} and W_{22} W_{23} . So in that way you have like this. So these are the W_1 components that are W_2 components like these W_d components like that. So with this we will first see what is going on in the feedforward propagation and that is what we have. So, as we have seen in the feedforward propagation, what we get is this output, output is nothing but activation inside this f again on the input followed by W multiplied by W' .

And then again the activations use my feedforward propagation . Now, using output I need to figure out what is my J and then do and then we have to do the computation of $\frac{\partial J}{\partial W'}$

(Refer Slide Time: 14:36)



So, what is this del J del W prime is nothing but summation like this. So, of course, what we get is del J del W prime summation 1 by 2 and we have this Y minus O square. Essentially this is my mean square function where we are talking about. And then of course, what is going to happen is you are taking the derivative with respect to W prime okay first we are trying to figure out what is del J del W prime and then we will also figure out what is del J del W .

So, here as you can see this will give us something like this square term will come down and we will have the summation followed by the output minus the estimates. The estimates and then you have this finally, we want the derivative with respect to W prime. So, you will now start applying this W prime on the O because Y is that some actual value that you have observed .

So, this is the value that is really there in your test data. And this is the value that you have actually observed . So, y minus O is something that you have as an output of the feedforward propagation. But the next thing is what you do this, you get the functional form because you are trying to do a differentiation of del O del W prime. So that gives you del since O is nothing but is basically del O if we just write it in terms of the intermediate steps. So O was like some f G . Because we were calling this entire thing G in our expression. So accordingly we write it as del O del G del W prime.

So now as you can see, this is the form that we get. No, but we just represent since I am doing del O del W prime, that would naturally give me f prime G followed by del G del W prime. So what

is f' prime G is nothing but a derivative of the activation function. Now, that is easy to compute, because for the activation function, we are considering simple differentiable functional forms. So it is nothing but the differentiable some first derivatives of that functional form.

So the trick really is part of the last component which is $\delta G \delta W'$, and that is something that will prevail with. Now, for the big neural network or all the components there will be many terms. Let us take as an example, just the first term. So, for our input example, let us just consider the first weight which was W_{11}' because as you can see, this $\delta G \delta W'$ is going to have many components .

This is going to have many components just like $\delta J \delta W$, this $\delta G \delta W'$ will also have a lot of components, and we are just trying to see what should be the first component. So, considering W_{11}' , which would be one of the components. So, $\delta G \delta W_{11}'$ that is what we are going to get because well, what is G . So, as you can see G is nothing but some value multiplied by W' .

What is this value, $f(XW)$. If you look back, this was nothing but my A . Something that we actually computed here. Yeah, is so A is $f(XW)$. And that is a value that we already have, which is components . So then this reduces too. So I am just considering one of the components and it will reduce to $W_{11}' A_{11} + W_{21}' A_{12} + W_{31}' A_{13}$, so on so forth.

Why are we really saying that let us elaborate it out. So, we are considering this $\delta G \delta W'$ and as we have seen W' will have different components. Now, we are considering one component which is $\delta G \delta W_{11}'$ and inside G what we have is this A which is nothing but $f(XW)$. And considering that we have already evaluated A and that A has got these values.

So, what are the points at which A is going to interact with W_{11}' . Well, that would be only the first point. So, just to make it more clear here, how does A really look like because we are considering A followed by W' . So, A will have a matrix form and then we have W' with all those values . And then if we just take the first part, then I have W_{11}' multiplied by $A_{11} + W_{21}' A_{12} + W_{31}' A_{13}$ like that.

And as you can see if I am applying the partial differential with respect to W_{11} prime only, then I will just get the first component of A , which is A_{11} all, sorry, we will just so, if we take the derivative and only consider first component, we get this kind of a value of A_{11} and the previous part is same because that is not going to change. Now, similarly for the other inputs examples, if we consider the other weights .

Then similarly, we will be able to figure out the other components where some scalar multiplied by A_{21} some scalar multiplied by A_{31} and all that . Now, as you can see that all these A 's are something we already have available as part of the feedforward computation, but the part that is coming out here is this delta 1 matrix . Because as we can see for del 1, we have these components delta 11 delta 21 delta 31 like $\delta_{11} \delta_{21} \delta_{31}$ like this, where this delta 1 is nothing but this part.

So, just like for $y_1 O_1$ and $f G_1$ we have delta 1 first component. Similarly we can compute the other components.

(Refer Slide Time: 21:58)

$$\frac{\partial J}{\partial W'} = \begin{bmatrix} \frac{\partial J}{\partial w'_{11}} \\ \frac{\partial J}{\partial w'_{21}} \\ \frac{\partial J}{\partial w'_{31}} \end{bmatrix} = \begin{bmatrix} \delta_1^T a_{11} + \delta_2^T a_{21} + \delta_3^T a_{31} \\ \delta_1^T a_{12} + \delta_2^T a_{22} + \delta_3^T a_{32} \\ \delta_1^T a_{13} + \delta_2^T a_{23} + \delta_3^T a_{33} \end{bmatrix} = A^T \cdot \delta^1$$

This can be expressed as $A^T \delta^1$

And if we consider it like this, then the overall representation that we will get will be something like this . Just like we have the delta J delta W prime. From that we have figured out what is the functional form considering only the way to W_{11} prime and the example one. If we keep on

considering the other examples, overall, the functional form that we will have for del J del W prime would be something like this .

Because w prime is going to have these different components like W_11 prime, W_21 prime, W_31 prime like that. And of course, there will be other elements here. And for each of them, we have to figure out this . this delta 11 a 1 delta 2 1 and delta 12, a 21 delta 13 a 31. Like this. Yeah. So, I hope this is clear, because as we can see for w prime, these are my components W_11 prime W_21 prime, W_31 prime.

And all we are doing is for the overall partial derivative, we are finding each of these terms component wise. And as you can see for each of these terms, if we consider each weight, each input example separately, we get the component terms for the derivatives. And overall, this is what we arrive at . Now, one way to write it would be because as you can see that we have the entire matrix here in a transpose form.

So, let us say it is just nothing but A transpose delta 1. I hope it is clear why we are writing a transpose A delta 1 because here as you can see, for the different components, we are essentially getting A in the transpose for now, just like we computed del J del W prime, we can do a similar thing for del J del W.

(Refer Slide Time: 24:23)

Chain Rule

$$\begin{aligned}\frac{\partial J}{\partial W} &= - \sum(Y - O) \frac{\partial O}{\partial G} \frac{\partial G}{\partial W} \\ &= - \sum(Y - O) f'(G) \frac{\partial G}{\partial A} \frac{\partial A}{\partial W} \\ &= \sum \delta^1 W^T \frac{\partial A}{\partial W} \\ &= \sum \delta^1 W^T \frac{\partial A}{\partial Z} \frac{\partial Z}{\partial W} \\ &= \sum \delta^1 W^T f'(Z) \frac{\partial Z}{\partial W} \\ &= X^T \delta^1 W^T f'(Z) \text{ (Derive!) } \\ &= X^T \delta^2\end{aligned}$$

Annotations:

- $J = \frac{1}{2} (Y - f(f(XW) \cdot W'))^2$
- $O = f(G)$
- $G = AW'$
- $A = f(Z)$
- $Z = X \cdot W$
- $\frac{\partial Z}{\partial W} = X^T$
- $\frac{\partial A}{\partial W} = \frac{\partial A}{\partial Z} \cdot \frac{\partial Z}{\partial W}$
- $\frac{\partial G}{\partial A} = \frac{\partial G}{\partial Z} \cdot \frac{\partial Z}{\partial A}$
- $\frac{\partial O}{\partial G} = \frac{\partial O}{\partial Z} \cdot \frac{\partial Z}{\partial G}$
- $\frac{\partial J}{\partial W} = \frac{\partial J}{\partial Z} \cdot \frac{\partial Z}{\partial W}$

So, what about $\delta J \delta W$. So, again, we will do a derivative . So, this one square has come out . So, minus summation $Y - O$, now, you have $\delta O \delta G \delta G \delta W$. So, if we just elaborate the left hand side here. So, essentially what you will get is J is summation half $y - I$ am just writing the O part in the expanded form . So that is what we have here. And then when I take the derivative with respect to G .

So what do we get $\delta O \delta G$ because I am taking the derivative of $\delta O \delta W$. So, I will make it $\delta O \delta G \delta G \delta W$. So, again what we can do is we have $\delta O \delta G$ and we have O is nothing but $f(G)$. G is nothing but AW prime as we have seen, and this A is nothing but $f(Z)$ and Z is as we have seen the original input, multiplied by the weight of the first layer. So that is the setting we have.

So as you can see, when you $\delta O \delta G$, that is nothing but is going to give you the f' which is essentially the derivative of the activation function of the hidden layer. And then you have to perform this $\delta G \delta W$, which you have broken down into 2 parts. So you have to do $\delta G \delta A$, that means this expression, so what is really that going to give you. So, if you do just by using normal linear algebra relation, I have G equal to AW prime.

If I take a derivative, $\delta G \delta A$, that is going to give me W^2 prime transpose and that is what we get. So we have the derivative here. Since G is AW prime. So δG I am sorry, $\delta G \delta A$ is going to give me W prime transpose using standard linear algebra rules. So that is what we get and then I have $\delta A \delta W$. So now I will break this $\delta A \delta W$ further to $\delta A \delta Z$ and $\delta Z \delta W$. So what is $\delta A \delta Z$.

As you can see, A is equal to FZ again, I have an activation function and $\delta A \delta Z$ is going to give me the differentiation of that activation function . So that is again f' of Z . And then I have $\delta Z \delta W$. So what is that? So again, for Z what do we have is $X \cdot W$. So naturally again, using standard linear algebra rules for products of matrices. If I do $\delta Z \delta W$, we are going to get is X transpose.

So that is what we have the X transpose. But the question is where did this so, we have solved for this part, how the W prime transpose comes, how the X transpose comes, how the f prime Z comes. But what about delta 1. Well, if you look back into our previous derivation, what was my delta 1. If you look here, delta 1 is minus summation, this thing , y minus O times the f prime of G.

So just apply it here. So in that way, when you get this minus summation Y minus O f prime of G, that reduces to your delta 1. So you already have it here. And then when you do del Z del Z del W, that is going to reduce to your X transpose. And this is your final form. Here, you replace delta 1 with W prime transpose and f prime Z using a new constant which is delta 2.

So, we are just trying to figure out how to derive this del J del W and this would be our expression fine. So, once we have figured out that will for del J del W, we have a representation in terms of A transpose del 1. That is my del JW prime sorry. And then for del J del W, we are every presentation, which is X transpose delta 2, we are going to use these values because they are giving me the gradients of J with respect to W.

And I will just use these gradient values to adjust the value of both W and W prime. And that is what happens in case of back propagation. So, through after this forward pass, I have got output values, I have computed the loss function, I have applied the derivatives, I have figured out using these relationships that we have established from here, we have tried to figure out how you get del J del W prime and del J del W. And then you use those gradient fellows to adjust and compute to reestimate the values of W and W prime. And that is something that you know, back propagates.

(Refer Slide Time: 31:17)

The slide has a dark header bar with icons. The main content area has a white background. At the top left, it says 'Summary'. Below that is a bulleted list:

- ▶ Feedforward propagation can be performed by a series of linear and non linear transformations involving matrix operations starting from the input layer.
- ▶ Backpropagation also involves a series of linear and non linear transformations involving matrix operations starting from the output layer.
- ▶ Each operation and transformation exhibits parallelism and scope for optimizations using a GPU.

At the bottom right of the slide, there is a small video player showing a man in a blue shirt speaking. The video player has a progress bar and some control buttons. Below the video player, there is a caption: 'Efficient Neural Network Training/Inference' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

So as a summary, what we are really doing is we are re-computing the values of W and W prime and propagating that backwards through the neural network and again doing the forward pass that is computing the entire classifications output by using these new values of W and W prime. So, the way you will, if we just try to summarize the way the neural network training was working is.

You have a feedforward propagation which is performed by a series of linear and nonlinear transformations involving matrix operations starting from the input layer, and then you have a back propagation which involves again a series of linear and nonlinear transformations involving matrix operations starting from the output layer. What do we mean. See, we have gone through the forward propagation that gives me the value of delta 1.

You now have to apply delta 1 to this matrix to get what is delta 2 . Because we have because you are going to use delta 12, to get a new value of this W prime. Because you have an original W prime, you have to subtract a transpose delta 1 first of all you compute delta 1 after doing the feedforward propagation, and then you readjust W prime with this relation.

Once and after that you also do what is like computing what is delta 2 because as you can see that computation of delta 2 clearly depends on delta 1. That is why we have a sequence here. After delta 1 is done, you can reestimate W prime using delta 1, you compute delta 2. Once you have

computed delta 2, you reestimate the w values and set all those values again in the neural network. So that is your back propagation.

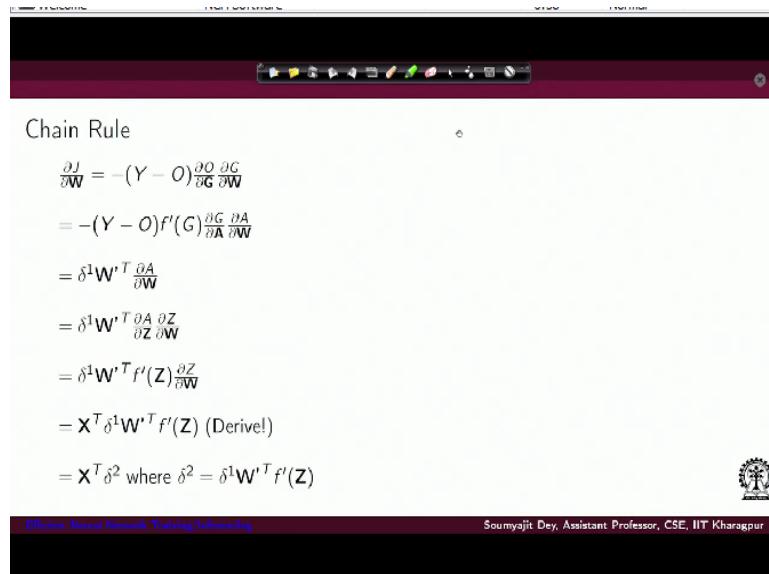
You are propagating back the values, readjusting values of W prime and W. So that is again a series of linear and nonlinear transformations but they start from the output layer. Now, as you can see that these operations actually have significant amounts of computations like transpose, computation like matrix multiplication, things that we have already seen, parallel systems like GPUs are very smart at doing . So that is why we will see how GPU architectures can be exploited for doing all these computations. With this will end the current lecture. Thank you for your attention.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-57
Efficient Neural Network Training/Inferencing (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming.

(Refer Slide Time: 00:29)



The screenshot shows a presentation slide with a dark header bar containing icons. The main content is titled "Chain Rule". Below the title is a mathematical derivation:

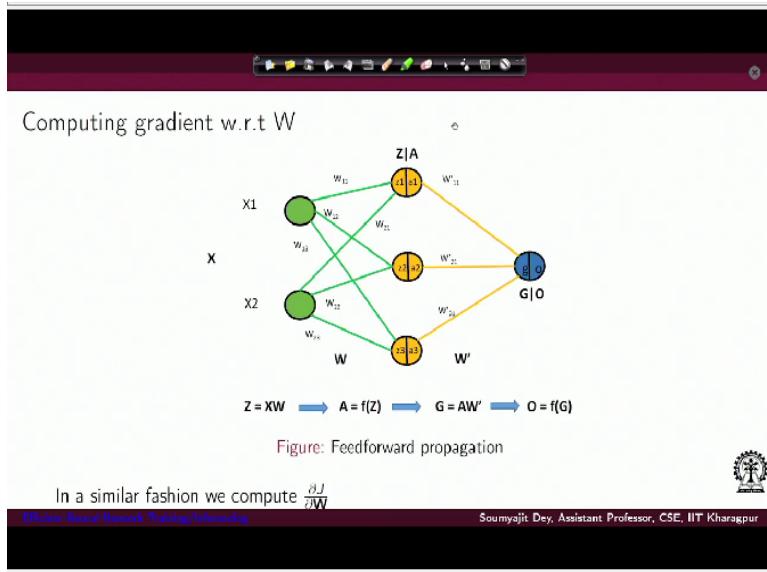
$$\begin{aligned}\frac{\partial J}{\partial \mathbf{W}} &= -(Y - O) \frac{\partial O}{\partial G} \frac{\partial G}{\partial \mathbf{W}} \\ &= -(Y - O) f'(G) \frac{\partial G}{\partial \mathbf{A}} \frac{\partial \mathbf{A}}{\partial \mathbf{W}} \\ &= \delta^1 \mathbf{W}^T \frac{\partial \mathbf{A}}{\partial \mathbf{W}} \\ &= \delta^1 \mathbf{W}^T \frac{\partial \mathbf{Z}}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial \mathbf{W}} \\ &= \delta^1 \mathbf{W}^T f'(\mathbf{Z}) \frac{\partial \mathbf{Z}}{\partial \mathbf{W}} \\ &= \mathbf{X}^T \delta^1 \mathbf{W}^T f'(\mathbf{Z}) \text{ (Derive!) } \\ &= \mathbf{X}^T \delta^2 \text{ where } \delta^2 = \delta^1 \mathbf{W}^T f'(\mathbf{Z})\end{aligned}$$

At the bottom of the slide, there is a footer bar with the text "Efficient Neural Network Training/Inferencing" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So we will just start with a small ((0)) (00:30) of our coverage in the last lecture. So, one was regarding this chain rule. So, if you just check, we have been earlier talking about this y's in vector notation, but since we are having a summation here that would mean we are doing a summation over the components errors. So, that should essentially be small y i's and also continuing further.

One other issue would be like here in the chain rules derivative since we are considering capital Y's which are the vectors. So the summation would really not be there.

(Refer Slide Time: 01:06)



Why so if we remember that we have already discussed how we do the forward pass, and how are the derivatives related, how to compute the derivative of the loss function with respect to the weight matrices w and w' , and how those are really used to figure out this value of the derivative that is in the second stage, it is this X transpose delta 2 where I can also have the delta 2 expressed as the value of delta 1.

And w' transpose and then the activation functions gradient at those values. So, with this, I believe we had already discussed what was a backpropagation rule that in the backpropagation pass you re-compute w' , so you re-evaluate using your computation of delta 1 and delta 2. You are essentially re-computing w' to be w' minus A transpose delta 1 and then you are also using delta 2 to figure out what will be the updated value of w . So, that would be w minus X transpose over delta 2 like that.

(Refer Slide Time: 02:10)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Summary". Below it is a bulleted list:

- ▶ Feedforward propagation can be performed by a series of linear and non linear transformations involving matrix operations starting from the input layer.
- ▶ Backpropagation also involves a series of linear and non linear transformations involving matrix operations starting from the output layer.
- ▶ Each operation and transformation exhibits parallelism and scope for optimizations using a GPU.

At the bottom of the slide, there is a footer bar with the text "Efficient Neural Network Training/Inference" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So, like this was our summary, that whenever you are doing the feedforward propagation, it is a series of linear and nonlinear transformations, essentially all in matrix operations. And during the backpropagation, you also have a series of such transformations. Again, that those can be , but they start from the output layer towards the frontal layers. So, that is why it is backpropagation. And we want to do all those things using the parallelism that is available in the GPUs.

(Refer Slide Time: 02:37)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Building a DL Library". Below it is a bulleted list:

- ▶ Provide constructs for specifying a network.
- ▶ Provide efficient routines for feedforward, backpropagation and gradient computation.
- ▶ Provide routines for training and testing.
- ▶ Should support parallel and distributed processing for the computation passes.

Below the list, there is a note: "DL Libraries like Tensorflow and Theano use a Computational Graph Abstraction for encoding a neural network." A small portrait of Soumyajit Dey is visible in the bottom right corner of the slide.

At the bottom of the slide, there is a footer bar with the text "Efficient Neural Network Training/Inference" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So, with this background, in case you are interested in building a deep learning library, so that would mean your library should have to support the following things that means it should provide constructs like how somebody can easily specify the neural network architecture, what kind of

efficient routines, I can implement as part of the library, so that I have feedforward backpropagation and gradient computation these features available.

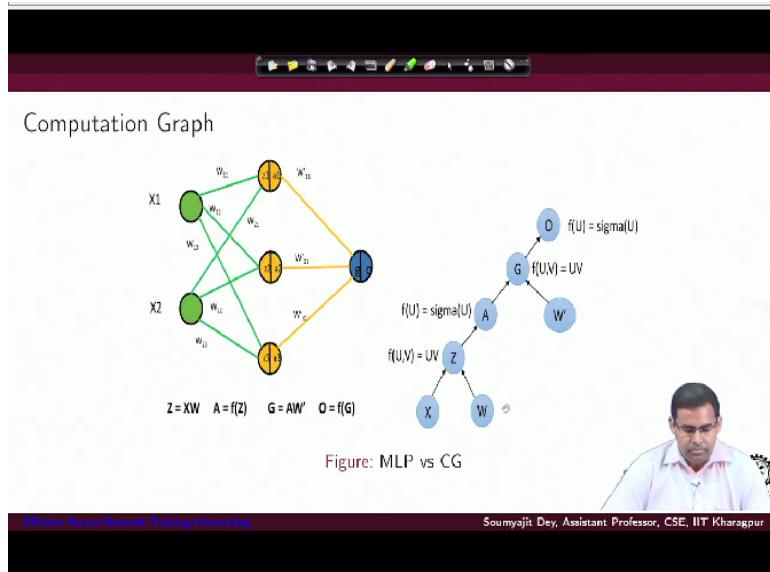
So they should just be available as function calls when I am building a complex neural network architecture and going to implement it for doing all these computations. So, of course, and using these routines, I will like to build high level routines for doing that neural network training as well as testing processes . Moreover, as we can understand that the library we like to exploit HPC resources.

So, all the operations have to be implemented, these operations of feedforward backpropagation gradient computations, they have to be implemented using suitable data parallel languages like CUDA or open CL or something else. And it should really be configurable. That means once I tell that this library should be running in this kind of an architecture, it should be explored.

To figure out well, let us say I am trying to say that I want this library to run in a CPU GPU system, where the GPU has this nice is a sense this many espies and all that, then I should follow the standards we discussed earlier that how to really define the architecture, how to write your program in such a way that all the variables get configured based on the architecture or resources that are available .

Something like this is already available for us. Many of us have already heard about this popular deep learning libraries like Tensorflow and theano. And they essentially follow all this , standards and discussions we have just done. But at the high level, they provide an abstract view of the neural network, they provide what we call as a computational graph abstraction, which helps you to encode the different parameters and the connections that should be there in a very complex neural network architecture.

(Refer Slide Time: 04:56)



So we are just trying to provide an abstract view on the left hand side, suppose you have this multi layer perceptron like we have discussed earlier. So, you have the input that input X vector is getting transformed with W and then you get something called this intermediate value that is a Z which gets transformed with the activation function, you get something called A, these are all intermediate values that you are computing.

Then again you have to do further transformation on A using these weights of W prime that gives you G and on this you apply the final activation function f again to get the output, like , we have to understand that this is the neural network computation which this architecture will do. And we like to encode it in the form of a data flow graph. So, that is the usual way in which a computer scientist likes to encode his computation.

So, it is basically like you have notes which kind of represent the compute input data or some transformation that has to be done . And then what the output of the transformation is provided to another node, where some further transformation will be done. So as you can see, on the hand side, we have an example computation graph where we have 2 nodes X and W which provide the input data.

And these data flows following these edges to this node, where I have computed the value Z and this node is level like this, $f(U, V)$ is UV . So, essentially I am trying to say that this node performs

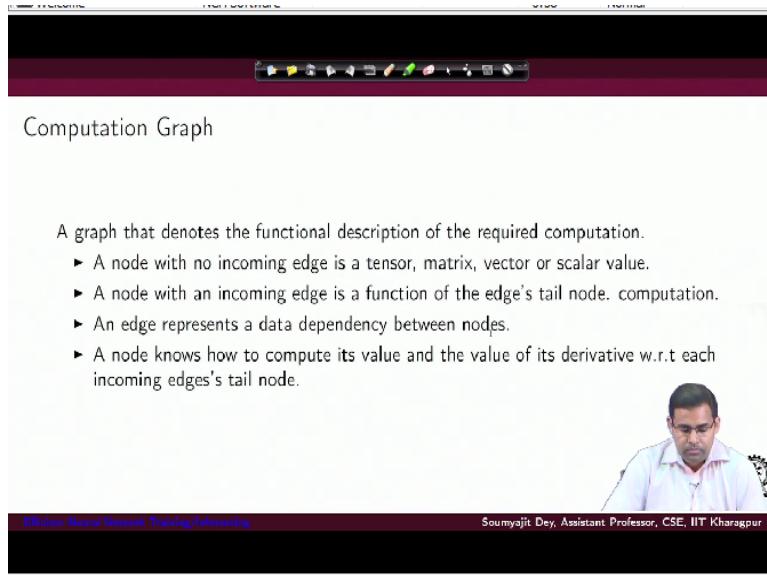
the following transformation. That is, it takes the left and operands and does a matrix multiplication, then the output will flow to another node I call that output as A . So, this is dependencies further captured by this arrow from node Z to node A.

And here how do I really get A. So the computation that will provide me A is again, denoted here using this function, f of U. So I am just saying that the function takes whatever argument, let us call it U, the output is sigma of U, sigma is an activation function here. And then again, these output A and some other input node W prime, these 2 inputs flow to give me another output G.

And this output is derived by considering the corresponding function levelling, where I am saying that this function is f which again takes 2 parameters and does their multiplication. So again, A and W prime gets multiplied to give me G. And then again, I have another activation function giving me the final output which is O. So the convention we are following here is every node has the data representation.

And following the dependency as data flows to an output node and the way I get output node is by a suitable function transformation and the function is provided as a level . So, this is how we are trying to signify how the computation, how the data flows through this transformation network in the form of a computation graph. So, better to say is just another way to represent it, where we are making the dependencies explicit.

(Refer Slide Time: 08:27)

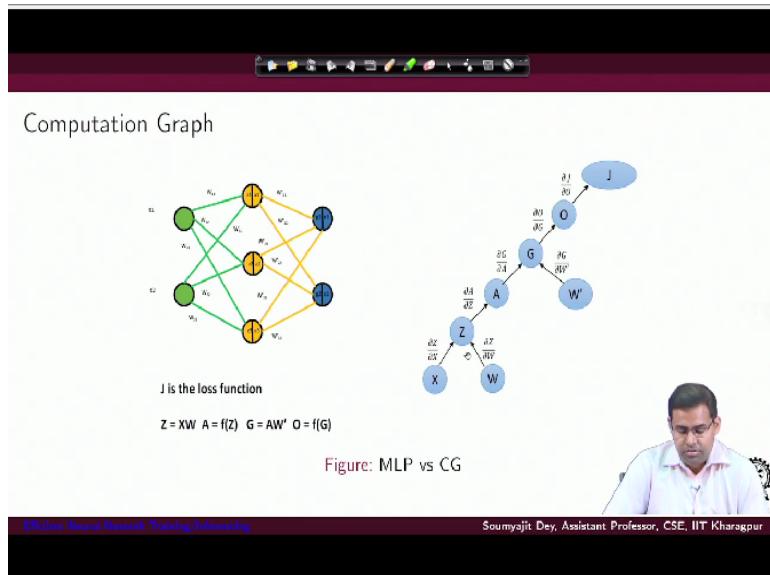


So, just to summarize this is a graph that will denote the functional description that I required for the computation. A node which has got no incoming edge is a tensor or it can be a matrix or a vector or a scalar value. A node which has an incoming edge is a function of the edge's tail node computation. The edge represents a data dependency between the nodes like we saw earlier, the edge just tells me that the data will flow in the direction of the edge.

And as long as the data is not available in the input nodes, the edges leading to some output node there, the computation is not going to happen. And node knows how to compute its value and the value of its derivative with respect to each incoming edge's tail node. Now that is important. So every node knows how to compute this value. And that is known by the associated function.

Here this node knows that it has to apply this multiplication function. And also, it knows how to compute the value of its derivative with respect to each incoming edge's tail node.

(Refer Slide Time: 09:36)

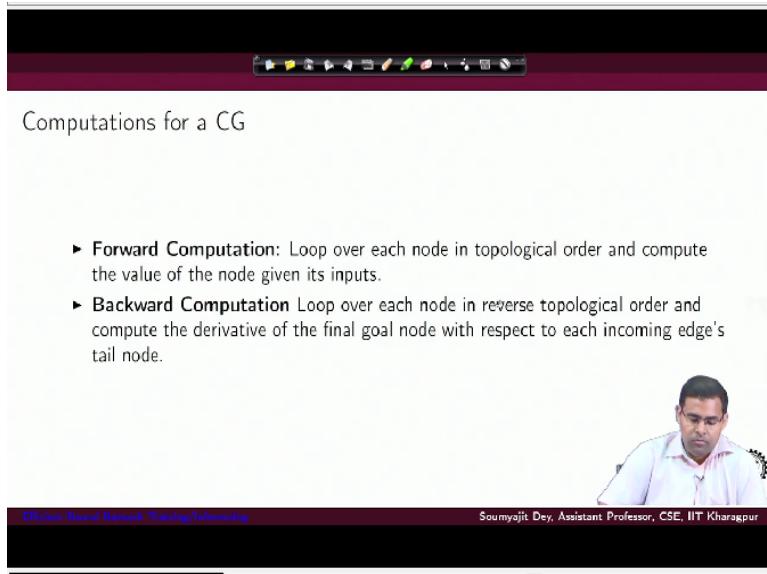


So if asked, this node can compute the derivative of Z with respect to W okay. Now, this is the next thing that we will use. So considering J as the loss function, again, let us recall that we have this computation going on. So X multiplied by W is going to give me Z . We then compute f of Z , which gives me A . We use A and W' prime to generate G , then we compute , apply f on G to get O .

And finally when we are using O , I compute the loss function. And as we have been discussing that every node knows, since the dependency of the node is always I can figure out the dependency by looking into the incoming edges. So for every node, I can compute its derivative with respect to the variables in these incoming edges, corresponding nodes. So, given Z , I can figure out this at dW are tracking backwards.

Let us say I want to compute some overall derivative, I can follow the derivatives chain rule as we know, and using the chain rule, I can compute each of the individual derivatives and compose them. So we will see an example. But before that, all I am trying to say here since I have this kind of a dependency, for every node, I can figure out let us say from J to O there is a dependency I can compute $\frac{\partial J}{\partial O}$. Similarly, I can compute $\frac{\partial O}{\partial G}$, $\frac{\partial G}{\partial W'}$, $\frac{\partial G}{\partial A}$, $\frac{\partial A}{\partial Z}$ so on so forth.

(Refer Slide Time: 11:12)

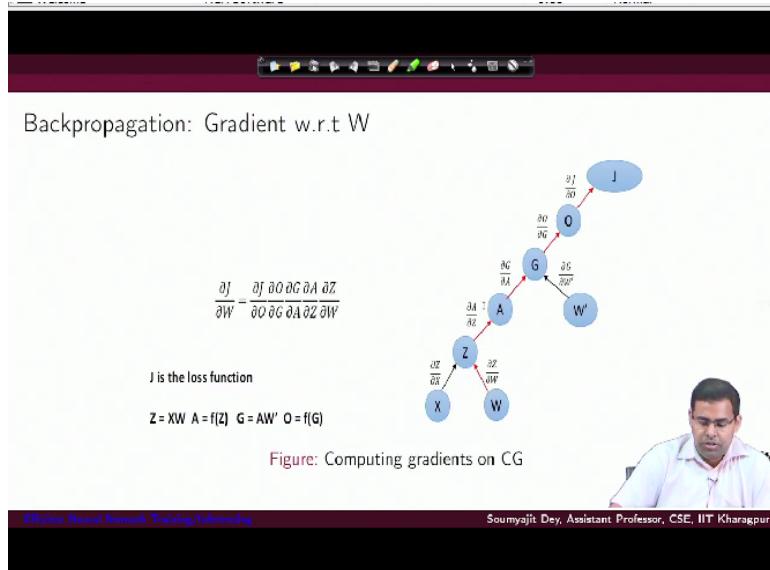


So, whatever is in the forward computation for the computational graph, so, it simply loops over each node in the topological order and computes the value of the node given its inputs, so, as we know that the topological order is nothing but a linear representation of the node ordering of a directed acyclic graph . So, essentially I can compute a topological order for any such graph where the node is dependent on some previous node.

That dependent node will come later on in the order . So, following the topological order, I can simply loop over the graph and compute the corresponding outputs. And then for backward computation, what am I supposed to do if you remember what we did in the neural network, you just went back from the output layer to the input layers , essentially the same thing. So, you loop over the nodes in the reverse topological order.

And you want to compute the derivative of the final node with respect to each incoming edge's tail node. That means you do this computation, you compute $\frac{\partial J}{\partial O}$, $\frac{\partial O}{\partial G}$ and like that. So, this is how I can do the forward and backward computation.

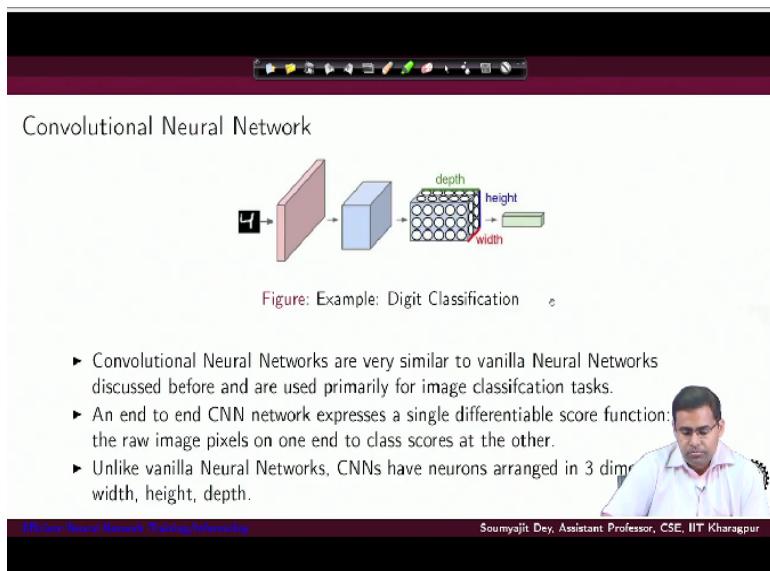
(Refer Slide Time: 12:27)



But how does that really help, our overall objective is to compute this $\frac{\partial J}{\partial W}$. So, as you can see, we have been discussing earlier that I can follow derivatives chain rule. And since each of these expressions would be available to me, because I have these dependencies for each node, I know their values and all that. So, I can just compute each of these individual derivatives and compose them to get what is the overall derivative of the loss function.

Similarly, I can also compute the derivative of the loss function with respect to W' and that follows these red mark arrows .

(Refer Slide Time: 13:05)



So, essentially this is how the computational graph abstraction is used to specify a neural network for a corresponding deep learning library. And then given suitable implementation of the different propagation passes that will be available in the library, the computation really goes on. So, from this point, let us switch to another alternative neural network architecture, which is better known as convolution neural network.

Well, this is very similar to normal neural networks that we have already discussed, but they are kind of also specialized in that their architecture is tuned towards image classification tasks. So, what is special about image classification tasks. Let us understand that an input image when we know that it is an image, it always is going to form a very regular structure, it is going to be a multidimensional matrix, where each of the pixel values are encoded for corresponding to different channels of colors .

And given that an image has got certain of intrinsic properties, the convolution neural network makes use of those properties to decide these architectural primitives and end to end convolutional neural network will express a simple differentiable score function, it will essentially map the raw image pixels from the input to class course at the other. So the class code can be many things , so you are essentially letting us say you are trying to recognize some numerals .

So essentially, you want this number 4 to be recognized, and you have many output possibilities. 10 digit possibilities. So the class for the digit 4 should get the highest value. And that is how you want this neural network architecture to work and assign the class code . But the way it differs is the CNNs since they are going to be tuned towards processing images only, they are arranged in some specific ways.

For example, when you are capturing an image, typically the arrangement even for a 2D image, your data structure shall be a 3 dimensional thing. Essentially, you should have the pixel values corresponding to height and width coordinates and also for each coordinate pair of height and weight, you are going to have 3 channels and you have to have memberships for all those color channels .

So, you have the input standard available in a three dimensional standard. And corresponding to that, the CNN will have neurons arranged in three dimensions as well. And also there is something important that, in general, when people dig up CNNs, they also make use of facts like well, the neural network architecture did not have full connections at every point like each intermediate node can represent a filter, which is tuned towards gathering a specific kind of information and it may not be gathering it from all parts of the image versus specific parts of the image and all that.

All you are trying to say is that once we decide that the input is an image, and these are the features of the image I am looking for, accordingly, I can decide weights and accordingly I can also decide the connectivity of the intermediate nodes.

(Refer Slide Time: 16:43)

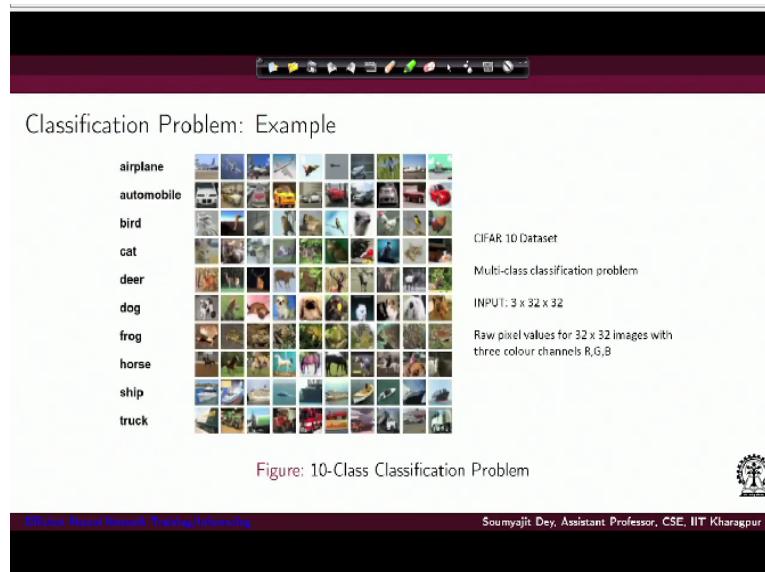
The screenshot shows a presentation slide with a dark header bar containing icons. The main title is 'Convolutional Neural Network'. Below the title, the text states: 'A CNN is typically made up of the following layers' followed by a numbered list: 1. Input Layer, 2. Convolutional Layer, 3. Pooling Layer, 4. RELU, 5. Fully Connected Layer. A note below the list says: 'The input and output for each of the layers 1-4 represent 3D image volumes. The fully connected layer is typically used at the end where the 3D image volume is flattened and fed as input.' At the bottom, there is a footer bar with the text 'Efficient Neural Network Training/Inference' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

So, holistically speaking, the CNN is typically made of the following layers, you have the input layer, after the input layer, you have these what we call as convolutional layers. For the convolution layer represent different image filters available as 3D masks which will be applied for mining information from the image. And then you have something called the pooling layer. Now this is followed by an activation layer typically that is taken as RELU.

And at the end you have what we call as a fully connected layer, which is just like the normal neural networks. The input and output for each of these layers to 1 to 4, they represent 3D image volumes.

So, up to the RELU layer, the input is a 3D image. After the convolution operation, I get 3D volumes. After the pooling, I will again get a 3D volume with some reduction in the length, the height widths etc. Even after RELU I have a 3D volume, but the fully connected layer is used at the end and the 3D volume is flattened, and it is provided as a linearized score.

(Refer Slide Time: 17:53)



So, for example, we will just show some examples here like which are popularly used data sets, for example, this CIFAR 10 data set. For that, I can have a multi class classification problem, which is pretty well known. I have input images of size 32 cross 32. Let us say I am trying to classify these images into these categories provided in the left hand side. And of course, I have 3 channels for red, green and blue. So, overall the in each of the input images is of the dimension 3 cross 32 plus 32.

(Refer Slide Time: 18:29)

Convolution Layer

- The Convolution layer is the core building block of a CNN and is computationally expensive.
- The input to the layer is a 3D image volume. The output to the layer is also a 3D image volume
- The dimensions of the output are dictated by three hyper-parameters - depth, stride and zero-padding.

 Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, the convolution layer, this is essentially the core building block of a CNN and this is a computationally expensive layer. And it works on the input images, which are 3D image volumes. And essentially, there are some convolution masks or filters which operate on these input volumes and provide an as output this layer is going to again produce a 3D volume. Now, the dimension of this output, they are dictated by 3 parameters. What is the depth of the layer at what stride it is going to work and the amount of zero paddings.

(Refer Slide Time: 19:09)

2D Convolution Example

Given a 2-D input image I and a 2-D weight filter(mask) W , the convolution operation slides the filter over the image, computes a neighborhood operation (weighted sum) over the elements of I and produces a 2-D output image.

$\sum \begin{matrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{matrix} = 3$

Will understand what it means. So let us take an example of a 3D convolution. But before that, let us just understand how this is again, , conforming to the basic neural network architecture. So instead of having inputs like X 1, X 2, etc., which we're discussing in our earlier things, essentially,

we were talking about test data or training or sorry, the training data or the test data depending on what kind of what you are doing, which will like vectors or matrices or tensors.

And essentially, the image is also following such a representation. For example, on the left hand side, we are considering a 2D image. But the thing is we are operating it with a weight matrix. And it happens to be that in this case, the weight matrix is basically an image filter or a mask and this and the way you choose the mask depends on what kind of operations you want to do on the image, what kind of information you want to take out of the image.

So here you have an image, here we have a 2D weight filter mask, let us call it W , because this is essentially like a matrix vector multiplication, you are doing matrix multiplication you are doing the image is getting transformed with this mask. But the operation you are doing, sorry, is a bit different from standard matrix, matrix multiplication. What you are really doing here is convolution operation, because that is how you operate filters on images.

What is convolution? How exactly do we convolution? What is the motivation behind doing that? As we know that these are standard image processing algorithms, and they are a standard transformation coming from signal systems theory, who is defined that will, by performing convolution with some specific filter, what information am I really going to get.

So we are really not looking into that whether but rather restricting ourselves to understanding that well how to implement a convolution. And the important point for us at this point is to understand that here, the weights do not do normal matrix multiplication, but rather that gets replaced with the convolution operation, which is the specific feature of convolution neural networks.

So, how does it really work? For example, if we take this 5×5 image, getting convolved with a 3×3 mask, essentially, this mask is going to slide over the possible locations on this image, and compute and output a 2D image like this, okay. (**Video Starts: 21:44**) So, let us try and understand how this is going to work. So maybe we will push in some values here. (**Video Ends: 22:01**)

So let us take it very simply, let us say it is all 1's. I am just writing some example values here. Now, when I do the convolution, what is really going to happen is in your output, the value at this location you are getting by applying this input mask on this possible location, so just overlay this mask over this part of the input image and then you do element wise multiplication. So what do you really get?

So essentially you are multiplying element wise. So the values that you are going to get are 0, then 1, then 0 then again 0 1 0 then again 0 1 0 , now just sum them up. So, you get a 3, so that is what you have here . So, is that simple. Now, this gives you the value of the 0 0th location of the output image. What about the next location? How do you get it? So, again just take the mask and overlay it, but by sliding it one part one position.

So essentially, you will be applying the mask on these locations. So, let us say these are my values. So again you can do the computation now. So all you get is . So, sum it up and that is going to give you a 0. So that is what you get. Similarly, you overlay the mask again here, fill up this entry. And then you downshift the mask by one position, fill up this entry, downshift, another position will have this entry shift one position, another position and downshift like that.

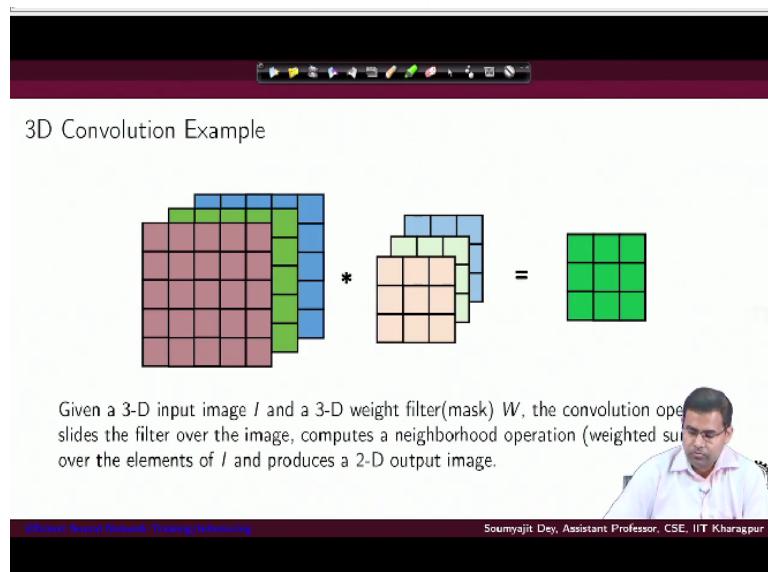
So in that way, you just compute all the values, given this 2D weight filter or the mask. So we can continue like this. (**Video Starts: 25:44**) So that is how you can build on this example. And here in this animation, we are just trying to show that for the output location, how the change of the **over** overlay plays a role when you compute the output values of the image. So as you can see, you have this 2D input image, you apply the 2D weight filter mask and you are getting a 2D output image.

So there is no change in the dimension. And of course, the dimension of this output image depends on 2 things, the dimension of the input image as well as the dimension of the mask. And in what is trying I am applying the operation. We will see what that means. So then, as you can see that for the first location, you are putting the mask in the left, top left position that is possible.

That is the overlay region, you shifted by one position and you keep on shifting like this in the output image. Move to the next row, again keep on shifting in that row, and so on so forth . This is how you do the 2D convolution. A standard image processing transform. All that is happening is we are now claiming this is getting done in the first layer or any layer which we are defining as a convolution layer of a neural network pipeline.

That makes it a specific pipeline and we call it a convolution neural network. All we are saying in some of the layers, we are not doing a generalized matrix multiplication, we are replacing it by this kind of image filter mask convolution operations to get a different to get and transform image vector fine. **(Video Ends: 27:33)**

(Refer Slide Time: 27:36)

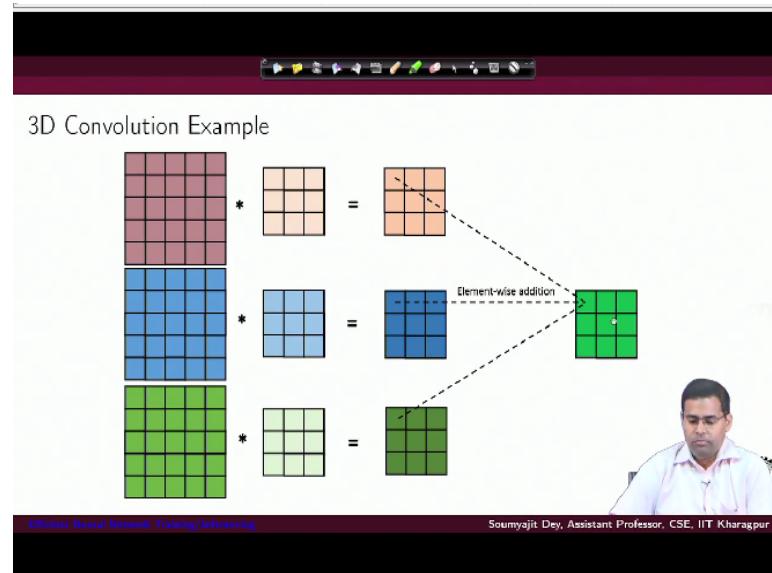


Similarly, this idea can hold for even 3D convolutions. So, just like we are considering 2D images and 2D masks, simply considered 3D image and convolving the 3D image so, there is a convolution we are denoting it here via 3D weight filter or mask W . So, what will happen here, the operation will slide over the image and compute the neighbourhood operation, the weighted sum. So essentially, the weights are getting multiplied with these values in the image that we saw.

And then finally, we are doing the summation and then you do it over the elements of i and produce the output image . So, essentially, you have a 3D input image, a 3D weight filter mask, and the convolution operation will slide the individual filters over the images and it will do the

neighborhood computation, like we discussed earlier already over the elements and produce the output image .

(Refer Slide Time: 28:43)



So here we are describing it further, what really is going on. So as you can see, these are on the left hand side we have the image and we are showing the different channels of the image. And then I have the 3D mask. So there are 3 2D masks essentially. So, that is what we have here; these are the convolution operation, and here we are explaining it with a broader diagram.

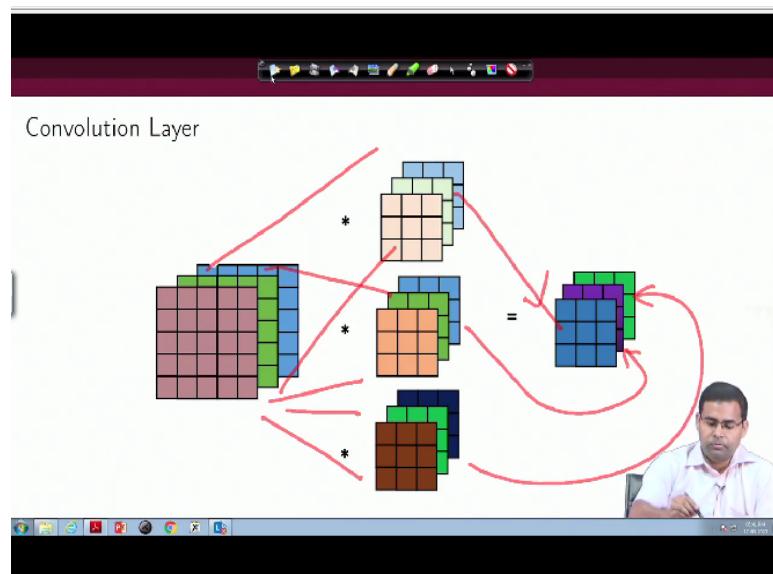
So, essentially you do pair wise convolution for each of the layers . So, you have the mask arranged with specific layers for specific image channels and you just do component wise convolution operation computation. So, this is for the first channel, this is for the second image channel, this is for a third image channel, you are just doing convolution for the corresponding mask layers.

And doing the operation like we have discussed earlier, you are taking this mask overlaying it over the image and then sliding it on both ways and computing each of the entries of the output image transformation. And how do you really compute after overlaying the mask you are doing element wise multiplication, essentially you are multiplying these image pixel information with the mask weights.

And finally doing a weighted sum of this region, again the weighted sum of the next region and so on so forth. And finally, you do element wise addition to just flatten this 3D outputs to a 2D image transformed output. So, here you are again doing element wise addition . So, first, you already have already done an element wise addition, which was also weighted by the mask for computing each of these entries.

And then for these 3 sets of 2D masks, you do individual element wise addition to get the values here, here, here, so on so forth .

(Refer Slide Time: 30:36)



Now, let us move forward.

(Refer Slide Time: 30:42)

Convolution Layer: Parameters and Hyper-parameters

- ▶ The Convolutional layer has a 3D weight matrix or an array of 2D weight filters which represent the learnable **parameters** .
- ▶ The dimensions of the output are dictated by four **hyper-parameters** - number of filters, filter dimensions, stride and zero-padding.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if we keep on doing it for multiple masks, because here we just considered a 3D image with one 3D mask, but in general, in a neural network architecture, I can have multiple masks . So, these are 3D images, there is one convolution to be done with one 3D mask, another convolution to be done with another 3D mask, another convolution to be done with another 3D mask. So, as we have seen just in the previous picture.

So, this convolution is going to provide me with one 2D output, this convolution is going to provide me with another 2D output. And again this convolution is going to provide me with another 2D output, so on and so forth. Now, we have already discussed the notion of hyper parameters. For example for our earlier normal vanilla neural network example. The hyper parameters were the different layers.

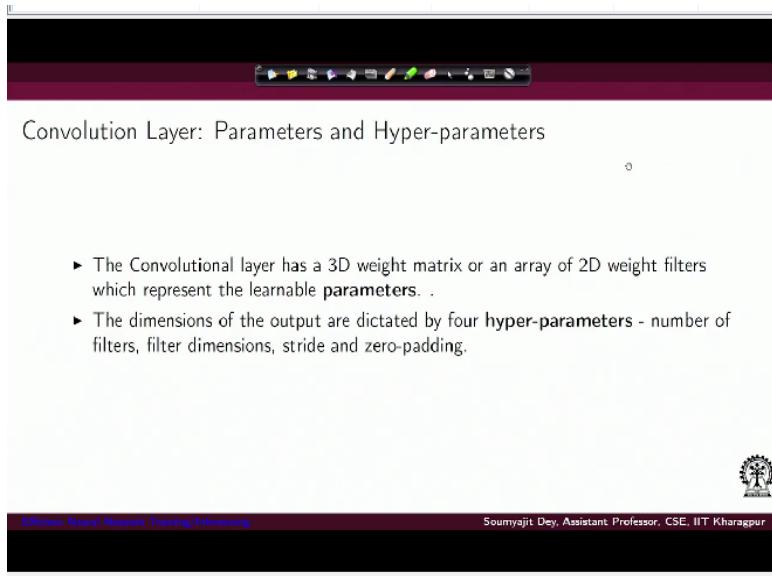
And the different set of the neural network that is essentially the architecture of the neural network, the weight matrices and all that. So it also decided what it should be in this case for the convolution layer. So, with this basic introduction to CNNs, we will end this lecture. Thank you for your attention.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-58
Efficient Neural Network Training/Inferencing (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming. So, in the previous lecture we have been discussing convolutional neural networks. And we are due to start with how to define parameters of the neural network.

(Refer Slide Time: 00:34)



The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Convolution Layer: Parameters and Hyper-parameters". Below the title, there is a list of bullet points:

- ▶ The Convolutional layer has a 3D weight matrix or an array of 2D weight filters which represent the learnable **parameters**.
- ▶ The dimensions of the output are dictated by four **hyper-parameters** - number of filters, filter dimensions, stride and zero-padding.

At the bottom of the slide, there is a footer bar with the text "Efficient Neural Network Training/Inferencing" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So, essentially, in a convolution layer, we have a 3D weight matrix or an array of 2D weight filters and they represent the learnable parameters like what would be the suitable values of these masks. And also the dimension of the output is dictated by these 4 hyper parameters which is what is the number of filters and the filter dimension stride and zero padding. So essentially the architecture of the CNN.

So again, we will just remember what we mean. So essentially, we are talking about how many filters to consider that it is like how many weight matrices to consider what will be the dimension of those filtered filters or the weight matrices. And then when we are performing the convolution, convolution also has an important parameter which is the stride of the convolution. And the other thing is the amount of zero padding that we want to do .

Because we will soon see how that affects the convolution operation. Now, where are the hyper parameters because essentially, your neural network architecture, how it really functions. It depends on this initial unknown quantity. Because it defines the architecture of the neural network.

(Refer Slide Time: 01:52)

Convolution Layer: Spatial Transformations

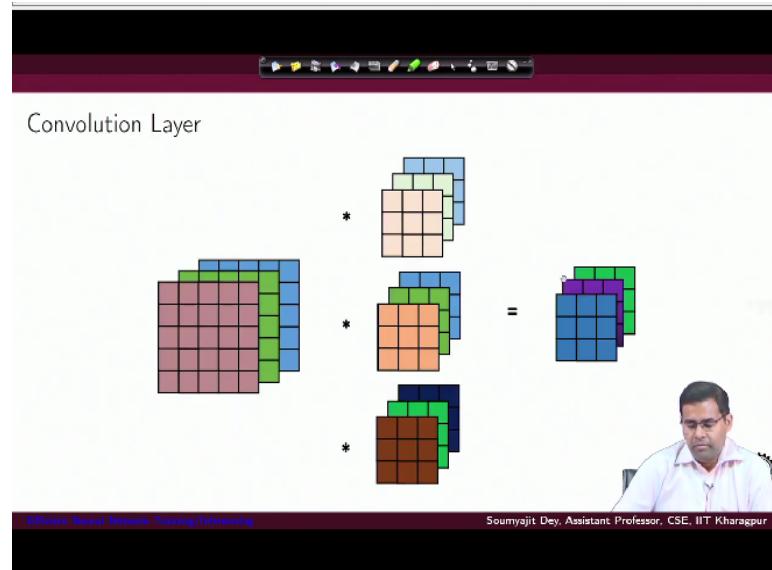
- ▶ The Convolutional layer takes as input a 3D image volume of dimensions $C \times H \times W$.
- ▶ The hyper-parameters for the layer are as follows.
 - ▶ M - Number of filters
 - ▶ F - Filter Size
 - ▶ S - Stride
 - ▶ P - Zero Padding
- ▶ The layer produces a 3D volume of dimensions $C' \times H' \times W'$ where
 - ▶ $C' = M$
 - ▶ $H' = 1 + (H - F + 2 * P) / S$
 - ▶ $W' = 1 + (W - F + 2 * P) / S$

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

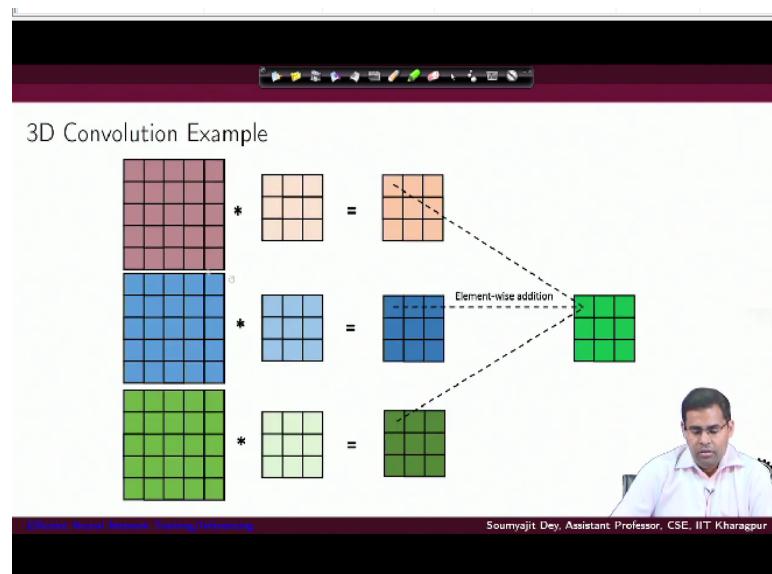
How many filters to take, what are the structures of the filters and finally, once you reset the hyper parameter set then you are trying to discover what will be the value of arguments inside those filters. And there are weight matrices, so that you get them loss function minimized. So, coming to the convolution layer. So, formally speaking, the layer takes as input a 3D image volume and the dimension is the number of channel cross height cross weight.

The hyper parameters let M represent the number of masks of filters, F represent the size of the filter, , it is an F cross F matrix like that, S is the stride, and P represents the amount of zero padding, we will soon see what that means. Now, what this layer is going to produce is going to produce a 3D volume of dimension, C prime cross H prime cross W prime. So as you can see there is a , in terms of dimension I have the channels for the input, they are now changing to C prime because the input image gets multiple different filters applied . For example, if you see here.

(Refer Slide Time: 03:17)

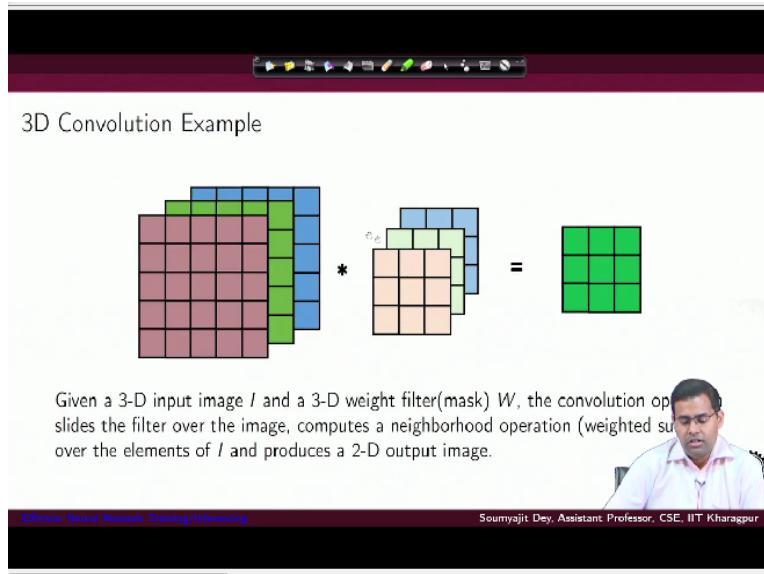


(Refer Slide Time: 03:18)



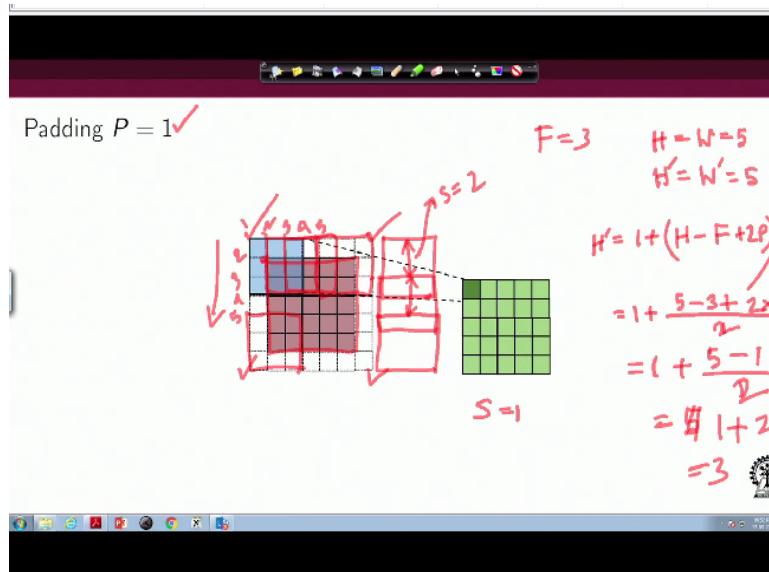
I have a 3D image, I applied a 3D mask, I got one output. Because essentially they are summing up to one mask .

(Refer Slide Time: 03:26)



So, the number of outputs that I really get depends on the number of masks I applied. For example, here if I apply 3 masks, each of them operating on 3D image, the 3D masks, each of them are going to give me 1 2 cross 2 output. So, for using the 3 masks essentially I get a 3D output like this . So, essentially C prime is M. There are a number of filters, but what is H prime. H prime is essentially the height that modified the height of the output. And W prime is the height and width of the output.

(Refer Slide Time: 04:02)



How to determine that. So, let us consider some examples to determine that and also there is some important parameter that is the zero padding, that how much of extra 0s should I surround the image with, so that I get to decide how many or what should be the output image dimension, we

will see what that means. So consider P equal to 1 that is the padding is 1, that means we have this as the original image.

So this is my image . So if I just write down for this image, height is equal to 1 2 3 4 5, width is 5. So these are all 5. And then I decided to put one layer of 0s, so the padding P is 1 like we already have here. So that increases the idea on which the mask can float around. Once I put padding on one layer, then I am essentially saying that the mask need not start overlapping the image from this point.

But rather it can start from this point and it can come up to this point, this point and go up to this point. So that increases the number of shifts that the mask can make. Why is that important, because that decides the output images height and width. So, as you can see, if I consider P equal to 0, then this mask of size 3 cross 3 can make how many stripes, it can make how many movements on the x axis, it can have one location, then the next location, then the third location.

So that makes the output images width to be 3. But now with the zero padding, what is the maximum number of locations on which the mask can traverse in the x direction. So as you can see, I can really start here, then I can move the mask here. And then here like that. So 1 2 3 4 and 5, because one of the best comes here and is done . So 5 possibilities, that is that makes W prime equal to 4.

And since H is also equal to W, so the mask and also take how many locations in this direction 1 2 3 4 5 , is the last possibility. So H prime equal to W prime equal to 5. Now, let us now go and walk back on the formula. So these are formulas. $1 + H - F + \text{twice of } P$ divided by S. So let us see how it comes. Now, in this case since I am moving the mask by one position at a time, so the stride S equal to 1 .

So if you just check it out, so effectively I have A as 1, the height is 5. Now of course, the mass has a width F, which is equal to 3. That means after this point it really cannot move. So the entire height - 3 +, you get 2 extra positions to move because of the padding ., Since padding is 1, one position this way, one position this way, so 2 into 1, and it should work out to 5 - 3 + 2.

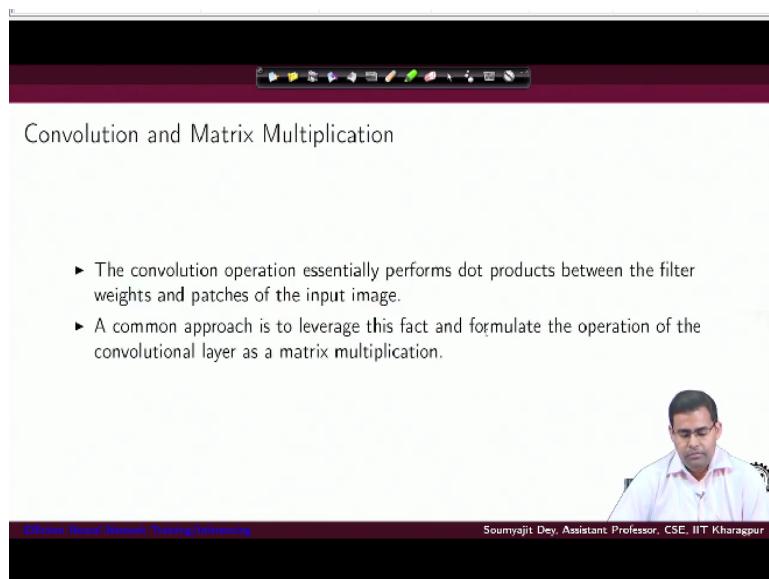
So that is $5 - 1$ by 1. So that is $4 + 1$, that is 5. It is coming off properly. And similarly, the calculation can be done for W prime. So for W prime, I will really have $1 + W - F + 2 P$ by S. But what about modifying the S. If you modify the S and make it 2 the naturally the strides get bigger, so, you get one mask position here, then the next mask position will be here. Because this is then $S = 2$.

And then the last mask position is here. Because, again you make another stride , so, 3 positions, so, I expect if I put $S = 2$, it should really give me the result as 3 and that is going to happen if you put $S = 2$. So, this is instead of 1, you have $1 + 4$ by 2, that is 2, so you will get a 3 here. So, that is how it works. **(Video Starts: 10:06)** So, with $P = 1$, this is how you make progress and compute the convolved output you are moving in strides of 1.

And you get all the values computed. If we just show a small simulation of $S = 2$, just like we discussed, you will move in half of 2 and similarly, just like we discussed, the output will be 3X3.

(Video Ends: 10:31)

(Refer Slide Time: 10:32)



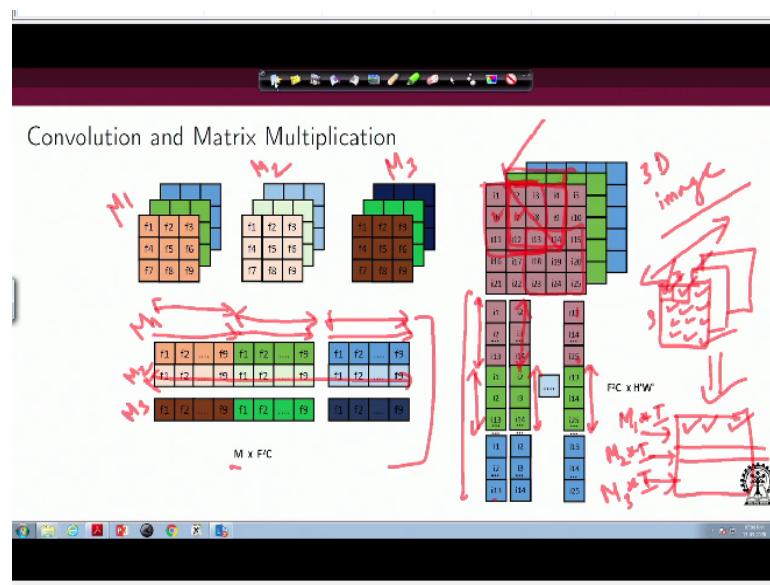
So, just to relate what is the difference now, instead of doing the matrix multiplication, you are now doing convolution and the operation essentially performs dot products between the filter weights and the patches of the input image on which it is working locally. By patch the region of

the input image on which we are overlaying the mask and doing the operation. Now, a common approach like if you want to do the convolution operation of the CNN efficiently is to leverage this fact.

And formulate the operation of the convolution layer as a matrix multiplication. So let us try and understand what is the advantage. So, just like we discussed, convolution is just individual dot products, all the filter weights and the input image, we already have efficient routines for performing matrix multiplication and different optimizations.

So, if we can just adjust the mask's weight values and the images, pixel values in such a way that the operation reduces to the matrix multiplication operation that can really help us to get the job done using available or optimized matrix multiplication routines. So there is a trick we are going to play for implementing convolution efficiently.

(Refer Slide Time: 12:03)



So let us see how it works. So, this is a very simple and nice example. So, consider on the left hand side, you have these masks you have 3D masks, and you what you really want to do is you have mask M 1 mask M 2 and you have mask M 3 at some region of the computation. What you really want to do is you have this 3D image and you want to transform this 3D image and get the transformed outputs for all these masks M 1 M 2 and M 3.

And so is just like you have the input image and you want these transformed outputs to flow to the next layer. And they can really be done in parallel because they are quite independent operations. So, as an optimization, what we can do is we can arrange the masks and the image data values in such a way that finally, the operation of computing the transformed outputs will boil down to the matrix multiplication operation.

Because as we have seen that individually the mask weights will be component wise multiplied will get component wise multiplied with the image pixel values. So, what we really do is very simple. Let us take mask 1, for mask 1, you arrange all the values in a single row, first layer, and then again from the next layer. Then again from the next layer , this is all for M 1. Do the same thing for M 2, do the same thing for M 3 like that.

And prepare one large matrix. What would be the size of course the number of masks times the size of each row. The size of each row is of course a square . So this is a square, $F = 3$ here F cross F , , I am speaking of capital F , which is the width or height of the filters. So, a square times the number of channels, 2 channels F square cross 3, that is the row size multiplied by the number of masks.

So that is one large matrix that you are preparing by using all these masks. Why do you do this? Well, that becomes clear when you look at what transformation we make for the image. So this is the image on the hand side, 3 channels for each channel and sorry, here we are talking about different masks. Each of them are 3D masks, and each mask finally gets to 1 row. And each of them and the second row is for mask 2, third row is for mask 3 like that.

Here I have the image. And for the image I have 3 channels for each channel I have a 5 cross 5 matrix . Now, if you think about what is really going to happen, when you do a normal computation, let us say I am trying to convolve the image with mask M 1, what am I really supposed to do? So I will take the first layer, let us say this brown layer. And then I am supposed to overly use this image at this position.

Then the next position assuming a stride of 1, and so on and so forth . So that gives me how much for this position, I am going to do a transformation. For the next position again I am going to do another transformation. And for each of the computations, I am going to calculate and store the value. So how do I do it in the case where I have modified this mask into that matrix that I have shown here.

How do I really do it. Because it is expecting the input data now in a different format. So what I am really going to do is I am going to duplicate this data into multiple positions. What I do is let us say this initial position of the mask, initial position would have been here. And I am supposed to do F_1 multiplied by i_1 , F_2 multiplied by i_2 , F_3 multiplied by i_3 , F_4 multiplied by i_6 , F_5 multiplied by i_7 like that.

Since I have F_1 to F_9 in a row, I just put this position of the data which is here. In the first overlay position of the mask, I just arrange it in the form for columns. So now the column contains $i_1, i_2, i_3, i_6, i_7, i_8, i_{11}, i_{12}$ and i_{13} , as you can see if I multiply this row, this row segment, let us say, if I just multiply this row segment with this column up to this, I get the value I was looking for, which I am supposed to put in an output.

So if I just write what I am supposed to do after all this , so for M_1 , I am going to do these multiplications and I am supposed to get 3×3 matrix 3×3 and a stack of them considering the other components this, this, this, they are going to give me a stack of these kind of 3×3 matrices. So now, what I am doing is I am supposed to compute each of these locations and they come out when I multiply this with this.

This row segment gets multiplied with this column segment to give this value. Again, this row segment, when it gets multiplied with the next column segment is going to give me the next value . So what is this column segment? Now I am going to shift the mask here. So now I am going to take values $i_2, i_3, i_4, i_7, i_8, i_9$. So, i_2 up to i_{14} . So, essentially this much, as you can see from i_2 to i_{14} .

And at the last I am expecting i_3 to i_15 . So i_2 to i_15 , sorry, and last means this location. So in this location, I am expecting these values. So that is i_3 to i_25 , as you can see i_3 to i_25 . So all I am saying is I want these individual values to be computed. But what I do is I arrange them like these columns, I arrange the input mask like this row segment.

And I multiply this row segment with all these columns to get the first layer of the 2D output done . Now, you can understand the same thing is going to work for the next set of channels , because now, if I just continue with the operation, then whatever mask values are there, in the next layer of M 1, they get multiplied with similar things here. And how does that help because I am not only going to compute this layer wise weighted values weighted sums.

But the next thing is I am supposed to add them across layers. Since I am supposed to add them across layers. Here in this case, it is getting reduced to an entire matrices row multiplied by column computation, because I have already computed M 1 up to this position, then you compute all the locations from F 1 to F 9 for the intermediate layer in the mask and you are already having the data for this second layer arranged here .

So, all you get is while you are computing these locations, you are also simultaneously computing the backward location for the same mask. That means, you are computing this as well as this value, but these are not the individual values that you are finally interested in what you are finally interested in, you are essentially interested in this entire column, entire row, pairwise multiplications and then the final addition.

That is going to give you the final 2D output matrix considering 1 mask M 1 , I hope this is clear. So, once you do this you get for M 1 what is going to be the value here, the next value, the next value and so on so forth . So, for M 1 I am finally expecting what 9 values and then they get multiplied here and finally, what do I really have essentially, and when you add them up you are going to have 1 value here, next value like that so on so forth.

But, overall you have successfully computed the entire thing for the 2D image. And then what are the total number of positions that you have covered. If you look at it minutely, it is exactly the

same as the size of the output. Considering 1 mask, but finally, you are going to have so many of them . So, the same thing will now repeat for mask 2, because you now have the entire mask 2 capture in a single row.

So, you just keep on applying this mask 2 on all the columns individually. So, in that way, you will have all the values that you are going to compute for mask 2 available for these columns, get them working with the rows and you get the final results for mask 2 computed in the next row. First row is computing all the values that you would have required for mask 1. So, this is essentially mask 1 transformation with image i.

Next row is mask 2 transformation with image i. Next row is mask 3 transformation with image i, so on so forth. I hope this is clear. Now, how are the entries holding up here. For mask 1, how many entries are going to be there. Well, it is really the number of overlay positions I can have for mask 1, which is essentially 3 cross 3, so I essentially get 9 entries. And that is what is going to happen because I have actually got 9, all the 9 possibilities for mask already captured here.

In terms of each of the columns, so I am really having 9 columns here. So here in 1 row with 9 entries, I am going to get the entire final output for mask 1, then for mask 2 in the next row, then for mask 3 in the next row, and that is how it continues . So as you can see, the whole point of doing this transformation and the multiplication is I am successfully able to convert this convolution of many weight matrices into matrix multiplication problem.

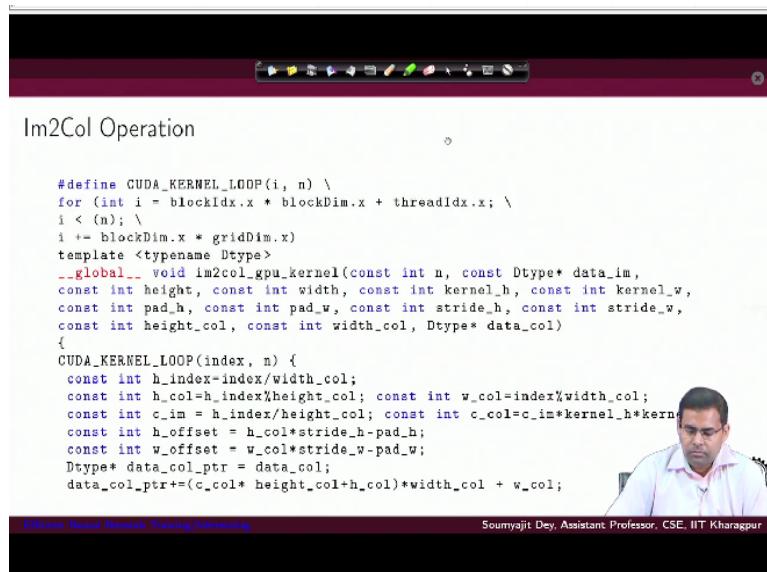
Where all these convolutions outputs are computed in parallel . What is the overhead? Of course, you have some data overhead, because essentially you are creating a data layout where entries are getting duplicated. Why is that happening? Well, here you do not have any duplication. You are just considering each of the masks and taking all of that together into a large matrix wherever you arrange it in a smart way.

So that the number of rows is the same as the number of masks, the number of columns is the same as mask size times the number of layers, which is obvious here. But so there is no duplication. The duplication is the way that you are arranging the image. Because here you come for each let us this

is the , one layer of the image, for that you have got 25 possible values and you are duplicating them.

Because you are trying to keep each of the possible overlay instances of the mask separate, since you are going to have 3×3 overlay instances, 9 overlay instances, so you are having to i 2 hear i 2 here you are having i 3 here i 3 will also be here. So, there will be many duplications like this. But as we are considering parallel programming language, essentially we will have multiple threads, who will be creating this data structure in parallel. And then engaging into the matrix multiplication task.

(Refer Slide Time: 25:41)

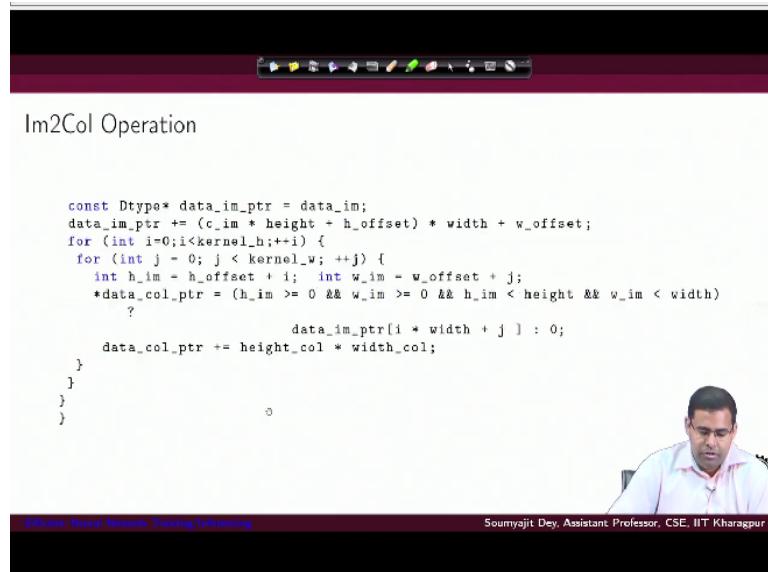


The screenshot shows a video player interface with a dark theme. The title bar says "Im2Col Operation". The main area displays a CUDA kernel code for "im2col_gpu_kernel". The code defines a template function with parameters: const int n, const Dtype* data_im, const int height, const int width, const int kernel_h, const int kernel_w, const int pad_h, const int pad_w, const int stride_h, const int stride_w, const int height_col, const int width_col, Dtype* data_col. It includes a CUDA_KERNEL_LOOP loop for i from 0 to n-1. Inside the loop, it calculates indices for columns and rows, and updates a column pointer. A watermark of a man speaking is visible in the bottom right corner of the video frame. At the bottom, there is a navigation bar with icons and text: "Official Neural Network Training/Inference" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

```
#define CUDA_KERNEL_LOOP(i, n) \
for (int i = blockIdx.x * blockDim.x + threadIdx.x; \
i < (n); \
i += blockDim.x * gridDim.x)
template <typename Dtype>
__global__ void im2col_gpu_kernel(const int n, const Dtype* data_im,
const int height, const int width, const int kernel_h, const int kernel_w,
const int pad_h, const int pad_w, const int stride_h, const int stride_w,
const int height_col, const int width_col, Dtype* data_col)
{
CUDA_KERNEL_LOOP(index, n) {
const int h_index=index/width_col;
const int h_col=h_index%height_col; const int w_col=index%width_col;
const int c_im = h_index/height_col; const int c_col=c_im*kernel_h+kernel_w;
const int h_offset = h_col*stride_h-pad_h;
const int w_offset = w_col*stride_w-pad_w;
Dtype* data_col_ptr = data_col;
data_col_ptr+=(c_col* height_col+h_col)*width_col + w_col;
}
```

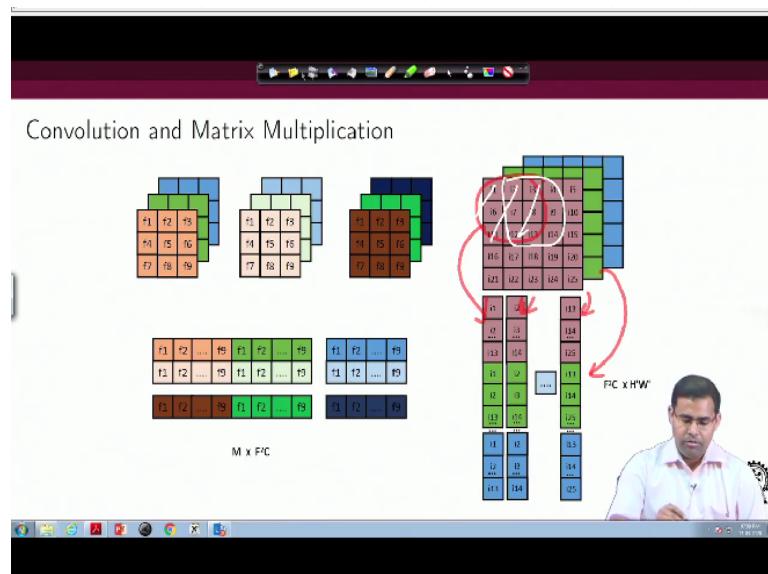
So, the creation of this image to column thus what is popularly called image 2 col **col** column operation is available in most of these free software's like the deep learning libraries like cafe. And this is one of those examples which you have just given, you can have a look at it with the slides that we provide, I think it is not required that we keep on talking about what is happening here in each of the lines.

(Refer Slide Time: 26:09)



So, just to repeat.

(Refer Slide Time: 26:13)



All we are doing is we are launching sufficient numbers of threads, so that each thread is responsible for copying one possible overlay of the image, , the mask position. So, for this, I will have one thread do the copy here. Another thread will do the copy here. Another thread will do the copy here, another thread for this, so on so forth. So, in that way the threads will be doing and doing the copy operations in parallel.

And well, will there be some coalescing happening here. I believe so, because if you see, let us say one thread is doing copy of all these data, the next thread is doing copy of these data points. So

naturally, , when this guy is copying this one, the next one is going to copy this, the next one will be copying this. So, you have scope of memory coalescing, and successful use of shared memory and all that, which you can really explore using the optimizations we have discussed earlier .

But if you just look at the vanilla code that we have provided in the next slide, we are not really making use of shared memory or other things, but we are engaging multiple threads to do the copies parallel from the global memory, and you can check as a task. How much is the scope for memory optimization here. How much is the coalescent here and all that .

(Refer Slide Time: 27:43)

Convolution and Matrix Multiplication

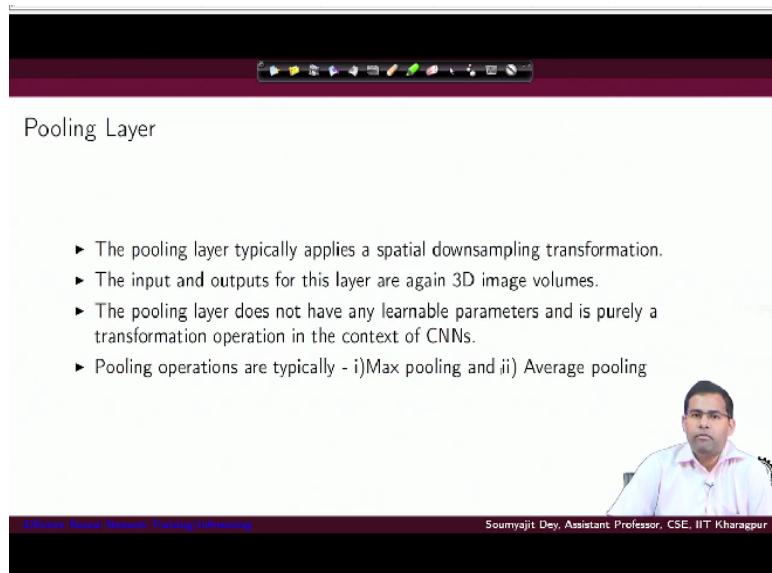
- ▶ The disadvantage of the im2col approach is extra memory, since some values in the 3D input are replicated multiple times in the columns.¹
- ▶ However, the benefit is that there are many very efficient implementations of General Matrix Multiplication that we can take advantage of (cuBLAS API).
- ▶ We'll discuss certain GEMM optimizations later.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, that is kind of a disadvantage, we can say that when I am doing the image 2 column approach, I require extra memory, since some of the values are replicated multiple times in the columns. But the good thing as we have discussed is that there are really many, many efficient implementations of the matrix multiplication or the way people popularly call it is general matrix multiplication or GEMM.

And people can take advantage of the GEMM operations. Once this image 2 column transformation is ridden. For example, there is this very popular CuBLAS library, which contains efficient implementations of GEMM.

(Refer Slide Time: 28:24)



Some of those GEMM optimizations are also things that we will be discussing later on. (**Video Starts: 28:29**) Well, that would be all about our convolution layer. The next thing that comes into space is the pooling layer. Because if you remember our convolution architecture, so (()) (28:38) we will just go back once. So, here, we talked about this multiple layers. So the input layer is there, then convolution then pooling and then the RELU layers.

So essentially, the pooling is nothing but you are doing a down sampling of the image. (**Video Ends: 29:01**) So summarize is typically a special down sampling transformation, the inputs and outputs both will be 3D image volumes and but there are not really any learnable parameters, unlike the CNNs and the convolution layers, but what do you do is you are trying to reduce the input image size.

Why are you interested in pooling? Because remember that while in the front, the input layer you may have these heavy images with multiple channels and all that but at the end if you are doing a classification kind of problem, you are only interested in the last this class course . So, you want a linear array with different scores to field it up . So, you there has to be intermediate layers interspersing the convolution layers where you will try to reduce the output image sizes .

Now, there are several possible pooling operations typically one is max pooling and the another is average pooling. (**Video Starts: 30:01**) So what does that really mean. So here by let us say these

are bigger dimension images, and I am trying to reduce it to a smaller dimension. So I can do a max pooling, that means I choose a mask of this size. And for all these values, I just put the maximum here, I just move it over a stride of the same size.

And again, I put the maximum value here. This is just a transformation, which is kind of down sampling because I am moving from a more information representation to a smaller information. And compact representation and losing information here. So that is the down sampling . So I could have done max operation or I could have also done an average operation. So both are possible in pooling layers. **(Video Ends: 30:51)**

(Refer Slide Time: 30:52)

Pooling Layer: Spatial Transformations

- ▶ The Pooling layer takes as input a 3D image volume of dimensions $C \times H \times W$.
- ▶ The hyper-parameters for the layer are as follows.
 - ▶ F - Filter Size
 - ▶ S - Stride
- ▶ The layer produces a 3D volume of dimensions $C' \times H' \times W'$ where
 - ▶ $C' = C$
 - ▶ $H' = 1 + (H - F)/S$
 - ▶ $W' = 1 + (W - F)/S$

Office: Research, Networks, Training/Inference
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So just to formulize I am considering 3D images here for pooling, number of channels, times height, width and then the hyper parameters of the layer are well again have to decide that what is the filter size that I am going to talk about, for example, in this picture, there is a 3 cross 3 filter, and it is moving in strides of 3, if I increase the stride size, I am going to have more amount of down sampling as we can see.

So once I choose the filter size and the stride size, what the pooling, , out of the pooling, I get a modified transformed output of dimension C' H' W' , where naturally the C' remains C same because nothing changes here, the number of channels is same, but of course, the

height and the width values get modified, following similar formulas that we like we have elaborated earlier. So it is simply 1 plus H minus the filter size divided by the stride.

(Refer Slide Time: 31:52)

The slide shows a CUDA kernel implementation for MaxPoolForward. The code defines a function that takes parameters like nthreads, bottom_data, num, channels, height, width, pooled_height, pooled_width, kernel_h, kernel_w, stride_h, stride_w, pad_h, pad_w, top_data, mask, and top_mask. It uses a CUDA_KERNEL_LOOP macro to iterate over index. Inside the loop, it calculates pw (index % pooled_width), ph (index / pooled_width % pooled_height), and nc (index / pooled_width / pooled_height % channels). It then finds the maximum value in the slice and updates the top_data and mask arrays accordingly. A watermark at the bottom right indicates the source is Efficient Neural Network Training/Inference.

```
template <typename Dtype>
__global__ void MaxPoolForward(const int nthreads,
const Dtype* const bottom_data, const int num, const int channels,
const int height, const int width, const int pooled_height,
const int pooled_width, const int kernel_h, const int kernel_w,
const int stride_h, const int stride_w, const int pad_h, const int pad_w,
Dtype* const top_data, int* mask, Dtype* top_mask) {
    CUDA_KERNEL_LOOP(index, nthreads) {
        const int pw = index % pooled_width;
        const int ph = (index / pooled_width) % pooled_height;
        const int nc = (index / pooled_width / pooled_height) % channels;
        const int n = index / pooled_width / pooled_height / channels;
        int hstart = ph * stride_h - pad_h; int wstart = pw * stride_w - pad_w;
        const int head = min(hstart + kernel_h, height);
        const int wend = min(wstart + kernel_w, width);
        hstart = max(hstart, 0); wstart = max(wstart, 0);
        Dtype maxval = -FLT_MAX; int maxidx = -1; //FLT_MAX --> max float
        const Dtype* const bottom_slice=bottom_data+(n*channels+nc)*height;
        for (int h = hstart; h < head; ++h) {
            for (int w = wstart; w < wend; ++w) {
                if (bottom_slice[h * width + w] > maxval) {
                    maxidx = h * width + w;
                    maxval = bottom_slice[maxidx];
                }
            }
        }
        top_data[index] = maxval;
        if (mask) {
            mask[index] = maxidx;
        } else {
            top_mask[index] = maxidx;
        }
    }
}
```

Efficient Neural Network Training/Inference
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So that gives you a smaller and simpler a transformed image to work with , you can just do a simple implementation of pooling kernel, so this is also taken from available repositories. Again, we are not really getting into discussing this operation is quite simple.

(Refer Slide Time: 32:08)

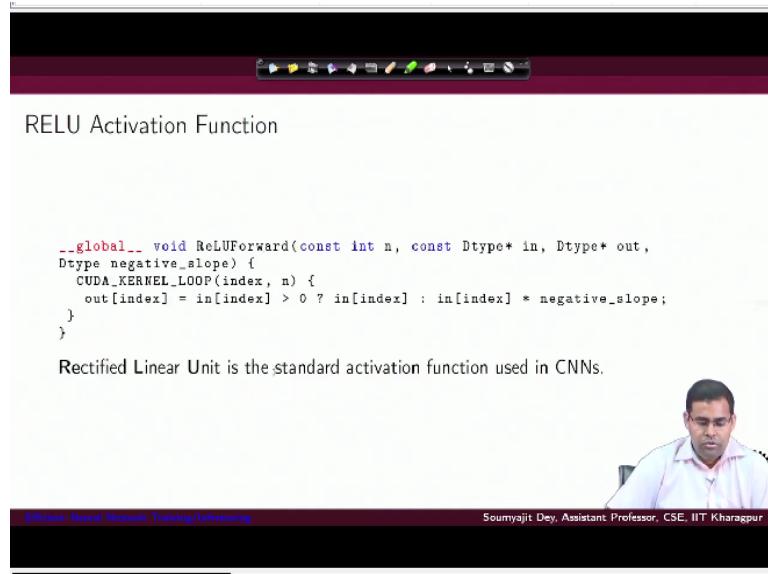
The slide shows a C implementation of a pooling kernel. It uses nested loops to iterate over height and width, comparing values in the input slice to find the maximum. The maximum value is then assigned to the output top_data array, and the index of the maximum value is stored in the mask array. A watermark at the bottom right indicates the source is Efficient Neural Network Training/Inference.

```
for (int h = hstart; h < head; ++h) {
    for (int w = wstart; w < wend; ++w) {
        if (bottom_slice[h * width + w] > maxval) {
            maxidx = h * width + w;
            maxval = bottom_slice[maxidx];
        }
    }
}
top_data[index] = maxval;
if (mask) {
    mask[index] = maxidx;
} else {
    top_mask[index] = maxidx;
}
```

Efficient Neural Network Training/Inference
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

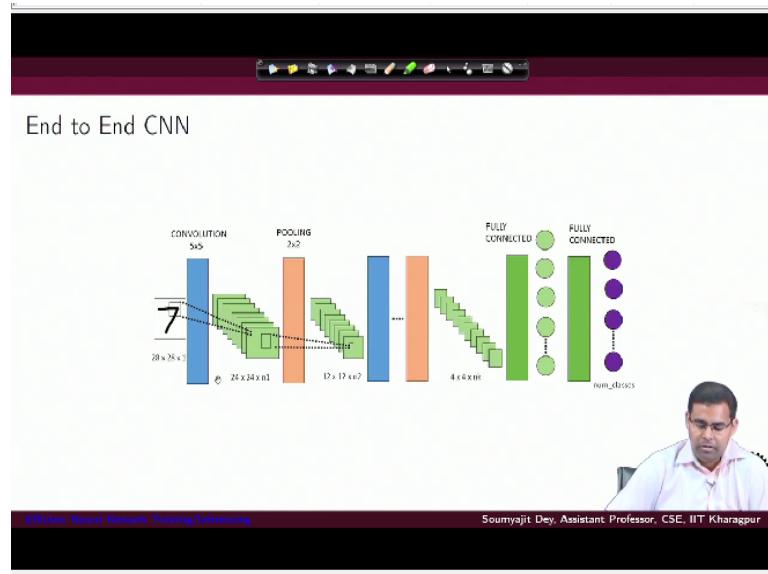
And the idea is that by doing the pooling, you are conveniently decreasing the transformed image sizes. So, that at the end of the pipeline, you really get the class course.

(Refer Slide Time: 32:24)



So, the other layer that you have is the activation layer, which is very similar to the ones that we have been talking about earlier. And for the activation function in CNN, the standard that we use is the rectified linear unit, which we have discussed earlier during our discussion on current neural networks.

(Refer Slide Time: 32:43)



So, if I look at the structure of an end to end CNN, you have an image here, and then you will have convolution layers, followed by pooling layers, again convolution layers, followed by pooling layers like that. And, of course, after every convolution, you are also , you are going to have RELU layers also. So again, if I just go back into that slide where we define the CNS. So that is what you have input, then convolution, pooling and then the RELU .\

And this will repeat many times you have convolution again, the pooling was down sampling and then RELU and all that. And the RELU is, like we have discussed earlier, it is giving you the required non linearity. And so in that way, you keep on transforming the image and also downsampling your size. And at the end, you have a fully connected layer, just like our normal neural network architecture.

And the fully connected layer will be reducing this, the transformed image to the final class course for the output layer where you have the different number of nodes as the number of classes that were talking .

(Refer Slide Time: 34:01)



Recap: Backward Propagation

- ▶ After the forward pass is completed, the first error term is calculated as $Y - O$ where Y is a column vector of true labels, O is a column vector of predicted labels.
- ▶ During the backward pass, for the layer with weight matrix W' , two items are computed:
 - ▶ The error term δ^1 is calculated as an elementwise product: $\delta^1 = (Y - O).f'(Z)$ where $f'(Z)$ is a column vector where each value represents the gradient of the activation function in the given layer.
 - ▶ The gradient of the layer i.e. $\frac{\partial J}{\partial W'}$ which is $A^T \delta^1$ where A was the input matrix for that layer.



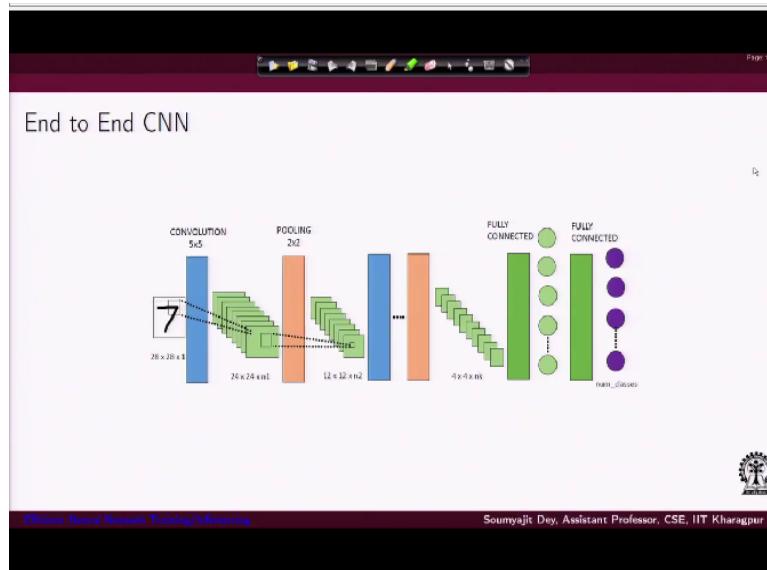
So that is how the CNN pipelines are arranged. The next thing that we will be looking into is how the data really flows across this pipeline, how the backpropagation and of course, the feed forward propagation works specific to CNN pipelines, what are the computations involved and how are they carried out okay. So with this will end the current lecture, thank you for your attention.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-59
Efficient Neural Network Training/Inferencing

Hi. Welcome back to the lecture series on GPU architectures and programming.

(Refer Slide Time: 00:27)



So, if we remember in the last lecture, we have been discussing how CNN pipelines work and what are the underlying computations that need to be done for such systems.

(Refer Slide Time: 00:41)

Recap: Backward Propagation

- ▶ After the forward pass is completed, the first error term is calculated as $Y - O$ where Y is a column vector of true labels, O is a column vector of predicted labels.
- ▶ During the backward pass, for the layer with weight matrix W' , two items are computed:
 - ▶ The error term δ^1 is calculated as an elementwise product: $\delta^1 = (Y - O).f'(Z)$ where $f'(Z)$ is a column vector where each value represents the gradient of the activation function in the given layer.
 - ▶ The gradient of the layer i.e. $\frac{\partial J}{\partial W'}$ which is $A^T \delta^1$ where A was the input matrix for that layer.

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Prof.

And based on that we will just try to now figure out what is the nature of the task graph model for CNNs, just like we have discussed the task graph models, more specifically the computational graphs that can be created for generic neural networks. So before going to that, we will just do a quick recap of our discussions on back propagation. So, what really happens is that if you remember the notations that we have discussed earlier, while completing the forward pass over a neural network, we were denoting the error term as $Y - O$, where Y is the column vector of true labels and O is the column vector of the predicted labels as we discussed earlier. And then during the backward pass, which started with a layer with the weight matrix W' , we are computing the following quantities.

For example, we first computed δ^1 , which was that product term that we figured out. Essentially δ^1 was like this $Y - O$ multiplied by the derivative of the activation function, which again is a column vector and each of the values represented the derivative of the function in that given layer. And once the δ^1 was computed, that actually helped us to figure out what is $\partial J / \partial W'$ and we have already derived that. For this the expression is $A^T \delta^1$ where A is the input matrix for that layer. It has been found by taking the input and doing a product with W and all that like we discussed earlier.

(Refer Slide Time: 02:34)

Recap: Backward Propagation

- During the backward pass, for the layer with weight matrix W , again two items are computed:
 - The error term δ^2 is calculated as $\delta^2 = \delta^1 W^T f'(Z)$
 - The gradient of the layer i.e. $\frac{\partial J}{\partial W}$ term which is $X^T \delta^2$ where X was the input matrix for that layer.

Soumyajit Dey, Assistant Prof.

So, if you remember, we now have the availability of δ_1 with us. Now, using δ_1 we can also calculate δ_2 like we discussed earlier. So, δ_2 is nothing but $\delta_1 W^T$ multiplied again by the derivative of the activations and using δ_2 , we can compute the derivative of J with respect to W . And this is given by $X^T \delta_2$ where X is the input matrix for that layer. So, these are the different terms that need to be derived while doing backpropagation.

(Refer Slide Time: 03:19)

Recap: Backward Propagation

- In a similar fashion, if there was one more layer with weight matrix W^o , then again the following computations would be done:
 - The error term δ^3 is calculated as $\delta^3 = \delta^2 W^o f'(Z^o)$ where $f(Z^o)$ represents the gradient of the activation function in the given layer.
 - The gradient of the layer i.e. $\frac{\partial J}{\partial W^o}$ term which is $X^o T \delta^3$ where X^o was the input matrix for that layer.

Let us consider a general scenario with l layers with each layer i having weight matrix W_i .

Soumyajit Dey, Assistant Prof.

Again, we would just recall that one of the derivatives of the loss function is $A^T \delta_1$, where δ_1 is the error multiplied by $f'(Z)$ which is the gradient of the activation and the other that is $\partial J / \partial W$ is computable as $X^T \delta_2$, where X is the input matrix and δ_2 is given by using δ_1 and the other expressions here.

So if we are just using these values in a similar fashion and if there are more layers, then what would really happen? So, let us recall the previous things. In our first neural network example, we just considered that there is a layer with the weights W and then there was this layer with weights W' and we derive δ_1 here. So, if we are just doing a pictorial description, δ_1 was this value and this was f' and then we will just multiply this with the A^T . And that essentially gave me this derivative that $\partial J / \partial W'$. So, that is essentially the computation we require for the last layer for W' and then we had this previous layer corresponding to W . For that we next computed this δ_2 right and for δ_2 we actually use δ_1 and then we got $\partial J / \partial W$.

And then we multiply it by the previous layer's transposed weights and of course, the f' prime and others. So, essentially as we can see that there is a dependency of the previous layer's loss derivative on the last layer. Now, if we just take this solution into a cascade then what is going to happen? So, suppose there are more such layers let us for example, add one layer with weights W_0 . Then we will have this if we just continue like the way we first had calculated delta 1.

We use delta 1 to calculate delta 2, right. We just continue like this. So one small problem we have, I believe this should be ∂W and is not going to be W' . (**Video Starts: 06:18**)

So just a minute. So when we are doing this, this is W prime. The loss derivative $\partial J / \partial W$ is using W prime which is the last layer and we are going to continue the same. (**Video Ends: 06:40**)

(Refer Slide Time: 06:41)

Recap: Backward Propagation

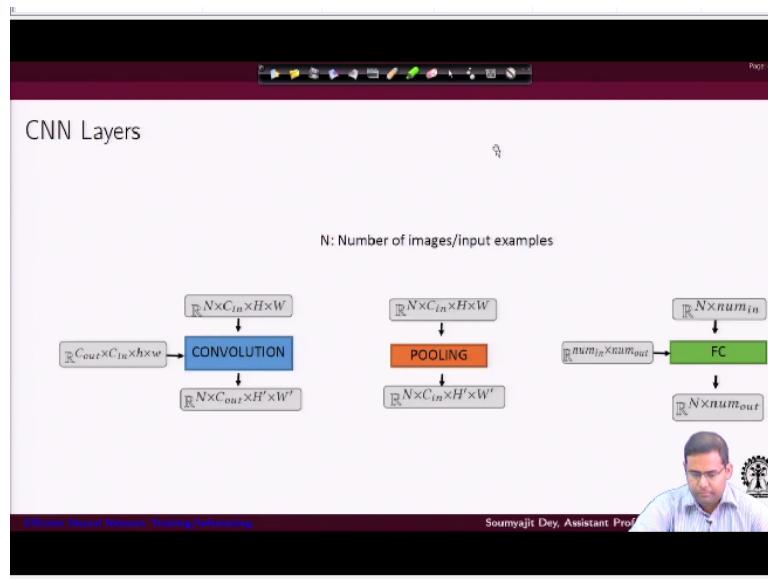
- ▶ During the backward pass, for the layer with weight matrix W , again two items are computed:
- ▶ The error term δ^2 is calculated as $\delta^2 = \delta^1 W^T f'(Z)$
- ▶ The gradient of the layer i.e. $\frac{\partial J}{\partial W}$ term which is $X^T \delta^2$ where X was the input matrix for that layer.

Then just like I am computing δ_2 , and I am using δ_2 as δ_1 multiplied by W^T . So essentially, W^T is the weight matrix for the layer in which I have computed δ_1 . So if we just try and segregate, here I have W . Here I have W' in the next layer. In this layer, I have computed δ_1 . In this layer, I am computing δ_2 , and I am using δ_1 and then W' , which is basically the weight of the next layer. And then I have this f' as well. So what we are really trying to say is that this is going to continue.

So now, if we bring in a new layer here, let us call it W_0 . So then we will have δ_3 here. And for calculating δ_3 , we will just see that the same thing is going to continue. So we will now have δ_2 and we will have W here and then we will have f' again for Z_0 , that represents the gradient of the activation function in the given layer. So when I am computing δ_3 is $\delta_2 W_0$ and then $f'(Z_0)$, which represents a gradient of the activation function of this layer.

Whereas for the previous it was just Z , where it was the gradient of the activation function of that layer. Now, for the gradient of the layer $\partial J / \partial W_0$, I can again figure out using $X_0^T \delta_3$, considering that X_0 is the input matrix for this layer.

(Refer Slide Time: 08:49)

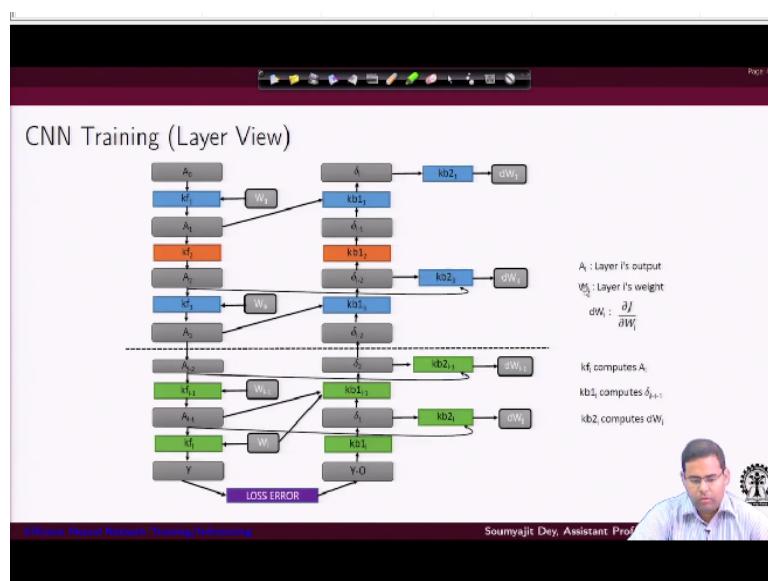


So the point is very simple for every layer. I can keep on computing these delta values. And then I am going to do a matrix multiply with the weight matrix, and then also a further matrix multiply with the activation function's output. So that is how things are going to progress. Now,

let us consider a very general scenario, where I have these L layers, and each layer i has a weight matrix W_i . So in general, as we have already discussed in CNNs, I have this different kinds of layers. For example, I have the convolution layer. So in convolution layer, I have these 2 inputs. So I have the image input and I have the mask to be convolved with the image that gave the convolved output. In pooling I have a transformed image input, and I am going to down sample it.

So the H and Ws are going to change to H's and W's whereas the other things remain same. And then I have the fully connected layer. So, in the fully connected layer, I am having the weights and I am having the input transformed image. And finally, it gets reduced to a linear array full of different class members.

(Refer Slide Time: 10:24)



I pull it down, and the down sample, I get A2. Again, there is a convolution layer. Now, of course, for every convolution layer, I not only need the image, but also the mask.

So I have a W1 here as an input. Let us say I have a W3 here as an input, and so on, so forth, right. So I have a convolution followed by pooling again, convolution followed by pooling like that. So here, I can think that these dots represent that there are more such different layers here. At the end, I am going to have the fully connected layers. So, let us say the green ones, represent the fully connected layers operations.,

So at the end of the fully connected layer, I will get the overall output of the network. From here, I can compute the loss. And then I can use this loss error, that is I can get Y and I can compute the error that is $Y - O$, and then I am going to use $Y - O$ to compute each of the previous layers δ values, because we have already discussed here, how I can in general compute the δ values of some layer based on the δ value of the previous layer and also use that layer's weight matrix, and the activation's gradient and all that, right. So with this here we are getting the error, and I am using this matrix. The kernel kb1 is actually representing the computation of δ_1 making use of , $Y - O$ and f' . So there is no requirement for any input of the weight matrices.

So I get δ_1 here. But then when I use δ_1 , to get δ_2 , if you remember the expression of δ_2 , which is something with respect to the previous one, in general, you need a weight matrix transposed and then this A will be transposed here. So there are dependencies from A to here. So this kernel is going to make use of A, is going to make use of the W of that layer in order to compute δ_2 .

Now let us again review why this is necessary. So as you can see, this δ_2 is nothing but δ_1 , the weights and the value of f' right and then we are multiplying it with the input matrix in the backward propagation. But what happens when you are just computing the first δ ? The first δ is $(Y - O) f'$, and then you are multiplying this with A^T , which is the input matrix to get the original loss function.

So coming here, when I have to go from this δ_1 to δ_2 , what is really happening? So I am using this δ_1 , and I am using this W1, i.e. this weight matrix. And I am going to use this A to get δ_2 .

Then I am again using δ_2 to get the previous one δ_3 , and like that, this will make progress. And then I come back to different pooling layers, where again, I will be computing this in the backward pass and I keep on computing $kb1_i kb2_i kb1_i kb2_i$ like that to get all the δ values here.

So, we are trying to show our task graph view of these different kernels that are going to perform these data transformations of the convolution layer, the pooling layer and the fully connected right. Now also we need to see that computing the δ_2 s and the δ_3 s in general that is not all, because what is my overall requirement? My overall requirement is to compute in general the loss with respect to the different weights.

So, let us represent this $\partial J / \partial W_i$ by this text dW_i . So, in general, I can say that this kf_i are the kernels which compute the output given matrices A_i 's. Then let us say $kb1_i$ represents kernels, which are actually responsible for computing the deltas in the backward pass. And similarly, let us say $kb2_i$ is a different class of kernel, which uses $kb1_i$'s outputs that is the δ_{l-i} 's.

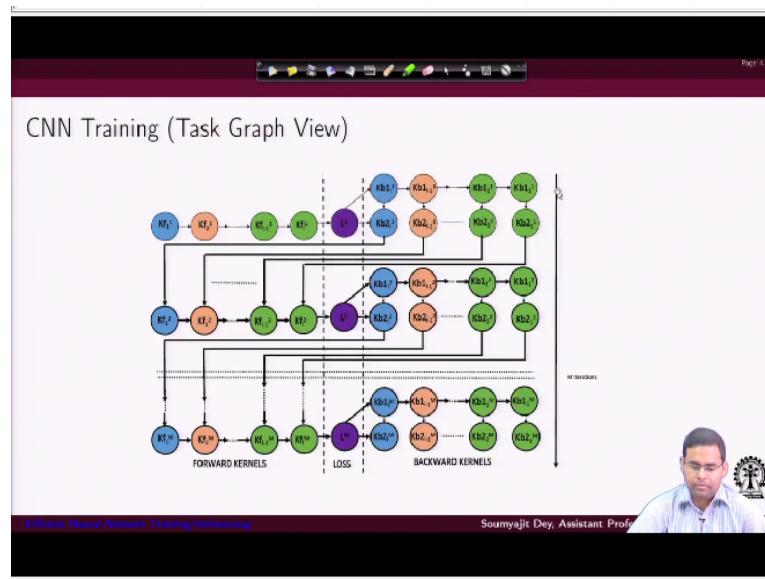
And also these compute the overall loss because if you remember, the final loss is basically this image matrix transposed multiplied by the deltas. So that is a final loss. So that is why I need these additional kernels with dependencies which are coming from the A 's. So I will just review what we exactly did. So if you see, the computation of every delta depends on the previously computed deltas value. Let us say I am taking δ_2 and δ_1 and the previous layer's weight matrix. As you can see here, delta 2 is going to use this weight matrix W and the activation function's gradient. So both of those. That is why I have this dependency here.,

And we are just trying to show that this activation function is available and the gradients' outputs are also available. Now, when are the gradients' outputs available? Well they get available immediately after you compute the A 's because finally, from the A 's you are passing them through the activation functions only to get the actual matrices. So here, you are using these W 's and A 's.

And the previous δ s for this $kb1$ kernels to get the next layer δ . So they form one class, then you use these $kb2$ class of kernels that get as input the δ s as well as the A matrices for different layers to give you the losses with respect to the weights in that layer. And that is how it will progress.

So, overall, if I see the task graph view of the CNNs forward pass, followed by the backward pass, I start with these values of W1 W2 W3 like this for all the layers and at the end of the backward pass I am successful in computing the derivatives of all these layers' weights. And then of course, we will do the obvious thing, which is that we will just update this W1 with $W1 + dW1$. Similarly, like that for the other W's and then we will do again the forward backward traversal like that right. This is just like a dependency oriented view of the CNN computation.

(Refer Slide Time: 19:09)



So, if we now try to have a look at how these different kernels or these different data parallel kernels are going to work, then what are their dependencies? So, as you can see, you have the convolution layer kernel which is doing the operation of computing the image transform followed by the activation function. Then you have the pooling layer kernels here, so convolution then pooling like that.

So, that sequence will continue and then you will have the kernels for the fully connected layer, these entirely form a forward pass, then you have functions which compute the loss values. So these are kernels for doing the loss value computation. And then you have these 2 set of kernels, one set of kernels which are just going to compute the delta values for each of these different layers.

And we have the other set of kernels, which use those δ values. So these are the kb2 class kernels. Of course, this is our own labeling scheme. Here, we are using them to compute the differentiated weights, i.e. the gradients of the weights. So that is what we get as output from these kernels. And these are the outputs that we are going to use. Now, of course, we will be using them to update the W values for the next forward pass.

So that is why their outputs go again, to the second instance, of the same kernels in the forward pass. So as you can see that forward pass kernels kf1 kf2, all of them had a superscript 1. Now we are just changing it to a superscript 2, just to highlight that this is a second iteration over the forward pass. So I have the $kf1^2$ $kf2^2$ representing convolution followed by pooling along the sequences and followed by the kernels in the fully connected layer.

Then again I go to the loss function kernel for a second time. So is the same kernel right. It is just operating on a second iteration right. And this loop again leads us to those sequence of kernel pipelines for computing the delta values and also the dW values, which I get just by using the deltas and the weight matrices. And then again, this continues, right. So, all the outputs of this set of kernels in the second iteration will be again passed to the forward pass kernels for their third iteration.

And that is the view. That is just like an unrolled viewed overlay dependency graph right. So essentially, I can say that in my system, I have got this set of parallel kernels which are working. Here I have the forward pass, the loss computation, the backward pass, and it keeps on happening for multiple iterations inside the loop with the updates going on. So this is like a task graph view for the convolution. And network training continues. Once it converges, we get the updated weight values.

(Refer Slide Time: 22:10)

The slide is titled "GEMM". It contains the following bullet points:

- ▶ GEMM is considered to be the core computational kernel in Deep Learning being used in Fully Connected Layers and Convolutional Layers.
- ▶ Several optimized versions of this has been developed for GPU computing architectures
- ▶ We have discussed a tiled shared memory implementation for GEMM before.
- ▶ We next focus on certain more optimizations for the same.

At the bottom left, it says "Efficient Neural Network Training/Inference". At the bottom right, it says "Soumyajit Dey, Assistant Professor" and shows a profile picture of the speaker.

But then the important part that we want to discuss, is , well, we have the task graph view, but we have also understood that all the operations that are really going on, which is basically the δ computation. It is all general, matrix multiplications, right. So as you can see the loss matrix multiplied by the gradients of the activation, and then again, using each of the layers δ s and multiplying them with the layer weights to get the previous layers δ s.

And again, continuing like that, also in each layer, using the value of the δ and the value of the corresponding image in case of the current input layer. If it is an intermediate layer, then the transformed image available for that layer, so on and so forth. So finally all of them are general matrix multiplications. So fundamentally, that is where we have to optimize.

So that is why this generalized matrix multiplication is considered to be the core kernel in deep learning, and is being used in both the fully connected as well as the convolution layers, because we have already seen earlier how that convolution operation is simply converted to the GEMM by doing a simple image to column transformation. And for doing this, there exist several optimized implementations for GEMM.

And previously, we have discussed some tile based shared memory implementations. And we will just focus a bit more on more optimized versions of the same.

(Refer Slide Time: 23:53)

The screenshot shows a presentation slide titled "GEMM". The slide content includes:

- Text: "► We consider Open Source implementations for Single Precision GEMM where the matrices are stored in column major format."
- Text: "► Before CUDA, scientific workflows demanding matrix computations used FORTRAN which considered matrices stored in column major format."

```
for (int m=0; m<M; m++) {  
    for (int n=0; n<N; n++) {  
        float acc = 0.0f;  
        for (int k=0; k<K; k++) {  
            acc += A[k*M + m] * B[n*K + k];  
        }  
        C[n*M + m] = acc;  
    }  
}
```

Reference: <https://cnugteren.github.io/tutorial/pages/page1.html>

A handwritten note in pink ink is overlaid on the slide, consisting of a large "Z" shape and a smaller "U" shape.

But there is something interesting we would like to talk about here, which is of course, this idea of multiplying large matrices has been there for quite a while. People will have been working with such large matrices and multiplications and if you see, there are popular routines for scientific computation, like BLAS, which implement underlying FORTRAN libraries, which have been doing this matrix manipulation operations very efficiently.

But one issue is that these libraries assume a column major format. So that means when we are storing the data in the matrix, we are not assuming that the storage is row wise. But rather it is like this. So it is a column major format. And the issue is that in the memory where I am storing I am traversing the matrix in a column major fashion,. Now when I am going to do column major traversal, the access expressions for the matrix are of course going to change.

(Refer Slide Time: 25:19)

GEMM

- We consider Open Source implementations for Single Precision GEMM where the matrices are stored in column major format.
- Before CUDA, scientific workflows demanding matrix computations used FORTRAN which considered matrices stored in column major format.

```
for (int m=0; m<M; m++) {
    for (int n=0; n<N; n++) {
        float acc = 0.0f;
        for (int k=0; k<K; k++) {
            acc += A[k*M + m] * B[n*K + k];
        }
        C[n*M + m] = acc;
    }
}
```

Reference: <https://cnugteren.github.io/tutorial/pages/page1.html>

CLBLAS
cuBLAS

And the important thing for us is this idea of column major format of metric storage has got carried forward into modern scientific workloads for were in popular libraries, like this CLBLAS, which is the OpenCL version of BLAS, and also cuBLAS, which is the CUDA version of BLAS

(Refer Slide Time: 25:43)

GEMM

- We consider Open Source implementations for Single Precision GEMM where the matrices are stored in column major format.
- Before CUDA, scientific workflows demanding matrix computations used FORTRAN which considered matrices stored in column major format.

```
for (int m=0; m<M; m++) {
    for (int n=0; n<N; n++) {
        float acc = 0.0f;
        for (int k=0; k<K; k++) {
            acc += A[k*M + m] * B[n*K + k];
        }
        C[n*M + m] = acc;
    }
}
```

Reference: <https://cnugteren.github.io/tutorial/pages/page1.html>

A MxK
B KxN

For example, if you just have a loop, here, we are just providing a simple example of a 3 level for loop for matrix multiplication, assuming a column major representation of the data. So what is really going to change here? So if you consider a column major representation of the data then in terms of math, you are trying to multiply matrix A and B of dimensions $M \times K$ and $K \times N$. In reality, you will have the matrices arranged in the other way around..

So, that would be like K and M and similarly here. So, there is a transformation like taking a transpose, but that is it.

(Refer Slide Time: 26:39)

The slide is titled "GEMM". It contains the following text and code:

- We consider Open Source implementations for Single Precision GEMM where the matrices are stored in column major format.
- Before CUDA, scientific workflows demanding matrix computations used FORTRAN which considered matrices stored in column major format.

```

for (int m=0; m<M; m++) {
    for (int n=0; n<N; n++) {
        float acc = 0.0f;
        for (int k=0; k<K; k++) {
            acc += A[k*M + m] * B[n*K + k];
        }
        C[n*M + m] = acc;
    }
}

```

Handwritten annotations explain the memory access pattern:

- $A \rightarrow M \times K$
- $B \rightarrow K \times N$
- $N \times L \rightarrow A[i][j]$
- $M \times K \times N \times L \leftarrow N[m][k] \times B[k][l]$

Reference: <https://cnugteren.github.io/tutorial/pages/page1.html>

That means when you are trying to access data points, so whatever was some $A[i][j]$ in my original problem That is going to change to some $A[j][i]$. Let me just write it more clearly. So it is different. But the problem is that also means that your programs are going to change. So, when you are really writing a row major code, your matrix multiplication loop internally would have been multiplying like this.

So, you may have been multiplying entities like $A[i][k]$ multiplied by $B[k][j]$, where you are iterating over k. But now that is really going to change. Well mathematically it is not, but in terms of the code, try to see what the memory access expression is. If this is the mathematical point that you are trying to use that is you are going to multiply $A[i][k]$ and $B[k][j]$ But here in terms of memory access, they will be at different locations.

Because since you are accessing in a column major way, you have to think it the other way around. And essentially you are going to use something different. So here, let us say for example, in my code the iterators are m, n and k. So, let me just consider it here. That it is all m's and n's,

like this and then since we are taking it column major, really we should have to think of the $A[k][m]$ entry.

So, that means whatever is the mathematical location i.e. $A[m][k]$, in our memory for where the storage is using column major fashion, the elementary access must be the other way around with respect to row major way and it is $A[k][m]$. So, you multiply k by capital M, the number of rows and add m and similarly for the other one the mathematical entities $B[k][n]$.

But when you are accessing the real entity in memory, since the storage is in columnar major format, so we need to we just take it the other way around. So, we will have N multiplied by the iterator here, considering that we are multiplying matrices of this size, $M \times K$. So A is of this dimension and B is of dimension $K \times N$. That is why, for the first one, the entry I was looking for was $A[m][k]$. You write it the other way around. That is you take the small k and multiply with M and add m here, because it is column major. And then for $B[k][n]$, you take it the other way around. You take small n and multiply it with a capital K and add small k. So there is a shift. So, this is a change that we have to keep in mind when we are doing the matrix multiplication operation assuming that the data is stored in column major format instead of row major that we normally do for C and similar languages.

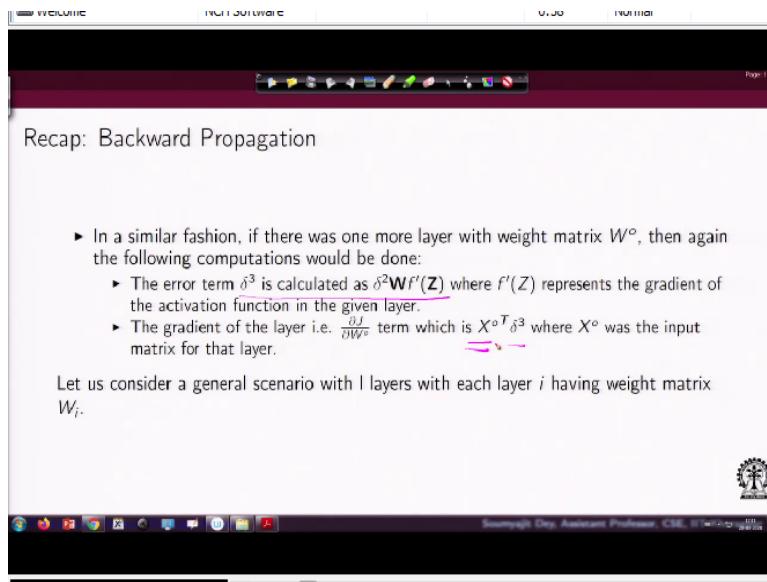
So, this is the code for C program like implementations of matrix multiplication. In the next lecture, we will touch upon the parallel versions assuming column major data storage and we will see how optimizations can be done for that. With this we will end here. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-60
Efficient Neural Network Training/Inferencing (Contd.)

Hi. Welcome back to the lecture series on GPU architectures and programming.

(Refer Slide Time: 00:27)

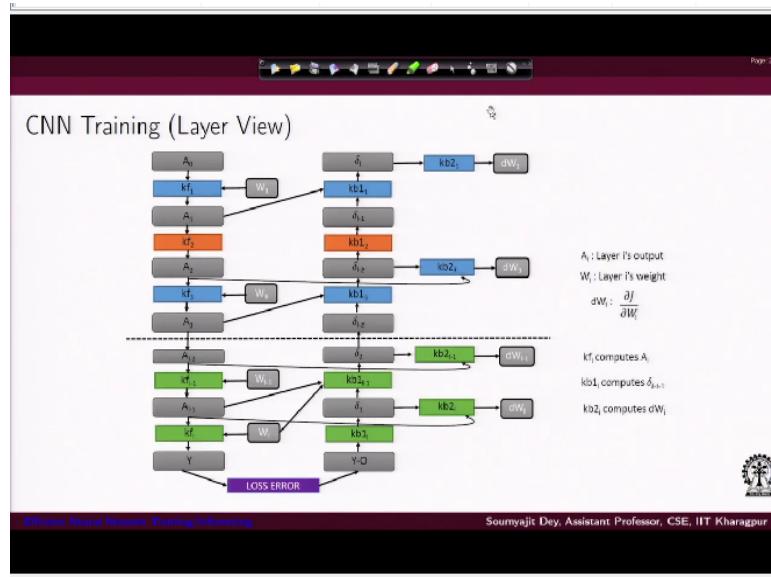


So, we will just revise once again our previous coverage. If you remember we have been talking about how to build a dependency graph for the CNN style computation. And in general, we also visited how the math for backpropagation works. What we could see in general is that we have a cascade of layers and we are trying to compute some error term for a specific layer.

So in general, the way we compute it is basically to multiply the error term i.e. I figure out the δ^2 for the previous layer, multiplied by the weight of the previous layer, and the gradient of the activation function in the given layer. And this in general would continue. This gives me the error term, you multiply that error term with the previous layers, or whatever is the input with respect to that current layer.

For example, if it is the input layer, then you multiply the currently computed error term with the input matrix. In general, if it is any intermediate layer, then the error term gets multiplied with the previous layer's output activation.

(Refer Slide Time: 01:49)



So with that, what we figured out was that if we look forward, i.e. how the pipeline computes the different indices that we discussed earlier, let there be the following kernels - The kernel kf_i , which computes the forward activation output A_i of some i^{th} layer, the kernel $kb1_i$, which computes the error term. Now, if you notice in this picture, for the error term we are writing δ_{L-i-1} .

The reason is that we are considering L number of layers and if you remember the error terms are computed backward. So, the last one would be for $i = L$ here. So, that would give you δ_1 , i.e. the error term for the last layer that you get. So, that is why for every kernel in some higher layer $kb1_i$ computes δ_{L-i-1} as per our indexing for this picture.

And we use this to compute the overall loss i.e. $\partial J / \partial W_i$ and that is something that we are representing by dW_i here.

(Refer Slide Time: 03:12)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Recap: Backward Propagation". Below the title, there is a bulleted list of points. A small video thumbnail of a man speaking is visible on the right side of the slide. At the bottom, there is a footer bar with text and a logo.

► In a similar fashion, if there was one more layer with weight matrix W^o , then again the following computations would be done:

- The error term δ^3 is calculated as $\delta^2 W f'(Z)$ where $f'(Z)$ represents the gradient of the activation function in the given layer.
- The gradient of the layer i.e. $\frac{\partial J}{\partial W}$ term which is $X^o T \delta^3$ where X^o was the input matrix for that layer.

Let us consider a general scenario with l layers with each layer i having weight matrix W_i .

Efficient Neural Networks Training/Inference
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And that, of course, is basically the computation of the error term δ multiplied by the previous output and previous activation input, which would be this.

(Refer Slide Time: 03:22)

The screenshot shows a presentation slide with a dark header bar containing icons. The main title is "Recap: Backward Propagation". Below the title, there is a bulleted list of points. A small video thumbnail of a man speaking is visible on the right side of the slide. At the bottom, there is a footer bar with text and a logo.

► During the backward pass, for the layer with weight matrix W , again two items are computed:

- The error term δ^2 is calculated as $\delta^2 = \delta^1 W^T f'(Z)$
- The gradient of the layer i.e. $\frac{\partial J}{\partial W}$ term which is $X^T \delta^2$ where X was the input matrix for that layer.

Efficient Neural Networks Training/Inference
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

(Refer Slide Time: 03:24)

Recap: Backward Propagation

- ▶ After the forward pass is completed, the first error term is calculated as $Y - O$ where Y is a column vector of true labels, O is a column vector of predicted labels.
- ▶ During the backward pass, for the layer with weight matrix W^l , two items are computed:
 - ▶ The error term δ^l is calculated as an elementwise product: $\delta^l = (Y - O).f'(Z)$ where $f'(Z)$ is a column vector where each value represents the gradient of the activation function in the given layer. Z is the output matrix of that layer.
 - ▶ The gradient of the layer i.e. $\frac{\partial J}{\partial W^l}$ which is $A^{T_l} \delta^l$ where A was the input matrix to that layer.

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

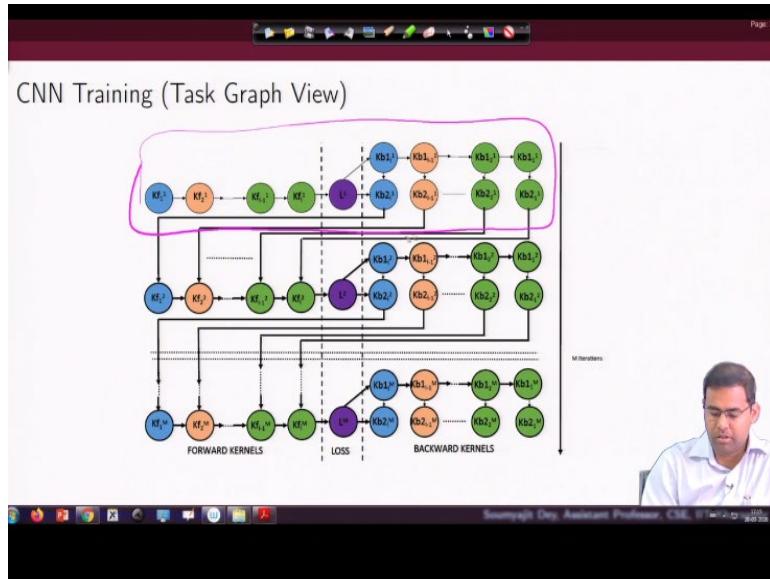
In general, this is $A^T \delta_1$ and that is what is done by this kb2 type kernel here. So, if you look into this picture in the forward pass I am just reintroducing this picture again after our last lecture, because there is some linkage here, which we need to mention. As you can see for every layer, I am introducing this index.

I mean, although the kernel is k , which is always computing the activation A_i for the i th layer, I call it k_{f_i} . For the next layer it is $k_{f_{i+1}}$, and so on and so forth. Basically, it is the same kernel, which is getting called multiple times for computing outputs of different layers. And in the backward computation, we have 2 types of kernels here, one type of kernel for computing the error term which is always outputting, the δ terms as you can see.

So I am outputting the δ terms for every kb_1 type kernel. And then these δ terms are used by these kb_2 type kernels to give me the differential loss i.e the differential weight. So essentially, though, this is basically multiplying these δ s with the A^T s. So that is why the dependency is from the δ term. There is an input dependency and there is also a dependency from this A_{i-1} .

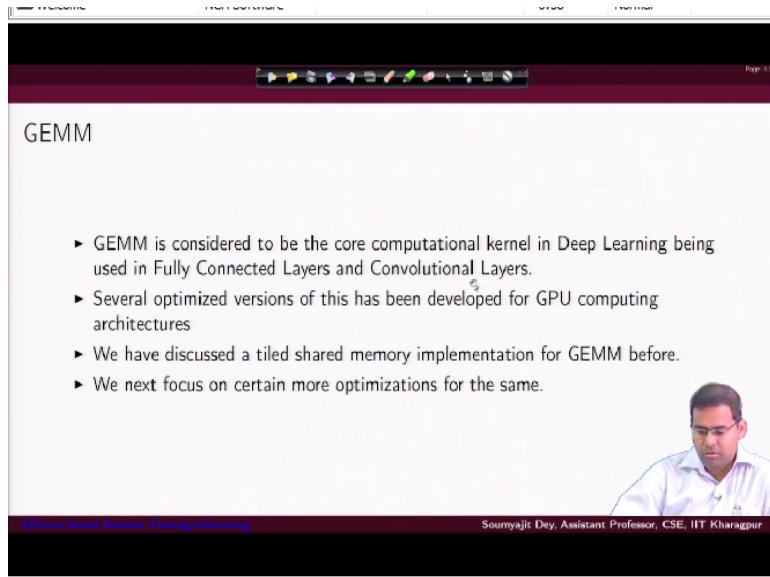
So, they are giving me the dW_i terms which I will add to these W 's and update them and again run the forward propagation. That will be again followed by the backpropagation.

(Refer Slide Time: 05:25)



So, pretty simple, if we now progress with this, this was our task graph view. So essentially, as we discussed that this part is our set of kernels, which are doing the entire computation. But it keeps on repeating. The computation is a sequence - forward pass, then loss computation, then backward pass, and then again, it repeats. Again, the backward pass outputs are used to update weights, and then you are going to run the forward pass and then the backward pass so on and so forth.

(Refer Slide Time: 05:56)



So with this, what we figured is the most important computation we will be requiring is the GEMM computations for this kind of neural network training.

(Refer Slide Time: 06:06)

GEMM

- We consider Open Source implementations for Single Precision GEMM where the matrices are stored in column major format.
- Before CUDA, scientific workflows demanding matrix computations used FORTRAN which considered matrices stored in column major format.

```

for (int m=0; m<M; m++) {
    for (int n=0; n<N; n++) {
        float acc = 0.0f;
        for (int k=0; k<K; k++) {
            acc += A[k*M + m] * B[n*K + k];
        }
        C[n*M + m] = acc;
    }
}

```

$\frac{M \times K}{A} \times \frac{K \times N}{B}$

Reference: <https://cnugteren.github.io/tutorial/pages/page1.html>

So, overall our observation had been that for doing, this GEMM computation will require to use our parallel programs and the major scientific libraries that are available eg. BLAS and in modern times cuBLAS actually use these Fortran based ideas for doing the matrix computation, You assume that data is in a column major format. Now, what would that really mean? Let us understand.

So, like we discussed that whenever we are talking about column major format, one of the obvious things that we considered for the the matrix is that every i,j^{th} term in your mathematical matrix can be accessed with a j,i index. Consider that this data is stored in the column major format and as you can see, that is what is happening. So, you are multiplying 2 matrices - A of dimensions $M \times K$ and B of dimension $K \times N$. So, you see the access expressions are a bit different. So, we are more habituated to writing a loop where I have this K iterating for a fixed M, And so, essentially the way we do that is I will have M times some small m which is the dimension. So, I have $M * m + K$ multiplied by B.

And the index for B would be of course, the column index. So $k * K + n$. But as you can see that it is opposite here. The reason is that this is column major format, like we already discussed. So one simple way to think would be that we will just consider the data entries i.e. whatever is your actual ij^{th} mathematical data in your computation, it can be accessed, once you flip the indices with j and i.,

(Refer Slide Time: 08:38)

The slide is titled "GEMM". It contains the following text:

- We consider Open Source implementations for Single Precision GEMM where the matrices are stored in column major format.
- Before CUDA, scientific workflows demanding matrix computations used FORTRAN which considered matrices stored in column major format.

```
for (int m=0; m<M; m++) {  
    for (int n=0; n<N; n++) {  
        float acc = 0.0f;  
        for (int k=0; k<K; k++) {  
            acc += A[k*M + m] * B[n*K + k];  
        }  
        C[n*M + m] = acc;  
    }  
}
```

Reference: <https://cugteren.github.io/tutorial/pages/page1.html>

A hand-drawn diagram illustrates the column-major storage of matrix A. Matrix A is shown as a grid of elements $a_{m,n}$. The diagram highlights the columns of the matrix, indicating that the elements are stored sequentially by column. Arrows point from the indices $a_{0,0}, a_{1,0}, \dots, a_{M-1,0}$ to the first column, and from $a_{0,M}, a_{1,M}, \dots, a_{M-1,M}$ to the last column. The matrix is labeled with A at the top left.

But just to have a more clear picture, consider the real arrangement of the data. As we know that we are going to now have that in memory, we have everything stored in a consecutive block. And for our usual C or similar style languages, we have row major format. That means you have a row followed by the second row in that way. But now, since we are considering column major format, what does that really mean?

So consider some matrix A. So, this is a normal mathematical matrix that we are trying to write and you have the data of the row 1 row 2 and all that. But when you are storing it, you are not storing it in a C style arrangement in the memory, but rather you are storing it into column major format. So, if I just write the data arrangement here, you are going to store the first column here in the first row.

So, that means you are essentially storing A. Let me use the 0 indices here, because finally I will be writing a C program to access them as $A[0][0]$. Then the second element in the column $A[1][0]$ like that and finally, you will be having $A[M-1][0]$ where M is the dimension since we are considering A to be of $M \times K$. So, $M - 1$ for the 0th column, right, we move on to the next. So, A for the second column right.

So, that was $A[0][0]$. So, now, so if I just write it you have a 0,0 here, and $M-1,0$ here. The next would be 0,1 and in this column what do you have? We have a $M - 1, 1$. So $A[0][1]$ and it all ends with $A[M-1][1]$. So that is how it is. Now, when you are really trying to access the data, i.e. you are going for some mk^{th} element here, what would be the location for that mk^{th} element? That is the problem. So as you can see, for the mk^{th} element what do I really need to do? Well, I have to make N number of skips over this data. So I want some access mk here. Sorry, this is M here. So I have to shift to the k th column. That means I have to skip through this k times M number of elements here.

And then I get to the column where I have the mk^{th} element. So for that, as you can see, I have to multiply k with M . That takes me to that position where I have it in that column. Now the elements of that column are located like this. I have a $0,k^{\text{th}}$ element, a $1,k^{\text{th}}$ element like that up to a m,k^{th} element. So that is why the shift is by M . I am just trying to make it clear that it is very simple. You just flip the i,j indexes, but physical in terms of the memory storage. This is what it would mean. You have to multiply k with M since you are shifting over k hops of columns here.

(Refer Slide Time: 13:06)

GEMM Optimization 1: Tiling

```

1  __global__ void GEMM1(const int M, const int N, const int K,
2  const float* A, const float* B, float* C) {
3      const int row = threadIdx.x; // Local row ID (max: TS)
4      const int col = threadIdx.y; // Local col ID (max: TS)
5      const int globalRow = TS*blockIdx.x + row; // Row ID of C (0..M)
6      const int globalCol = TS*blockIdx.y + col; // Col ID of C (0..N)
7      __shared__ float Asub[TS][IS]; __shared__ float Bsub[TS][TS];
8      float acc = 0.0f; const int numTiles = K/TS;
9      for (int t=0; t<numTiles; t++) {
10         const int tiledRow = TS*t + row; const int tiledCol = TS*t + col;
11         Asub[col][row] = A[tiledCol*M + globalRow];
12         Bsub[col][row] = B[globalCol*K + tiledRow];
13         __syncthreads();
14         for (int k=0; k<TS; k++)
15             acc += Asub[k][row] * Bsub[col][k];
16         __syncthreads();
17     }
18     C[globalCol*M + globalRow] = acc; // Store the final result in C
19 } // Launch Parameters: <<(M/TS,N/TS),(TS,TS)>>

```

So with that, this would be our normal code for the multiplication, where I am also assuming that we will do some tiling based optimization. Now, of course, we are familiar with tiling based optimizations for matrix multiplication. So, this is basically that code only. So what do we really do? We use this matrix multiplication kernel. We compute the rows and columns which are the,

for the threadIdx.x and threadIdx.y. And we use them to figure out what is the global row and column values. So, in general, if you recall, our idea of doing normal matrix multiplication, we are going to launch a block of threads, where every thread block will try to compute the elements for a tile. Inside a tile, or in the corresponding view for a block of threads, every thread is trying to compute one element in the tile. And what was the way that we progress?

(Refer Slide Time: 14:16)

The previous kernel can be improved by increasing the amount of work of each thread (thread coarsening). The PTX code for the inner k loop (lines 14-15) for two iterations is as follows:

```
ld.shared.f32 %f50, [%r18+56];
ld.shared.f32 %f51, [%r17+1792];
fma.rn.f32 %f52, %f51, %f50, %f49;
ld.shared.f32 %f53, [%r18+60];
ld.shared.f32 %f54, [%r17+1920];
fma.rn.f32 %f55, %f54, %f53, %f52;
```

It can be observed that only one out of every three instructions is useful! Increase work per thread to reduce number of local memory accesses.

Scalability: Assistant Professor, CSE, IITM

So if you just remember, let me just redraw the picture again. So let us say this is your B, this is your A and you are trying to produce an output matrix C here. If you break to tiles, the tiles indirectly define the way you are going to launch the thread blocks. So for one block of threads is corresponding to computation of one tile. So every thread which is going to compute this element will have to do a computation of elements here.

It has to essentially multiply the elements here with the elements here, right. And how does it really do it? Well, it does by coordinating with other threads i.e. for this, the thread is first responsible for loading this data. It will be using 2 shared memory locations to load these data points.

It will use this inside this shared memory. It will be loading these data points and all the other threads in the block will be loading the other thread data points. Once that is done, if you look

back into the code, this is the position where all those loadings will be done by all the threads collaboratively. So, here as you can see, so, we are in the first iteration of this loop.

All the threads have loaded the first data of the first tile right. And then they go to this second loop here. In this loop what is going on? Now, this thread is going to compute a multiplication of the data, which has been loaded by all threads here i.e. the data that has been loaded by every thread here in the shared memory. So this multiplied by this gives you the partial sum. In the next iteration of the outer loop for all the threads in the block, we load the data for the intermediate block here, and then this block here. And then again, every thread will perform the corresponding activities. So that now after bit ends, the thread, which is here, has performed a partial sum of data up to this much and after this much right here. And it continues like this. So that was our idea of tiling. Just a small recall here.

So if you remember, so this is the position where one thread is loading 2 data points into the shared memory i.e. Asub and Bsub. And it is waiting here for every other thread in the same tile to do the load. And then it is falling into this loop, where it is progressing with all the data in that row for Asub and in that column of Bsub to do the computation of the partial sum. And once everything is done, for all the tiles, it will update into the global memory.

So that is how things were going on for normal tiling. No. Again, if you remember that since is the column major formatting, you can see the access expression here for Asub and Bsub is just reverse to what we have been seeing earlier. So that is why, by tradition, k comes first, because it is in the column major now, right. So it is $Asub[k][row]$ instead of $Asub[row][k]$. And similarly, $Bsub[col][k]$ instead of $Bsub[k][col]$. This is just the opposite from earlier.

Now, earlier, if you remember, we were happy with the result of this kind of tiling based matrix multiplication kernel. But now we will try to see whether this can be accelerated further. Well, of course, we have understood that this is the most important operation for neural network training and it is also is going to be popular with many other kinds of workloads.

So this needs to be further optimized. So right now, all of our efforts would be in terms of optimizing this GEMM kernel. Now I will just point out once again that the difference when we are speaking specifically with respect to the GEMM operations for the cuBLAS or similar kinds of libraries for linear algebra, we will be using this column major representation. We will because you need to remember that is going to also play a role in our understanding of how things are going to happen here. So, for A_{sub k} comes first followed by a row and for B_{sub} you have col followed by k, instead of the reverse that we will be expecting in general to happen. Let us understand why is this important.

Well, that would mean that the way that the data is getting access is a bit different, right. So that means here when I am accessing the data. Where is the access from? The access of this data is from shared memory. The access of these data are global loads to share memory. This is a load from the shared memory to the internal registers for doing the fused multiplication and add right.

There is a call to fuse multiply and add unit which is there in the GPU. Now, this global loads are going to take their own time. Now, again as we know that one of these loads will be coalesced and the other will be not because one of them will be row wise and the other would be column wise. But here as you can see, this is for from the shared memory to the global to the fuse multiply and update unit with the output going to a register of type acc and the variable name for the register is here. Now, try to analyze how this program performs. Let us just have a look into the intermediate code of the PTX code that will be generated for this kernel. So this peeks into the PTX code for the inner loop lines 14 to 15. That is it right. So this is the inner loop.

As you can see, we are doing 2 shared loads from the shared memory because of course A_{sub} and B_{sub} are in the shared memory. And we are then engaging the fuse multiply and add units right. So, in each iteration of the loop, per thread shared activity is as follows. You are doing 2 shared loads followed by one fuse multiply add operation, again. In the next iteration of the loop, you are doing 2 shared loads and the real operation we are doing is one fuse multiply add.

So as you can see that, it is basically like only one out of every 3 instructions is useful for computation. But the other 2 are for memory accesses to the shared memory right. So if I assume

that I have a significant number of fuse multiply add units, they are not really getting used. So the fundamental reason is that I have lesser occupancy, because I am not making good use of the number of threads I have launched.

Rather than that, I can make full utilization of the GPU bandwidth by increasing the amount of work that I am delegating per thread , because here as you can see, the third activity I am really getting is one fuse multiply and addition operation for every 2 shared loads. We like to increase it. How would that be possible?

(Refer Slide Time: 22:50)

```

1 __global__ void GEMM2(const int M, const int N, const int K,
2 const float* A, const float* B, float* C) {
3 //Code for thread identifiers
4 //Code for initializing Local memory
5 const int numTiles = K/TS; float acc[WPT]; // WPT-> Work per thread
6 for (int w=0; w<WPT; w++) acc[w] = 0.0f; //initialization
7 for (int t=0; t<numTiles; t++) {
8     for (int w=0; w<WPT; w++) { //RTS = TS/WPT : Reduced Tile Size
9         const int tiledRow = TS*t + row; const int tiledCol = TS*t + col;
10        Asub[col + w*RTS][row] = A[(tiledCol + w*RTS)*M + globalRow];
11        Bsub[col + w*RTS][row] = B[(globalCol + w*RTS)*K + tiledRow];
12    }
13    __syncthreads()
14    for (int k=0; k<TS; k++)
15        for (int w=0; w<WPT; w++)
16            acc[w] += Asub[k][row] * Bsub[col + w*RTS][k];
17    __syncthreads()
18 }
19 for (int w=0; w<WPT; w++) C[(globalCol + w*RTS)*M + globalRow] = acc[w];
20 }// Launch Parameters: <<<(M/TS,N/TS),(TS,TS/WPT)>>>

```

So for this, we will start to apply our thread coarsening techniques that we have seen earlier. So fundamentally what we are really trying to do? We will try to ask that well, even inside a tile, I did not have the same number of threads in the block , as is required for computing the number of elements in a tile rather than that in my thread block launch. Let me have a smaller number of threads for computing the entry of the entire time.

That would mean now for one thread, I am not just delegating it the activity reserved for computing one final output in the matrix. But rather I am asking it to compute more than one entry in the final matrix.

(Refer Slide Time: 23:40)

```

ld.shared.f32 %f82, [%r101+4];
ld.shared.f32 %f83, [%r102];
fma.rn.f32 %f91, %f83, %f82, %f67;
ld.shared.f32 %f84, [%r101+516];
fma.rn.f32 %f92, %f83, %f84, %f69;
ld.shared.f32 %f85, [%r101+1028];
fma.rn.f32 %f93, %f83, %f85, %f71;
ld.shared.f32 %f86, [%r101+1540];
fma.rn.f32 %f94, %f83, %f86, %f73;
ld.shared.f32 %f87, [%r101+2052];
fma.rn.f32 %f95, %f83, %f87, %f75;
ld.shared.f32 %f88, [%r101+2564];
fma.rn.f32 %f96, %f83, %f88, %f77;
ld.shared.f32 %f89, [%r101+3076];
fma.rn.f32 %f97, %f83, %f89, %f79;
ld.shared.f32 %f90, [%r101+3588];
fma.rn.f32 %f98, %f83, %f90, %f81;

```

For 8 iterations of the inner k loop, there are 8+1 loads from the local memory for 8 FMA (instead of 8+8).

So if I just draw a picture, it would be like this that let us say this is my final output. And I am going to launch tiles and all that. So earlier, whatever was the activity for one thread? Let us say I am asking one thread to do the computation for 2 elements, or in general, maybe more than 2 elements. So that amount depends on what is my amount of coarsening that I am pushing into the thread. Let us say 8 elements.

So we will see how it makes progress. So now as you can see, I will introduce a variable WPT work per thread. So of course, one thread will be now accumulating values for more than one output. So instead of keeping acc as a single value i.e. a single local variable, which maps to a single local register per thread, I will make it an array acc[WPT], so that the thread is accumulating values for multiple outputs.

And then I hope you can understand that things are going to be simple. It is like so earlier, you had this loop. Let us just compare the previous and the current implementation. So earlier, you had the outer loop where you were doing the loads followed by the accumulate. Well you still have the outer loop. This is your simple loop just for initializing the acc array.

But then this is your normal loop of the previous work. But here earlier you were just doing 2 shared loads, right, this one and this one. Now you are doing those 2 shared loads inside this loop. This loop is running from 0 to WPT - 1. That means you are performing that many shared loads for both A as well as B, right. And you have to compute the shared load indices by looking into the values of the offsets and all that which I think you can easily figure out.

Just to understand that for per thread shared activity, we have increased the number of loads here. Well how does that help because till now, we have not seen how the compute increases per thread because we have increased the number of loads per thread. Well, now, let us look into the compute part. So, after the loading is done, here, you have the compute. Earlier, your compute was just this loop followed by this multiplication, right.

But now, inside this loop, you have another loop where you are doing this multiplication. So earlier, what was the situation? Let us say for this entry, you are computing this multiplied by this. But now you have 2 entries. So you are going to compute this multiplied by for one thread, the amount of activities, let us say you have 4 entries, you are going to compute by this thread. So you have to load data from here.

And I mean, you have to multiply the data from here along with the data from all these 4 columns, right. So there is the per thread shared activity you have inside the loop now. That is why you have this internal loop. But what is the beauty of this internal loop? The beauty of this internal loop is that of course, here when I am drawing I am using the normal matrix representation, you have to just switch it accordingly.

Because as you can see that the indices are again in column A and all those formats, but if we are trying to understand as you can see that when this loop is making the computations for these 4 points, it can take one value here. It has to multiply that value with the value here, here, here and here. So one load of this and multiple loads of data from Bsub, and that many number of multiplications, go to the next iteration of the loop, one load from Asub, multiple loads from Bsub and that many multiplications.

So there is the good thing. Because since you are delegating more activity for this thread for that one load is common for multiple loads of data from the other array right. So that is the good thing since this is constant, you have less number of loads at least for one of the inputs and that increases your per thread computation. Now, I hope this is clear. So, the first good thing is you have coarsened the amount of activity per thread.

And the coarsening is more for the compute part rather than the load part. So, if I just compare it with the previous implementation, the amount of loads from shared memory to the registers for doing the fuse multiply and add are drastically getting reduced by a factor which is the thread coarsening factor. Compare the number of loads here, in the PTX, considering that you have coarsened with a value of 8.

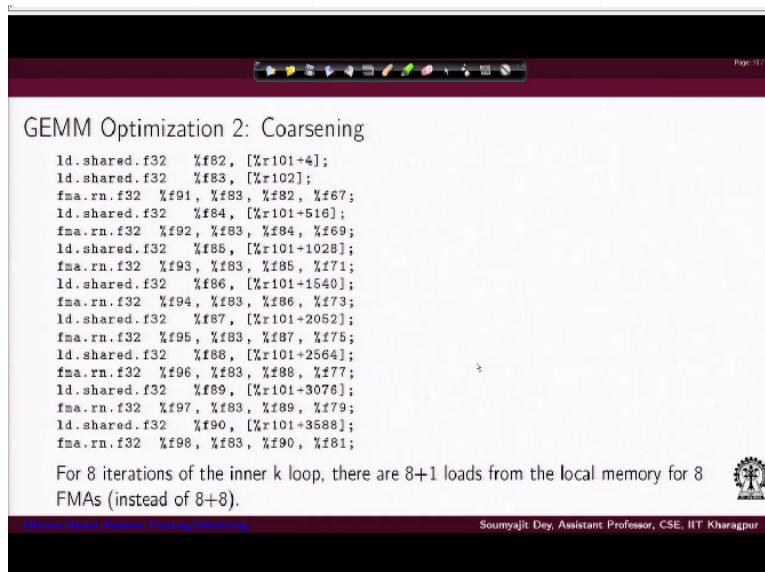
So for 8 iterations of the inner loop what is going on? You load this Asub once. You have to load Bsub 8 number of times to achieve 8 number of fuse multiply adds. Earlier the number of loads was $8 + 8$, but now it is $8 + 1$ right. So, this is not the inner 8 iterations. It is basically the innermost loop actually. So, with this optimization will be ending this lecture and in the next lecture we will see some more optimizations. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-61
Efficient Neural Network Training/Inferencing (Contd.)

Hi. Welcome back to our lecture series on GPU architectures and programming.

(Refer Slide Time: 00:28)



GEMM Optimization 2: Coarsening

```
ld.shared.f32 %f82, [%r101+4];
ld.shared.f32 %f83, [%r102];
fma.rn.f32 %f91, %f83, %f82, %f67;
ld.shared.f32 %f84, [%r101+516];
fma.rn.f32 %f92, %f83, %f84, %f69;
ld.shared.f32 %f85, [%r101+1028];
fma.rn.f32 %f93, %f83, %f85, %f71;
ld.shared.f32 %f86, [%r101+1540];
fma.rn.f32 %f94, %f83, %f86, %f73;
ld.shared.f32 %f87, [%r101+2052];
fma.rn.f32 %f95, %f83, %f87, %f75;
ld.shared.f32 %f88, [%r101+2564];
fma.rn.f32 %f96, %f83, %f88, %f77;
ld.shared.f32 %f89, [%r101+3076];
fma.rn.f32 %f97, %f83, %f89, %f79;
ld.shared.f32 %f90, [%r101+3588];
fma.rn.f32 %f98, %f83, %f90, %f81;
```

For 8 iterations of the inner k loop, there are 8+1 loads from the local memory for 8 FMA (instead of 8+8).

Efficient Neural Network Training/Inferencing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So if you remember in the previous lecture, we have been talking about optimizing the GEMM computations. So we discussed one optimization, which was application of thread coarsening over a tiled matrix multiplication.

(Refer Slide Time: 00:39)

GEMM Optimization 3: Wider loads

- In the previous implementation we increased the amount of work in the column-dimension of C.
- The same optimization trick can be done for the row-dimension
- The additional advantage for optimizing across the row dimension is using wider data-types.
- Increasing the work per thread (WPT) in the row-dimension of C can be done by considering vector data-types instead of loops over WPT.
- NVIDIA GPUs do not support vector operations (such as multiply or add) in hardware but possess special wider load and store instructions both for the off-chip and the local memory.

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, let us start with the next optimization. So, if you remember in our previous optimization, we increased the amount of work per thread, and that was in the column dimension of the output, right. Well, why do we say it is the column dimension? Because when we are giving the examples, essentially we are talking in rows, since it is basically in the column major format. So Whatever is the row in the mathematical matrix, it is actually in the column dimension here, right. Now, we can just apply the same optimization trick for the other dimension right.

(Refer Slide Time: 01:24)

GEMM Optimization 3: Wider Data Types

```

19     for (int k=0; k<TS/WIDTH; k++){
20         vecB = Bsub[col][k];
21         for(int w=0; w<WIDTH; w++) {
22             vecA = Asub[WIDTH*k + w][row];
23             switch (w) {
24                 case 0: valB = vecB.s0; break; case 1: valB = vecB.s1; break;
25                 case 2: valB = vecB.s2; break; case 3: valB = vecB.s3; break;
26                 case 4: valB = vecB.s4; break; case 5: valB = vecB.s5; break;
27                 case 6: valB = vecB.s6; break; case 7: valB = vecB.s7; break;
28             }
29             acc.s0 += vecA.s0 * valB; acc.s1 += vecA.s1 * valB;
30             acc.s2 += vecA.s2 * valB; acc.s3 += vecA.s3 * valB;
31             acc.s4 += vecA.s4 * valB; acc.s5 += vecA.s5 * valB;
32             acc.s6 += vecA.s6 * valB; acc.s7 += vecA.s7 * valB;
33         }
34     }
35     __syncthreads()
36 }
37 C[globalCol*(M/WIDTH) + globalRow] = acc;
38 } // Launch parameters: <<<(M/TS, N/TS),(TS/WIDTH, TS)>>>

```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

But how do you really do it? Well, the way you can do it is by using wider data types. Now, what is a wider data type? We will soon see. But let us understand how this is really going to help. First of all for these optimizations, we will not carry over the previous optimization here.

Although we understand that using both optimizations together can help. So now let us consider that we are just considering how wider load instructions can really help me.

So this is actually done by using vector data types instead of loops over the WPT variable that we discussed earlier. Now, in NVIDIA GPUs, we do not have support for vector operations like vector multiply or vector add, like vector operations in CPUs. And of course, we do not need them because we are having multiple scalar units, which work using the idea of warps.

So we do not have vector operations. But there is something called wide load and store instructions both for the off chip, as well as for the shared local memory. So what does that mean? I am not going to have multiple addition or multiplication operations on vector data, but I can have vector kind of wide loads. I can load multiple data together using a suitable vector data type.

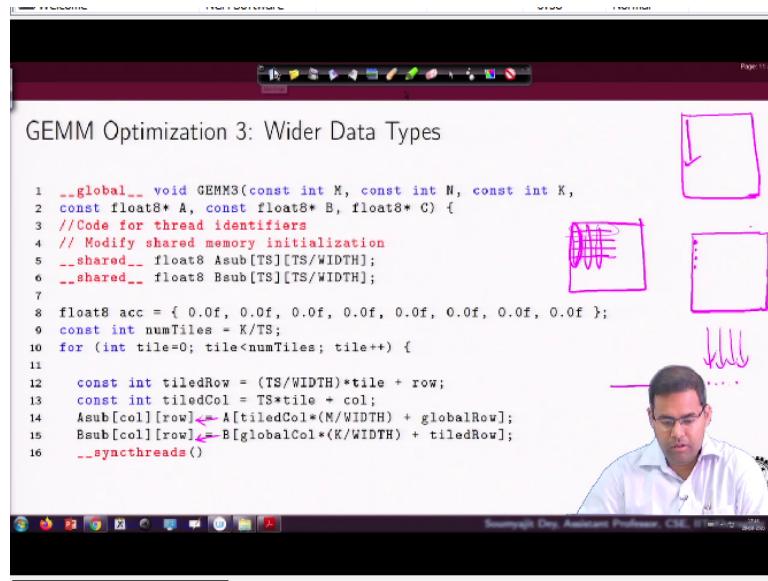
Let us see how that really works. So when I am really using a wide data type, one example would be this float8. So essentially, I declare float8 kinds of variable. Let us say for Asub and Bsub array which are in my shared memory, I am declaring them as of type float8. And that would mean, each element in that array can accumulate, 8 constituting floating point data.

So that also means , I do not need that much width of Asub that was there earlier. So I decrease the column dimension here for Asub by TS/WIDTH. Where TS is the tile size. So that is how my Asub and Bsubs will get reduced with respect to columns. But on the contrary, what is happening each of the elements are getting wide by 8. Each entry in the matrices will now be having 8 consecutive floating point data.

Again, what are we really trying to do? We are trying to increase the total amount of data i.e. values that get loaded both from the global memory. So from global memory I am using one load instruction and making it explicit that you load these many together and put it in a wide variable. Similarly consider this as a shared memory and you are trying to load it into the GPU register for doing some operation.

That will also be a wide load from the shared memory. So now, of course, the usefulness of the instructions can be exploited only if we can change our programs with W. So what will be the changes? Let us understand the implications here.

(Refer Slide Time: 04:54)



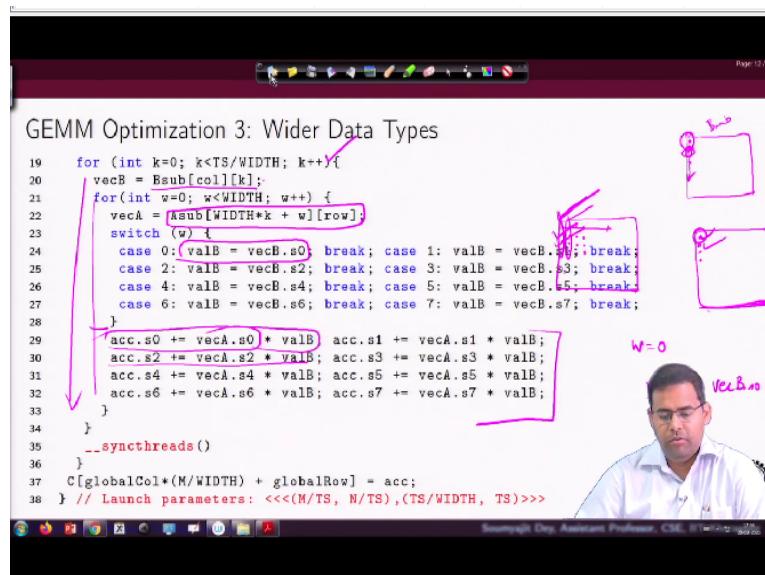
So what we are really looking at now is something like this. So let me first draw one tile. So in this one tile inside that entire C matrix, I am saying that, the thread should compute, let us say a wide number of let us say some multiple entries together. And again, I will say I am trying now in the column dimension. Because now they are going to be these things that will be accessed in parallel. So, let us see how the wide loads are going to happen here. Again, we have the outer loop iterating from 0 to number of tiles, which remains the same. And here I have the loading part to the shared memory. If you remember in the previous coarsened version, we introduced a loop here. That is not going to be there.

Because now I am not applying thread coarsening. I am just playing with wide loads. So, now, when I am defining this operation that will from global A and B arrays load values to Asub and Bsub. So, since the data type here is this float8 , I will have the wide vector loads operating. So, with each load I will get multiple let us say since this is float8, I will get 8 consecutive float values getting loaded into the tiles.

But then there is a problem. Let us understand. So, the mathematical arrangement of the data was like this and like we discussed you will have 1 row here to traverse and you have multiple columns to traverse right. So, you multiply like this - these in the mathematical domain, but now when you are trying to work it out for these many data points here, what do you really need in the input tiles Asub and Bsub?.

So, if you have to get this done, then you need data here like this. And you need values like this. There is a requirement. You want to work on them. And since it is a column major format, these values are all located in one entry of the matrix, right. But you do not really want to multiply this entry with this entry. Because, you really want to multiply these entries with these entries. Get this, but using normal mathematical matrix multiplication. I hope this is clear. Let me draw this thing again in a better way.

(Refer Slide Time: 08:09)



So we will just say that I want to compute more elements in the matrix row. So I will employ a wider load instruction. This will help me to compute multiple values together. If I employ the wider load instruction, then what I get here in the Asub and Bsub, is data like this. And due to collaboration I am drawing in terms of 4 instead of 8, just because of course, we need to get the meaning correct.

And the wider load instructions here are giving me values like this. The point I am trying to make here is to compute this, we understand that we have to go like this. But how is the data there in the variable? There is a problem. The data is there. These consecutive data points are there in one location of Asub, the next set of values are there in the next location of Asub. That would mean, you look inside the value and segregate individual parts from the value and then use them to do component wise multiplication and compute these values right.

Let us figure out what does that really mean. So now after the loading is done, which is very simple, let us go to the compute loop and see how things are going to change. So of course, this is the outer loop for the computation.

(Refer Slide Time: 09:51)

GEMM Optimization 4: Rectangular Tiles

- ▶ The Tesla K40 GPU which has 48KB of shared memory per SM, on which multiple thread blocks can execute.
- ▶ For a 32×32 tile, we consume $2 * 32 * 32 * 4 = 8KB$ per work-group, so there is some headroom left.
- ▶ Since both matrix A and B share the dimension K, we create rectangular tiles.
- ▶ We can also pre-transpose the matrix B using the optimized transpose kernel used before.

```
#define TSM 64           // The tile-size in dimension M
#define TSN 64           // The tile-size in dimension N
#define TSK 32            // The tile-size in dimension K
#define WPTN 8             // The work-per-thread in dimension N
#define RTSN (TSN/WPTN)    // The reduced tile-size in dimension N
#define LPT ((TSK*TSM)/(RTSM*RTSN)) // The loads-per-thread for
```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

But the issue is with the inner loop because as I am saying, that for computing each of these values, I need access to consecutive values in Asub, because each of the values are of vector type. And I need to look into each of the components. So what do we do. Well, we take a piece of value in some vector B, which is fine, because I need these values which are anyways together.

But for Asub, I have to have a loop, which will run through the entire width, which is 8 in our implementation. Consider it 4 right. In each iteration, I take 1 element of Asub, in the first iteration. What am I really supposed to do? I am supposed to look into each of these components individually, multiply them by each of these components, and store them as partial sums here.

In the next iteration of this inner loop I am going to take the next Asub value which is again, accessible as one of this. But then I have to look inside into its components. And I have to multiply these components, again with the pieces of values. I hope this is clear. What is the problem? These values are in columns, but actually I need to compute like the rows.

So I have to look inside the components. That is what is done by the switch case. So here, as I progress through this inner loop, with respect to W equal to 0 to width, what really am I doing in each iteration of this loop? I pick up one of these Asub values, and then I sample out. Well, for each thread, I am trying to sample out which component of this vector I would be interested in. I know that I have picked it up and I have stored it in vecA. But I have to realize now that for the other vector, which are the components with which I am going to multiply them? Because as you know that the first component of vecA needs to be multiplied with the first component of the vecB. Of course, let me first define. So vecB is for storing this right. And vecA is storing this value right now. In the next iteration, vecA will store the next value and like that. Also, we need to figure out a way to access the components. Now for float data types, the way that is done is that you just do a normal structure like computation. For this vector vecA.s0 gives you the first component vecA.s1 is going to give you the second component, so on so forth. So what do I really need to do.

So you take vecA and you need to multiply. I am speaking for W equal to the 0 case. So what do you need to do? You need to take vecA.s0 component and multiply it with vecB.s0 component. So for W equal to 0, I need vecB.s0 component. So that is why I pick up this and here what am I doing? I am multiplying vecA.s0 component with this and thus giving me the partial sum for this.

But parallelly I have other things also. For this right now I have with me in valB, vector B's s0 component. Well, do I need it for some other computation. Of course, I would need it here for s0. But in future when I pick up the next vecA, I would again need it, right. So, what is happening when I am accumulating for the s0 component? This is what I am doing. But what about the other components?

So as you can see, right now, for all the other components, let us say this component. I would need vecA's component s2. Let us say, the third component that is s2. I would need vecA's s2 component to get multiplied with s0, which is already in valB. So that is what takes place here and similarly for all the components. I hope we are making some sense here. So essentially, what am I really doing?

By using this switch case, we are taking out this variable, the first value and then by all these accumulation statements, we are multiplying this 0 component of vecB with all the different components of vecA. And that is what I require. I required all these values to get multiplied with this as a partial sum for here. And this continues with the loop making progress.

Now, I will take the next value of vecA and for each component of vecA, I will have to sample out what to do. Which component of vecB to pick and which will be the next component? And I will now use this component for all the other ones. So again, as you can see, there is a lot of saving happening. Like with one load, for this vecB, whatever component I am getting, I am choosing that appropriate component.

And I am doing the multiplication of that component with everything else, for the first component of the vector A, and in that way this is computed. So just to summarize for this multiple values, for each of them, I am making a vector load like this. And I am multiplying these vector's different components with the components that are here. I am just choosing which component to multiply.

Because if we can understand for computing this value, I need this one to get multiplied with this. For computing the next value I need the next one that is vecA.s1 to get multiplied with the same value. For the next one again, I need the vecA.s2 to get multiplied with the same value. That is why I am choosing this value here using the switch case. And I am using that valB in all the accumulations.

Again as I progress, I will choose one component here. I will choose the second component, right for vecB which is this value. By the way, it is important to note that vecB is loaded once and is getting used entirely in this loop. vecB is not changing. In one iteration of the loop as long as I am inside this inner loop, I have this vecB.

So once I load it for each of the components , I am just figuring out which component to multiply with the different components of vecA. And I am just using it. So, I load vecB once, I load vecA multiple times - the number of times that is required to fully do the compute of the final entries. And in each case, I select the suitable component from vecB and do a multiplication with all the components of vecA. And that is how it progresses.

So, as you can see that this is just a different idea with respect to thread coarsening. What we are doing is that for per thread, we are doing wider loads so I get more amount of data per load and then that gives me a problem like the data is all in columns. To use them in a suitable way I make use of this intelligent loop structure here.

Now, coming to the next optimization, which is rectangular tiles, so, what is this. So, if we see in our previous examples typically we use tiles of size 32 x 32. Now that requires how much memory? Well, 2 of the shared memories, each of size 32 x 32 right. So $32 \times 32 * 2$ multiplied by 4 bytes because it is float.

So that is 8 KB per work group or let us say per block. So that is not much. Why? Because even if we consider Kepler's old GPU like Tesla K40 it has 48 KB of shared memory. So, I have more shared memory there. So, what I can do is, I can create bigger tiles. So, it is a small mistake here.

So, this is called creation of rectangular tiles. And also there is something important that I can do. Since I am multiplying A and B together you can see that from B, I am doing an access and from A that data, since it is a col major representation. So, that means for A the columns of A are in memory right. But for B, I need the data in the other way around, right.

So, as we know that when you are multiplying matrices and loading data for one, I will get this memory coalescing effect for loads. For the other I will not get it. But a good way to optimize that is that well, we transpose the other matrix. So, in this case, I can just transpose B right. So, because if you see that for B, the access is not memory friendly. So, I will just transpose B.

Because we have already figured out earlier in our discussions with respect to matrix transpose computation, it is really possible to do very efficient transpose computation, where the amount of time required for doing a transpose is not too high with respect to a normal memory copy. So assume that we first do a transpose so that the matrix B is available in the global memory in a transposed way.

Well, what does that help in? Now when I have B transposed, when I am bringing data, I get the good effect of memory coalescing happening for both matrices A and B. So now let us just go through some basic definitions. Let us say TSM defines the tile size in dimension M, TSN for dimension N and the tile size in dimension K is 32. Also, we will increase the work per thread.

So, we will bring in the previous optimizations. So, we will bring in the work per thread coarsening optimization, let us say 8 and with this what is the reduced tile size? So, as you can see that since I have more work per thread, so, the tile size in the dimension N would be used if you just take the actual dimension divided by the work per thread WPT.

So, in general, what is the total number of loads per thread for a single tile? How many loads you really have to do in a single time? Well, that would be the tile size in dimension K multiplied by the tile size in dimension M divided by the reduced tile size for both the dimensions. Just like RTSN, I can have an RTSM right. So you have the original tile size in the dimensions. You just reduce it.

I mean, you just divided it by the reduced tile size because now you are going to do more activity per thread. So that tells you what is the number of loads that you have to do for each thread. So let us call it LPT i.e. loads per thread.

(Refer Slide Time: 23:18)



GEMM Optimization 4: Rectangular Tiles

```

1  __global__ void GEMM4(const int M, const int N, const int K,
2  const float* A, const float* B, float* C) {
3 // Code for Thread identifiers
4  __shared__ float Asub[TSK][TSN]; __shared__ float Bsub[TSN][TSK+2]; //Padding
5  int numTiles = K/TSK; float acc[WPTN];
6  for (int w=0; w<WPT; w++) acc[w] = 0.0f;
7  for (int t=0; t<numTiles; t++) {
8    for (int l=0; l<LPT; l++) {
9      int tiledIndex = TSK*t + col + l*RTSN;
10     int indexA = tiledIndex*M + TSN*get_group_id(0) + row;
11     int indexB = tiledIndex*N + TSN*get_group_id(1) + row;
12     Asub[col + l*RTSN][row] = A[indexA]; Bsub[row][col + l*RTSN] = B[indexB];
13   }
14   __syncthreads()
15   for (int k=0; k<TSK; k++)
16     for (int w=0; w<WPTN; w++)
17       acc[w] += Asub[k][row] * Bsub[col + w*RTSN][k];
18   __syncthreads()
19 }
20 for (int w=0; w<WPTN; w++) C[(globalCol + w*RTSN)*M + globalRow]
21 }

```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, once we start using all this, we will not try to discuss the entire thing in great detail, but we will just try to identify how things are really going to help us. So, first thing, the first optimization here for rectangular tiles is that well, we have taken B as already transposed matrix. So that would give me some acceleration when I am loading data from the global memory for both A and B. But then there is a problem when I am loading the data.

(Refer Slide Time: 24:08)



GEMM Optimization 2: Coarsening

```

1  __global__ void GEMM2(const int M, const int N, const int K,
2  const float* A, const float* B, float* C) {
3 //Code for thread identifiers
4 //Code for initializing Local memory
5  const int numTiles = K/TS; float acc[WPT]; // WPT-> Work per thread
6  for (int w=0; w<WPT; w++) acc[w] = 0.0f;
7  for (int t=0; t<numTiles; t++) {
8    for (int w=0; w<WPT; w++) { //RTS = TS/WPT : Reduced Tile Size
9      const int tiledRow = TS*t + row; const int tiledCol = TS*t + col;
10     Asub[col + w*RTS][row] = A[(tiledCol + w*RTS)*M + globalRow];
11     Bsub[col + w*RTS][row] = B[(globalCol + w*RTS)*K + tiledRow];
12   }
13   __syncthreads()
14   for (int k=0; k<TS; k++)
15     for (int w=0; w<WPT; w++)
16       acc[w] += Asub[k][row] * Bsub[col + w*RTS][k];
17   __syncthreads()
18 }
19 for (int w=0; w<WPT; w++) C[(globalCol + w*RTS)*M + globalRow]
20 }// Launch Parameters: <<<(M/TS,N/TS),(TS,TS/WPT)>>

```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So let us look at the previous code version which was optimization 2. So, this is how I was doing the load. So I was transferring the data here from B, which in the shared memory representation of B. So, that was all nice when from B, I was loading. But the load was not coalesced. But the store in the shared memory was really happening in consecutive locations.

So when I read the data, it was already in a nice way. It is arranged properly right. But now, when I am loading from B transposed, I will have an opposite effect here. When the data is getting stored in the shared memory, I actually lose the advantage which I gain for the shared load, by doing the load of the transposed matrix. Well, does that mean that I do not want to do a B transpose from B?

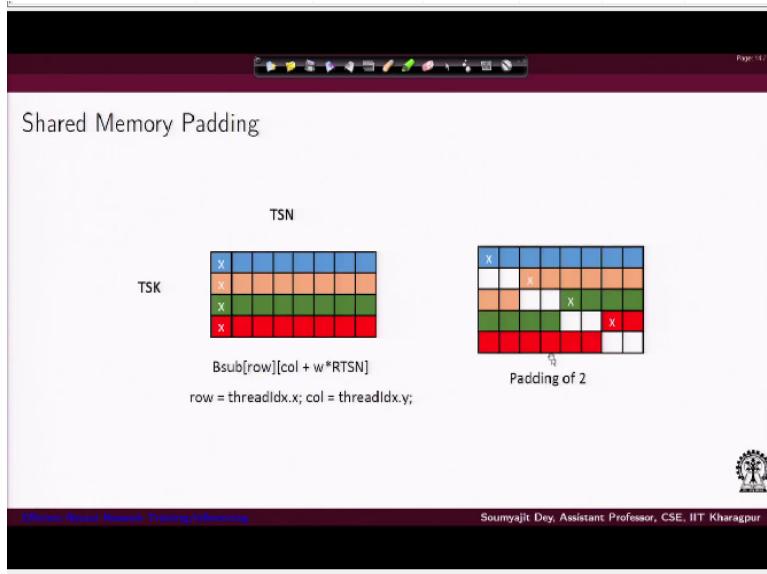
No, I really want to do it. Why? Because global loads are expensive. I really want to minimize the time there. So definitely I will do it. But what I will also do is, when I store here in the shared memory, I will flip the indices. So that is what we do. If you see here, I have row here in the second index. But for rectangular tiles we will actually load in the opposite way. So this was your original coarsen code for optimization 2.

As you can see, I have flipped the indices for Bsub here. So row comes first right. So, that means, I am loading in a nice coalesced manner. But also to preserve the advantage of storage in the shared memory and not losing out on that what I do is a flipping. I hope that is clear. So apart from it, the rest of the things are fine. Only issue is that you may just have to check that we have used a bit of OpenCL semantics here like get group ID.

And all those which of course, is a CUDA code we are trying to see and so that you may just replace it with a suitable CUDA format here fine. So, once that is done, we have got an advantage that now my loads are optimized. And by flipping these parameters here, I have also kept the advantage of having consecutive data in shared memory in a proper way. And then as you can see, you have a normal sync thread.

And then a computation of the data just like previous things, but there will be one problem here, which you have to understand.

(Refer Slide Time: 27:23)



That here again, when you are going to load the data from the shared memory, that is these operations when you are going to load the data from the shared memory like this. This is where you are doing the accumulation. Now, just to remember, what really we did. We did a coalesced load. We enabled coalesced load by doing a transpose, but then to get the advantage of having nice consecutive operations in the shared memory, I have flipped the storage here. But that also creates a problem because after I flip the storage, what I get is that the data gets stored in the shared memory like this. So then that would mean that I will have lot of bank conflicts. When I am now loading the data from the shared memory to the accumulation loop. Again, this is a bit tricky so let me just repeat. I just made nice coalesced loads from the global memory.

But since it is all transposed data, when I am putting it in the shared memory, it is coming for the mathematical matrix's, columns. So that is not the exact order in which you would like to access them. So in order to alleviate that problem with respect to access of the shared memory, you have switched columns here. Now once you have switched columns here, the problem comes with respect to the warps which are going to access this shared memory. Just look at the way the warps are going to access the shared memory.

When you are accessing this shared memory B_{sub} , you are moving by columns. You are operating on columns, because this is just for alleviating your issue with storage. But now when you are going to operate the data from the shared memory, you are accessing first by columns,

and then your warp moves over to the next k. Now, this would give you a problem because you will have lot of bank conflicts.

So, we will do a simple optimization that we have studied earlier, which was just putting in extra locations which are just padding locations. So earlier without padding, that is not having extra locations in your shared memory in your array in the shared memory, your data would have come in like this. But now your data would be coming in like this because there is first data, and then after all the padding's, the next data point is here, next one is here, next one is here.

So, these are the data points which the warp will actually access right. I hope this is clear, but still I will just repeat. So, we have already understood why transpose is necessary. But as you understand the difficulty with transpose will be that when you are in the shared memory you are not having the regularity of the data that you were having earlier because you have optimized the loads. Now your stores have actually disturbed the constitutive nature of the data.

In order to preserve that, you have switched the row and column indices here. But then the problem due to this would be that when consecutive thread ids of the warp will be accessing the shared memory, we would get consecutive data points in the same bank. For alleviating that we use the padding trick that we learned earlier, which is just to add extra locations here.

All you do is you will just add extra locations. Put in more than that is required. And that would actually again offset the locations here. So that when the warps are going to load the data, you avoid the bank conflicts.

(Refer Slide Time: 31:15)

GEMM Optimization 4: Rectangular Tiles

Key changes:

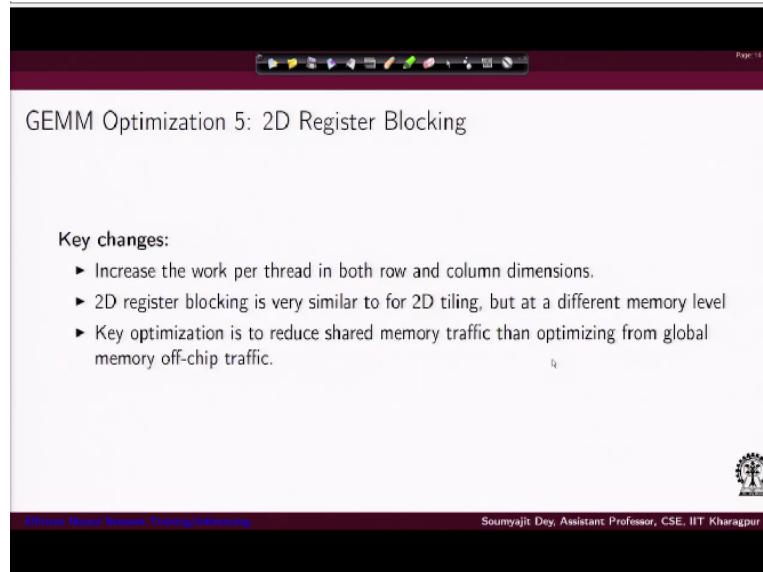
- ▶ Global loads from matrix B (since it is transposed)
- ▶ Shared stores to matrix Bsub (untranspose in local memory)
- ▶ Padding by 2 reduces shared bank conflicts. Note that we pad the memory by 2 rather than 1 to align data to 64-bit (two floats) so that we can benefit from 64-bit loads from local memory.

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, that is the good thing about rectangular tiles. It helps you in optimizing global loads. And so the second important thing is you made the changes to optimize the shared stores. And then you padded the extra locations to avoid bank conflict. Now, why 2 locations? Why not 1 location? That is also an important thing. The reason is in your shared memory for Kepler and other next generation architectures, the banks are 64 bit wide. So that means you have 2 consecutive data points sitting in the same bank anyway. So if your warp is accessing 2 consecutive data points, this is going to have a bank conflict.

So, if you are padding with 2, then you are forcing that this is the memory location as I am showing. So, this is one bank and this is the next bank. In that way you are forcing that 2 consecutive threads in the warp access different banks and there is no conflict. So, this is why padding is done by a factor of 2. Because the warps are going to do 64 bit loads from the local memory. And the loads are 64 bit per, thread since the shared memory is also 64 bit.

(Refer Slide Time: 32:42)



So, with this we will end the current lecture and in the next lecture we will look for some other optimizations. Thank you for your attention.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-62
Efficient Neural Network Training/Inferencing (Contd.)

Hi. Welcome back to the lecture series on GPU architectures and programming. So, if you recall, we have been discussing different possible GEMM optimizations. For example, we started with the basic GEMM. The first optimization was just pushing in our normal tiling concepts i.e. how the idea of tiling helps in matrix multiplication when done with shared memory based tiles.

(Refer Slide Time: 00:51)

The previous kernel can be improved by increasing the amount of work of each thread (thread coarsening). The PTX code for the inner k loop (lines 14-15) for two iterations is as follows:

```
ld.shared.f32 %f50, [%r18+56];
ld.shared.f32 %f51, [%r17+1792];
fma.rn.f32 %f52, %f51, %f50, %f49;
ld.shared.f32 %f53, [%r18+60];
ld.shared.f32 %f54, [%r17+1920];
fma.rn.f32 %f55, %f54, %f53, %f52;
```

It can be observed that only one out of every three instructions is useful!
Increase work per thread in order to reduce number of local memory accesses

IIT Kharagpur

Efficient Neural Network Training/Inferencing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

But then we also saw that leads to a small amount of compute with respect to the loads and that can be further improved.

(Refer Slide Time: 01:00)

```

1 __global__ void GEMM2(const int M, const int N, const int K,
2 const float* A, const float* B, float* C) {
3 //Code for thread identifiers
4 //Code for initializing Local memory
5 const int numTiles = K/TS; float acc[WPT]; // WPT-> Work per thread
6 for (int w=0; w<WPT; w++) acc[w] = 0.0f;
7 for (int t=0; t<numTiles; t++) {
8 for (int w=0; w<WPT; w++) { //RTS = TS/WPT : Reduced Tile Size
9 const int tiledRow = TS*t + row; const int tiledCol = TS*t + col;
10 Asub[col + w*RTS][row] = A[(tiledCol + w*RTS)*M + globalRow];
11 Bsub[col + w*RTS][row] = B[(globalCol + w*RTS)*K + tiledRow];
12 }
13 __syncthreads()
14 for (int k=0; k<TS; k++)
15 for (int w=0; w<WPT; w++)
16 acc[w] += Asub[k][row] * Bsub[col + w*RTS][k];
17 __syncthreads()
18 }
19 for (int w=0; w<WPT; w++) C[(globalCol + w*RTS)*M + globalRow] = acc[w];
20 }// Launch Parameters: <<<(M/TS,N/TS),(TS,TS/WPT)>>>

```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, all that we did was we coarsened the threads.

(Refer Slide Time: 01:05)

```

ld.shared.f32 %f82, [%r101+4];
ld.shared.f32 %f83, [%r102];
fma.rn.f32 %f91, %f83, %f82, %f67;
ld.shared.f32 %f84, [%r101+516];
fma.rn.f32 %f92, %f83, %f84, %f69;
ld.shared.f32 %f85, [%r101+1028];
fma.rn.f32 %f93, %f83, %f85, %f71;
ld.shared.f32 %f86, [%r101+1540];
fma.rn.f32 %f94, %f83, %f86, %f73;
ld.shared.f32 %f87, [%r101+2052];
fma.rn.f32 %f95, %f83, %f87, %f75;
ld.shared.f32 %f88, [%r101+2564];
fma.rn.f32 %f96, %f83, %f88, %f77;
ld.shared.f32 %f89, [%r101+3076];
fma.rn.f32 %f97, %f83, %f89, %f79;
ld.shared.f32 %f90, [%r101+3588];
fma.rn.f32 %f98, %f83, %f90, %f81;

```

For 8 iterations of the inner k loop, there are 8+1 loads from the local memory for 8 FMA's (instead of 8+8).

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And in that way, we could actually reduce the ratio of load with respect to the computes. And to say in other words, we actually were able to increase the amount of compute per load, by doing the coarsening optimization.

(Refer Slide Time: 01:22)

The slide has a dark header bar with icons. The main title is 'GEMM Optimization 3: Wider loads'. Below the title is a bulleted list of points:

- In the previous implementation we increased the amount of work in the column-dimension of C.
- The same optimization trick can be done for the row-dimension
- The additional advantage for optimizing across the row dimension is using wider data-types.
- Increasing the work per thread (WPT) in the row-dimension of C can be done by considering vector data-types instead of loops over WPT.
- NVIDIA GPUs do not support vector operations (such as multiply or add) in hardware but possess special wider load and store instructions both for the off-chip and the local memory.

At the bottom, there is a footer bar with the text 'Efficient Neural Network Training/Inferencing' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

And then we exploited the wider loading GEMMs that are available in NVIDIA GPUs because they do not support vector operations like multiply and add, which is vectorized. But they actually provide you with wider load and store instructions.

(Refer Slide Time: 01:36)

The slide has a dark header bar with icons. The main title is 'GEMM Optimization 3: Wider Data Types'. Below the title is a block of CUDA code:

```
1 __global__ void GEMM3(const int M, const int N, const int K,
2 const float8* A, const float8* B, float8* C) {
3 //Code for thread identifiers
4 // Modify shared memory initialization
5 __shared__ float8 Asub[TS][TS/WIDTH];
6 __shared__ float8 Bsub[TS][TS/WIDTH];
7
8 float8 acc = { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f };
9 const int numTiles = K/TS;
10 for (int tile=0; tile<numTiles; tile++) {
11
12     const int tiledRow = (TS/WIDTH)*tile + row;
13     const int tiledCol = TS*tile + col;
14     Asub[col][row] = A[tiledCol*(M/WIDTH) + globalRow];
15     Bsub[col][row] = B[globalCol*(K/WIDTH) + tiledRow];
16     __syncthreads();
```

At the bottom, there is a footer bar with the text 'Efficient Neural Network Training/Inferencing' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'.

So we made good use of this float8 kind of data types here for using the wider load instructions for filling in the tiles. That is how we will put it.

(Refer Slide Time: 01:51)

```

19     for (int k=0; k<TS/WIDTH; k++){
20         vecB = Bsub[col][k];
21         for(int w=0; w<WIDTH; w++) {
22             vecA = Asub[WIDTH*k + w][row];
23             switch (w) {
24                 case 0: valB = vecB.s0; break; case 1: valB = vecB.s1; break;
25                 case 2: valB = vecB.s2; break; case 3: valB = vecB.s3; break;
26                 case 4: valB = vecB.s4; break; case 5: valB = vecB.s5; break;
27                 case 6: valB = vecB.s6; break; case 7: valB = vecB.s7; break;
28             }
29             acc.s0 += vecA.s0 * valB; acc.s1 += vecA.s1 * valB;
30             acc.s2 += vecA.s2 * valB; acc.s3 += vecA.s3 * valB;
31             acc.s4 += vecA.s4 * valB; acc.s5 += vecA.s5 * valB;
32             acc.s6 += vecA.s6 * valB; acc.s7 += vecA.s7 * valB;
33         }
34     }
35     __syncthreads()
36 }
37 C[globalCol*(M/WIDTH) + globalRow] = acc;
38 } // Launch parameters: <<<(M/TS, N/TS),(TS/WIDTH, TS)>>>

```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant

But that also required us to go through this complex switching of values, which we have already explained in the last lecture , for supporting the wider data types and all that.

(Refer Slide Time: 02:03)

- ▶ The Tesla K40 GPU which has 48KB of shared memory per SM, on which multiple thread blocks can execute.
- ▶ For a 32×32 tile, we consume $2 * 32 * 32 * 4 = 8KB$ per work-group, so there is some headroom left.
- ▶ Since both matrix A and B share the dimension K, we try to create rectangular tiles.
- ▶ We can also pre-transpose the matrix B using the optimized transpose kernel used before.

```

#define TSM 64           // The tile-size in dimension M
#define TSN 64           // The tile-size in dimension N
#define TSK 32           // The tile-size in dimension K
#define WPTN 8            // The work-per-thread in dimension N
#define RTSN (TSN/WPTN)   // The reduced tile-size in dimension N
#define LPT ((TSK*TSM)/RTSN) // The loads-per-thread for a tile

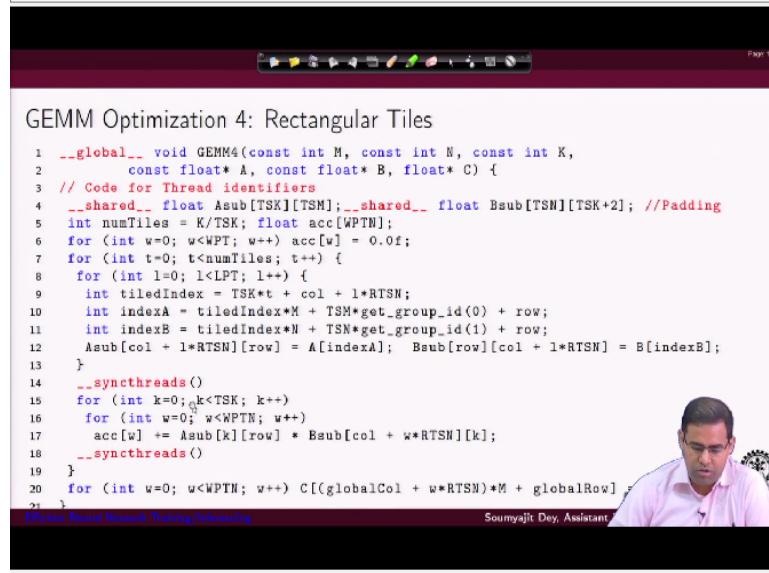
```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant

But the next thing that comes is something that we called rectangular tiles. So we will just do a brief recap on why that is required. So in the rectangular tiles part, if you remember that K40 GPUs have got 48KB of shared memory. Whereas for these tiles that we have defined, they were 32×32 tiles, 2 of them, each storing 4 bytes of data. So it was overall 8KB. So that is the basic reason that why you would want to have a rectangular tile.

Now well, what do we do with that? So let us just browse through this idea of rectangular tiles once again. So the idea was pointing to the fact that since we have a large amount of shared memory, we can increase the tile size and that is what we did. So we increase the tile size from 32×32 to tiles of size 64.

(Refer Slide Time: 03:03)



GEMM Optimization 4: Rectangular Tiles

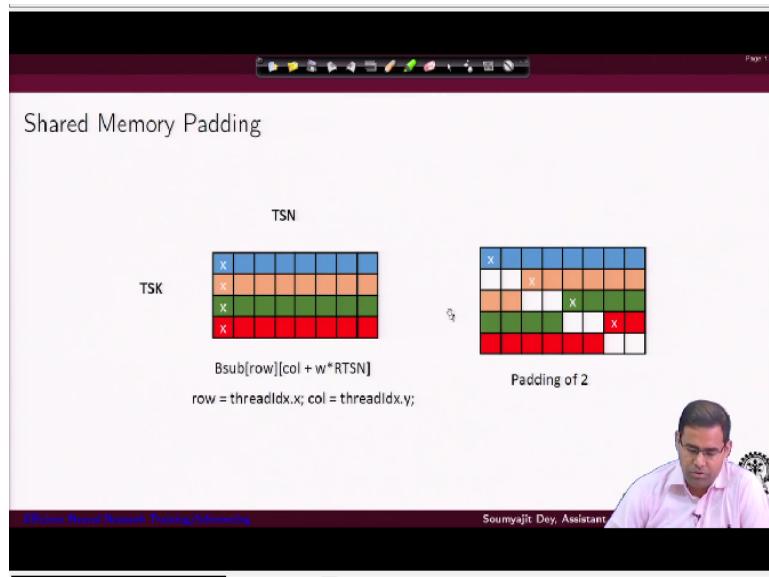
```

1  __global__ void GEMM4(const int M, const int N, const int K,
2                        const float* A, const float* B, float* C) {
3  // Code for Thread identifiers
4  __shared__ float Asub[TSK][TSM]; __shared__ float Bsub[TSM][TSK+2]; //Padding
5  int numTiles = TSK/WPTN;
6  for (int v=0; v<WPTN; v++) acc[v] = 0.0f;
7  for (int t=0; t<numTiles; t++) {
8    for (int l=0; l<LPT; l++) {
9      int tiledIndex = TSK*t + col + l*RTSN;
10     int indexA = tiledIndex*N + TSM*get_group_id(0) + row;
11     int indexB = tiledIndex*N + TSN*get_group_id(1) + row;
12     Asub[col + l*RTSN][row] = A[indexA]; Bsub[row][col + l*RTSN] = B[indexB];
13   }
14   __syncthreads()
15   for (int k=0; k<TSK; k++)
16     for (int w=0; w<WPTN; w++)
17       acc[w] += Asub[k][row] * Bsub[col + w*RTSN][k];
18   __syncthreads()
19 }
20   for (int u=0; u<WPTN; u++) C[(globalCol + u*RTSN)*M + globalRow] = acc[u];
21 }
```

Soumyajit Dey, Assistant Professor

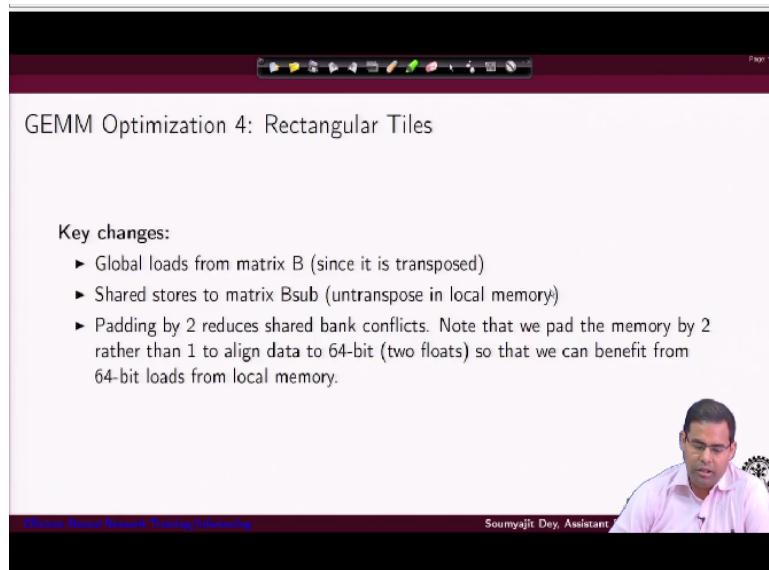
So that is what we discussed. We will increase the rectangular tile size. And , there was also this optimization that since we are supporting bigger tiles, why not store the matrix B in a transpose way. Because then you have the advantage of memory access coalescing while loading from the global memory. But then we also figured out that leads to a problem with the shared stores. For that we needed to flip the indices here, so that the shared memory stores are done in a nice way. So that they can be fetched again properly.

(Refer Slide Time: 03:50)



But that also led to a problem that when I am now looking for loading from the shared memory, they are all falling in the same bank. So, in order to reduce this bank conflict, we added the data here. So, that is a short summary of how rectangular tiles really helped us.

(Refer Slide Time: 04:00)



So these are the changes. You do global loads from B where B is pre-transposed. And then you do shared store through the matrix Bsub in such a way that the load is optimized. And the way you optimize the load is you paired the data types here with 2 elements, because in each bank you have memory width of 64 bits for the loads. So you paired with 2 positions here. And that gives you a huge advantage over the baseline implementation.

(Refer Slide Time: 04:38)

GEMM Optimization 5: 2D Register Blocking

Key changes:

- ▶ Increase the work per thread in both row and column dimensions.
- ▶ 2D register blocking is very similar to 2D tiling, but at a different memory level
- ▶ Key optimization is to reduce shared memory traffic than optimizing from global memory off-chip traffic.

Soumyajit Dey, Assistant Professor, IIT Kharagpur

Now, let us just start with the next optimization, which is what we call as 2D register blocking. So, just to remember that in case of rectangular tiles, we had to do the transpose and then store properly in the shared memory, while doing the transformation from the local memory. And then we had to pair the data here. So, these were the other 3 things we needed to do simultaneously.

Now again coming to the register blocking part, earlier when discussing the optimizations where we are doing the coarsening, I would say in 1 dimension because if you remember while we were doing thread coarsening, we are computing for multiple data points in 1 dimension. When we were supporting wider loads again we were doing a similar coarsening in the other dimension.

So, the general idea could be that you increase the work per thread in both the row and column dimensions. So, this is what we call as 2D register blocking. So, essentially we are doing the register loads in such a way that it is essentially similar to 2D tiling that we discussed for the shared memory based optimization for matrix multiplication. But this is done at a different level. Let us just replicate the idea that you are doing a shared 2D load of data from the global memory to the shared memory. Let us just replicate the idea and load data in a similar way from the shared memory to the global memory right. So that is what we will be calling as 2D register blocking. So it is similar to 2D tiling, but it is done at a different memory level.

In the earlier case, it was from the global memory to the shared memory. In this case is from the shared memory to the registers..

So, what really the optimization helps in is that it helps to reduce the shared memory traffic. Just like using the earlier optimization of normal standard tiling, what we are really doing where we are really optimizing the global memory off chip traffic? So in this case, you are using the same idea between shared memory and the register file to reduce the shared memory traffic.

(Refer Slide Time: 07:02)

GEMM Optimization 5: 2D Register Blocking

```

1 #define TSM 128           // The tile-size in dimension M
2 #define TSN 128           // The tile-size in dimension N
3 #define TSK 16            // The tile-size in dimension K
4 #define WPTM 8            // The work-per-thread in dimension M
5 #define WPTN 8            // The work-per-thread in dimension N
6 #define RTSM (TSM/WPTM)   // The reduced tile-size in dimension M
7 #define RTSN (TSN/WPTN)   // The reduced tile-size in dimension N
8 #define LPTA ((TSK*TSM)/(RTSM*RTSN)) // Loads-per-thread for A
9 #define LPTR ((TSK*TSN)/(RTSM*RTSN)) // Loads-per-thread for B
10 //Since TSM and TSN are considered to be equal, load-per-thread for A (LPTA)
    is equal to loads per thread for B (LPTR)
11
12 dim3 blocks(M/TSM, N/TSN);
13 dim3 threads(TSM/WPTM, TSN/WPTN);

```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor

So let us just have a look at how it works. So we will just use our usual definitions, the tile size in dimension M. Remember that we are multiplying $M \times K$ and $K \times M$ matrices, tile size TSM in dimension M, tile size TSN in dimension N and tile size TSK in dimension K. This is what we had set. And then if you remember earlier, we are defining work per thread while doing coarsening. We will just use the same idea in terms of work per thread in dimension M and work per thread in dimension N.

So, if we are trying to coarsen the work in both of these dimensions, then we will have a reduced tile size in both the dimensions. So, these are tiles which will be based on the shared memory for loading data to the registers. So then the reduced tile size in dimension M would be just like the tile size in dimension M divided by the amount of work we are now giving to a single thread in dimension M.

So it is just tile size in dimension M divided by the work per thread that we are now trying to define in the same dimension. So that gives me RTSM. Similarly, I can have RTSN. So that is essentially nothing but TSN by the work per thread that I have defined in that dimension N. So, once this is done, then I can define what is the amount of data load that each thread has to do for A and B matrices.

In this simplistic case, we will consider 2 of them to be same here. So LPTA is nothing but the overall tile size for the $M \times K$ matrix divided by the reduced tile size in the dimension M multiplied by the reduce tile size in the dimension N. So that gives you the loads per thread for A. Similarly what happens to loads per thread for B? Well you consider the original tile size in dimension K and tile size in dimension N for the matrix B and you divide it by the reduced tile size in the respective dimensions. So, essentially you are dividing the original tile size by the reduced tile size and that is giving you the amount of loads that a thread has to do for A and similarly, the amount of load a thread has to do for B right. So, since this TSM and TSN are considered to be equal in our case, the loads per thread for the matrix A i.e. LPTA and the loads per thread for matrix B i.e. LPTB are going to be same. So, with this we can then go and define the blocks and the grid. So, they can be just like this. I am dividing M by TSM, N by TSN. That would be my dim3 definition of blocks and similarly for threads.

(Refer Slide Time: 09:59)

```

1 // Use 2D register blocking (further increase in work per thread)
2 __global__ void myGEMM6(const int M, const int N, const int K, const float* A,
3 const float* B, float* C) {
4
5 // Thread identifiers
6 const int tidm = threadIdx.x; // Local row ID (max: TSM/WPTM)
7 const int tids = threadIdx.y; // Local col ID (max: TSN/WPTN)
8 const int offsetM = TSM*blockIdx.x; // Work-group offset
9 const int offsetN = TSN*blockIdx.y; // Work-group offset
10 // Local memory to fit a tile of A and B
11 __shared__ float Asub[TSK][TSM];
12 __shared__ float Bsub[TSN][TSK+2];
13 // Allocate register space
14 float Areg;
15 float Breg[WPTN];
16 float Acc[WPTM][WPTN];
17 // Initialise the accumulation registers
18 for (int wm=0; wm<WPTM; wm++)
19     for (int wn=0; wn<WPTN; wn++)
20         acc[wm][wn] = 0.0f;
21

```

GEMM Optimization 5: 2D Register Blocking

Soumyajit Dey, Assistant Professor, Dept. of EEE, Jadavpur University

So, what will be my basic steps when I am going to execute this 2D register blocking. So, of course, one thing is to be sure that I am increasing the work per thread further by using this optimization. For identifying the threads we use, each thread will initiate the following local variables. So let us say I define this tidm and tidn just for noting down the local thread ids with respect to the columns and the rows.

And of course, the maximum value each of them will be this TSM divided by WPTM and similarly TSN divided by the WPTN like we had defined earlier. And similarly we can compute the offsets which will be used soon i.e. exactly from which block, the thread shall start working. That is figured out by multiplying this tile size in the dimension M with the block ID for the thread.

And similarly, the offset in the dimension N can be found by the tile size in dimension N multiplied by the block id in the Y dimension. So, the next thing that comes is that we have to define the memory to fit a tile for A as well as a tile for B, right. So, how do you really do it? Well, we already have these TSKs and TSMs defined. So that gives me Asub as TSK x TSM, and Bsub as TSN x TSK + 2. If you remember our earlier idea of padding we had introduced, that is actually getting carried over here. So the next thing you have to do is initialize a 2D accumulation register. Earlier, it was a 1D array, but now since it is 2D register blocking, you are initializing these acc in a 2D array with all 0s for the float values, right.

(Refer Slide Time: 12:02)

GEMM Optimization 5: 2D Register Blocking

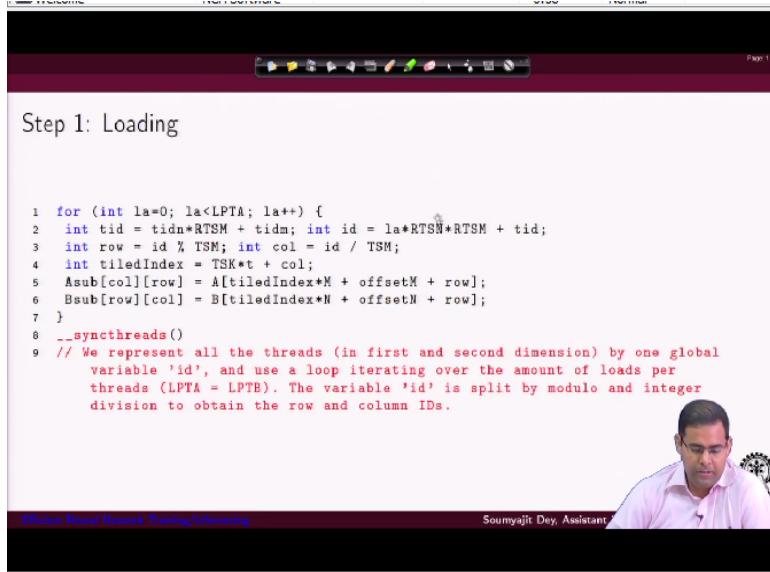
```
1 // Loop over all tiles
2 int numTiles = K/TSK;
3 for (int t=0; t<numTiles; t++) {
4 // Step1 : Load one tile of A and B into shared memory
5 // Step2: Loop over the values of a single tile and perform the computation
6 }
7 // Step3: Store the final results in C
```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant

But then comes the question. What is the overall operation that you have to do for per thread for every tile? So, this is your number of tiles - K divided by the TSK i.e. your tile size in the K dimension. So, that is the total number of tiles you have in that K dimension over which you have to hop right. So, this is the tile loading loop, i.e. the outer loop of the multiplications.

So, this gives you the number of tiles and inside this you have the same old steps to be done for this newer setting. You have to load 1 tile of A and 1 tile of B into the shared memory and then you have to loop over the values of a single tile and perform the computation just like earlier. Only thing is that you have more work per thread right now, and at the end of the entire computation when all these load and compute for all the tiles is done you just write back the values in the result matrix C.

(Refer Slide Time: 12:58)



So here comes the load part. So how do you really load? Of course, one may start thinking that I have got this 2D data to load. But we will do it in one single loop here. The way we are doing it is, as you can see, for each position, I am just iterating from 0 to LPTA, right, the load per thread value. So this is the amount of data I am supposed to load and since LPTA is equal to LPTB, just by iterating, over this 2, I can do the loading for the both the Asub and Bsub matrices. And from where am I supposed to load? Well, we are representing all the threads in the first and second dimension by one global variable ID. So let us understand, what is the problem here? So we have already defined the work per thread. But now, I mean, ideally you want to start thinking that this would be a 2D loading.

So why do not I have a cascade of 2 loops for doing the loading? Instead of that what we are doing is that we also know how many loads one thread is supposed to do. So let us just iterate one loop from 0 to that maximum number of loads that one thread is supposed to do, and figure out a global position for each thread. You figure out a global position in the array from where you are supposed to do the load.

So essentially we are trying to figure out by computing an id. And then, that position, I can just figure out because I know the M, and what is the offset inside it. And then I can go to the corresponding row. So by computing these values, I can just figure out from where to load. And

then the other thing I will do is that, I will figure out the id of the thread by using this local computation. You can just easily check how this ID is getting computed.

And if you just do a percentile and a divide operation, you can just find out what is row and column index in Asub and Bsub where you are supposed to put the value. Mind that your Asub and Bsub have opposite modes of access, because for B you are having the transposed matrix. So just like in the previous case here, we would say so, these are always the same as rows.

So, in using this loop, you are just identifying here through this access expression from which location in the matrix A you are doing the load from and which location in the matrix B you are doing the load. And by doing this computation of id with this formula, you are just figuring out which number to load and then you are multiplying it by this reduced tile size to go to that corresponding location. And then you are just doing an offset with the tid which you are already computed here. So with this, you are able to go to the locations of both A and B matrices. So, again, we will just repeat that here, we computed the tile index. And then you are just multiplying the tile index with M and N, because these are the corresponding dimensions to look for in the A and B matrix right. Mind that B is transposed here.

So, these are the dimensions to look for in the A and B matrix. And then from A and B, you are doing our load to Asub and Bsub just like we have discussed earlier. So, the variable ID that we which we have computed here, you are just then doing these 2 operations to figure out what is the row and column value where to load.

(Refer Slide Time: 16:52)

Step 2: Performing the computation

```

1 // Loop over the values of a single tile
2 for (int k=0; k<TSK; k++) {
3     // Cache the values of Bsub in registers
4     for (int wn=0; wn<WPTM; wn++) {
5         int col = tidn + wn*RTSN;
6         Breg[wn] = Bsub[col][k];
7     }
8
9    // Perform the computation
10   for (int wm=0; wm<WPTM; wm++) {
11       int row = tidm + wm*RTSM;
12       Areg = Asub[k][row]; // Cache a single value of Asub into a register
13       for (int vn=0; vn<WPTN; vn++)
14           acc[wm][vn] += Areg * Breg[wn];
15   }
16 }
17
18 // Synchronise before loading the next tile
19 __syncthreads();

```

Once the loading is done, you are going to run this computational loop where you are actually going to perform the multiplications and for that, you do this. This is your outer loop, then inside the outer loop, first notice that the outer loop is definitely going to run for K values right up to TSK. That is the tile size in K. And then, because again, the next time we load and again, we are going to run these for TSK. And that is just like how you do computation with tiles as we all know. So then, the thing that you do is you cache the values of Bsub in the registers. Now that is an important step. So observe what is going on. So, you have got this value of Bsub. Now you are going to load them in this array Breg. Now, why is this important? Observe that Bsub is shared. So it is in the shared memory, but where is Breg?

Breg is a local array, which means this will be located in the register. So you are now going to cache this value of Bsub. So, you are just computing these column index for the same K and getting the value from Bsub and you are loading it into Breg right. So, in that way, with this loop, you are loading all the values of Bsub that you require for multiplication in Breg.

Well, what is the next step? So, this actually caches all the required values for Bsub and then you get to the computation. So, now, for the real multiplication computation, you have this for loop where you are running it for this $wm = 0$ to WPTM, right. And here, what you are going to do is well first you compute the row index from using this tidm and these wm values and that gives you the exact value of Asub, which is going to be multiplied.

Well, so, as you can see, you are essentially caching a single value of Asub into the register and that is the value that is going to be multiplied with all the values of Breg continuously and stored in the accumulator for the corresponding value. So, we are not repeating this because as you can understand this follows the similar pattern like our earlier optimizations. You need all the consecutive B values, but you need this single A value.

That is why you have already cached these B values. So, as you can see, this is all the work for one thread. So, the real multiplication by the thread is really happening here. And since you have 2 loops, so, in that way, you are getting a coarsened set of values done, right. So, if we just repeat inside C, you have got these tiles defined for 1 tile. Well, these are the reduced tile sizes. I am drawing for one thread. You are now making that thread do a 2D computation.

So, this loop is actually iterated over the number of computations that is to be done. Work per thread in M is in one direction of course. You have this A and B. So, this outer loop is going over the work per thread in direction M and the inner loop is going over the work per thread direction N but for a single position of this direction M, you load 1 Areg value right and for that you are going to use these sequence of Breg values.

So, in every outer iteration what you are doing is that you are caching a single value and then in the inner iteration, you are going to use this Areg value with this array of Breg values. So, what is important to notice is that every thread has got per thread activity. For every thread for each tile, you have got some per thread activity. So if you see this is per thread activity for a thread any one time.

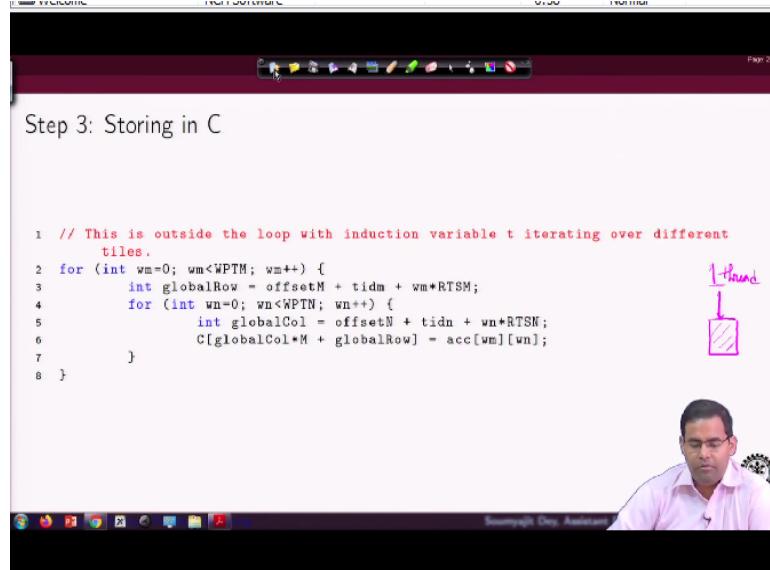
So for that single tile, what you are really doing is that you are first figuring out for this tile, what are the values I am supposed to load. So, this is where the threads' loading of values from the shared memory to the registers are getting done. So, that is why we have cached the values of Bsub in registers. So, here, inside this compute loop, there is some amount of memory activity going on.

The memory activity to be more specific is for loading values from the shared memory to the registers. And this is what we would say is the crux of the optimization. Like earlier, we have already seen this optimization which is like loading from global memory to shared. And next, the current optimization is that you load from shared to a 2D register. So, that is what you do. As you can see inside this loop for one value of K, you are first making a sequence of loads from Bsub.

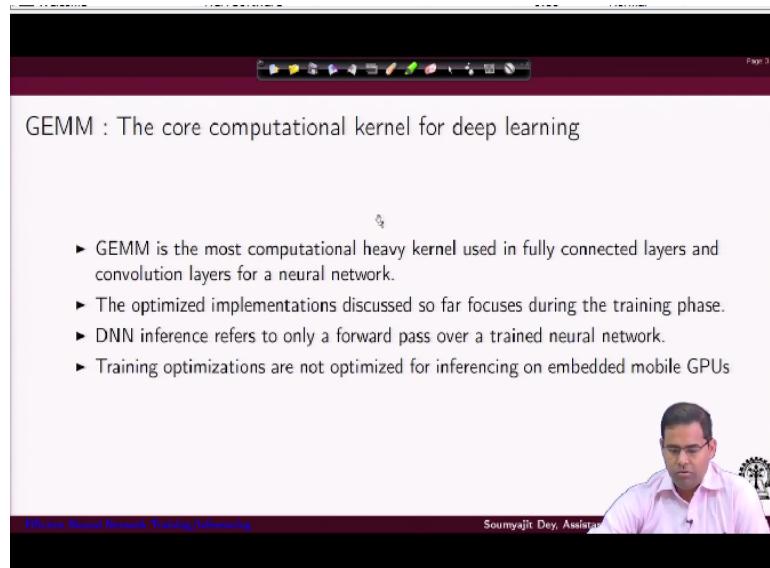
So, you are making a sequence of loads from Bsub to here, and then that is the activity for this thread. And similarly, for other threads also, they are all making their own corresponding caching of values of Bsub for this loop. And for this thread, after it has got these values cached, it goes to perform the real computation. So what it does is inside this for loop, first it performs the computation of the row index. Then it uses the row index to compute what value is to be loaded into Areg for the corresponding Asub shared memory. For that, it has got this index i.e. which row and K is getting computed. And then here, it is keeping it constant, and multiplying it continuously with these Breg values. So it is multiplying this Areg continuously with Breg values and it is iterated here over this.

Well once this is done, again it will load from the outer loop, it will again, load another corresponding value of from Asub to Areg. And then again in the inner loop, it will multiply this Areg value with a sequence of values from Breg and that is how it will work.

(Refer Slide Time: 25:22)



(Refer Slide Time: 26:31)



So this is our idea of how these different possible optimizations can be performed for doing the GEMM computation. So just to recall, all we do here is that we are using our earlier optimizations together, but doing it in a 2 dimensional way, by coarsening the threads in both dimensions. Well here we have not used the wider load idea. That is also something you need to understand. But the important thing is for you to figure this out , i.e. how the computes work as I am giving a 2D amount of computation to be done for each thread. That is why now I have a cascade of loops here for doing all those computations and also the loading of the data. And as I recall that there are 2 levels of loading. Here you have a load going on from the global to the shared memory. And then in your earlier compute loop, you now have an amount of load first done from the shared to the registers.

And these 2 steps we are now kind of differentiating here. And then I am using the cached values to multiply i.e. I am caching a single value of Areg and multiplying it with a sequence of values of Breg, and so on so forth through this loop cascade. And then again, I am storing it back using another loop cascade. So right now I have more work per thread. And these are good optimizations in the sense that in modern GPUs you have sufficiently big shared memories and if you can just choose a proper tile size and a suitable thread coarsening factor in the 2 dimensions then this can really help. So the summary here would be that GEMM is the most computationally heavy kernel which is used for fully connected layers, and also the convolution layers of neural network and the optimized implementations that we have discussed so far. They focus on the training phase of course, and if we are speaking about the other phase of neural network work like inferencing, that refers only to a forward pass over a trained neural network. And so in general, these optimizations we have discussed will likely be more important for an HPC kind of process where you are primarily going to do large neural network training in significantly powerful computers, whereas inferencing is the more popular workload for embedded GPUs.

Because in an embedded GPU, you can have an inferencing engine for doing a lot of stuff, like recognizing somebody's image from a camera, or doing some object detection. Or maybe live streaming of video, doing a face detection and all that are inferencing kind of workloads. So they also require different kinds of optimizations, which you can figure out.

(Refer Slide Time: 29:41)

DNN Pruning and Sparse Matrix Operations

- ▶ Several works have been proposed over the years that focus on pruning the weights of a neural network.
- ▶ This essentially implies that the weight matrices are now sparse in nature.
- ▶ Implementations for sparse matrix operations would be beneficial in this context.

Efficient Neural Network Training/Inference
Soumyajit Dey, Assistant

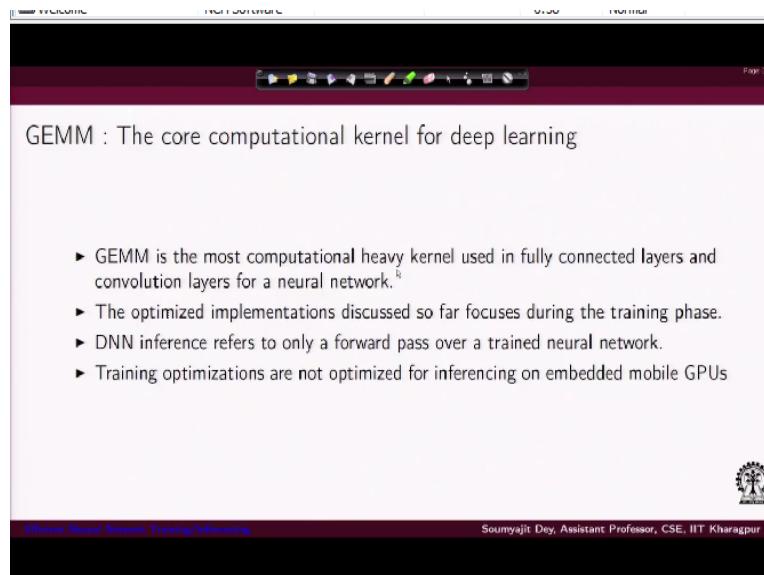
So, with this maybe we will like to close this current lecture. Thank you.

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-63
Efficient Neural Network Training/Inferencing (Contd.)

Welcome back to the lecture series on GPU architectures and programming. So, if you remember in the last lecture, we have covered these different this GEMM optimizations i.e the different ways in which the GEMM kernel can be optimized.

(Refer Slide Time: 00:37)



GEMM : The core computational kernel for deep learning

- ▶ GEMM is the most computational heavy kernel used in fully connected layers and convolution layers for a neural network.
- ▶ The optimized implementations discussed so far focuses during the training phase.
- ▶ DNN inference refers to only a forward pass over a trained neural network.
- ▶ Training optimizations are not optimized for inferencing on embedded mobile GPUs

Efficient Neural Network Training/Inferencing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

(Refer Slide Time: 00:39)

DNN Pruning and Sparse Matrix Operations

► Several works have been proposed over the years that focus on pruning the weights of a neural network.
► This essentially implies that the weight matrices are now sparse in nature.
► Implementations for sparse matrix operations would be beneficial in this context.

Q & A

Q: What are the benefits of using sparse matrix operations in DNNs?

A1: Reduced memory footprint and faster computation times.

A2: Simplified training and inference processes.

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So in the current lecture, we will like to focus on other ideas like how these sparse matrices are operated on. Now why is that an important thing? Because people have often found that for these neural networks pruning is actually useful. Because if a DNN can be pruned and the resulting matrices can be sparse, then that can actually lead to many nice sparse matrix operations, which are often found useful.

So several works have been proposed over these years that focus on pruning the weights of a neural network. Because, if you prune the weights, then that leads you to a lighter representation of the network. And there are many nice sparse matrix algorithms available on how to make good use of such algorithms. So, if we consider such modern pruned DNN kinds of networks , where the weights have been pruned, we have a neural network, which is represented by a sparse matrix. Such algorithms then become very relevant and some of them are also very useful for GPU style implementations. So in that way implementations for sparse matrix operations are very beneficial in this context.

(Refer Slide Time: 02:09)

Sparse Matrix Vector Multiplication: SpMV

- ▶ There exists different formats for storing sparse matrices.
 - ▶ Diagonal format
 - ▶ ELLPACK
 - ▶ Coordinate Format (COO)
 - ▶ Compressed Sparse Row Format (CSR)

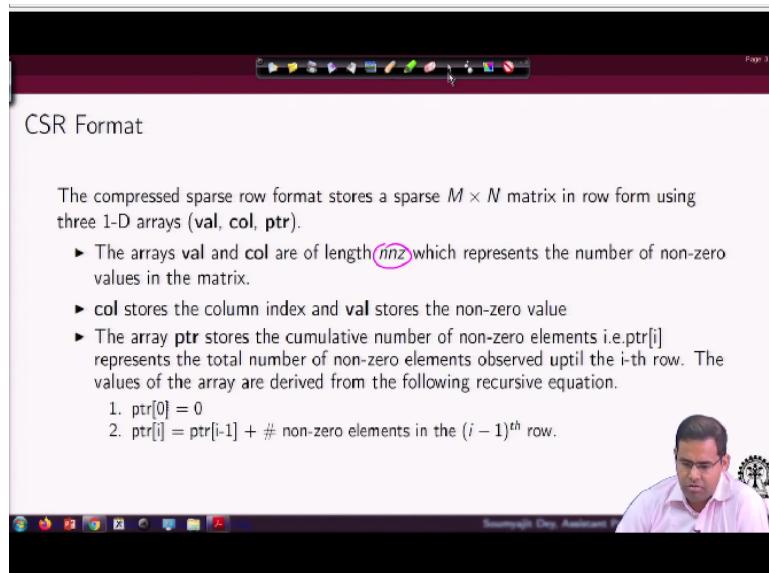
Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor

And the particular algorithm we will be talking about here is sparse matrix vector multiplication. So, you have a sparse matrix and you are multiplying it with a vector. Now, that is also an important operation that you will have to do inside the neural network right. In general, this sparse matrix vector multiplication in short form is known as is SpMV. Now, of course, we have to understand that the very reason we want to exploit the sparsity of a matrix is the absence of non trivial elements.

The significant absence of non trivial elements in a matrix occurs in general. If you are storing a matrix in the memory, you are actually consuming $O(n^2)$ amount of space. But if you are considering a sparse matrix where most of the entries are, in general, trivial or 0 , then we would like to use an alternate storage format where I do not encourage this kind of $O(n^2)$ amount of storage requirement.

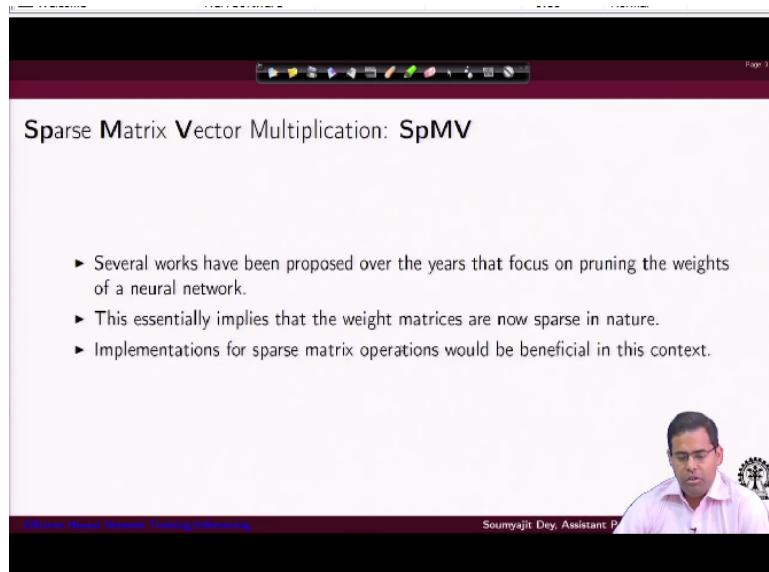
So specifically for sparse matrices, there are several such well known formats, like diagonal format, ELLPACK format, coordinate formats COO, and also compressed sparse row format, CSR.

(Refer Slide Time: 03:32)



So what we will do, in this lecture, is that we will talk specifically for the CSR format, how multiplication operation on matrices are performed using the CSR format and how that can be optimized considering a CUDA based parallel implementation. So this is what we will focus on. So I will just recap.

(Refer Slide Time: 03:57)



Considering large neural networks which we are going to really face, people perform this kind of weight pruning. Now, weight pruning will lead to a sparse matrix and that is the reason you want to do weight pruning. Because with a sparse matrix you can have nice algorithms to work with and nice storage methods. So one of them is the CSR storage method, which we like to introduce here.

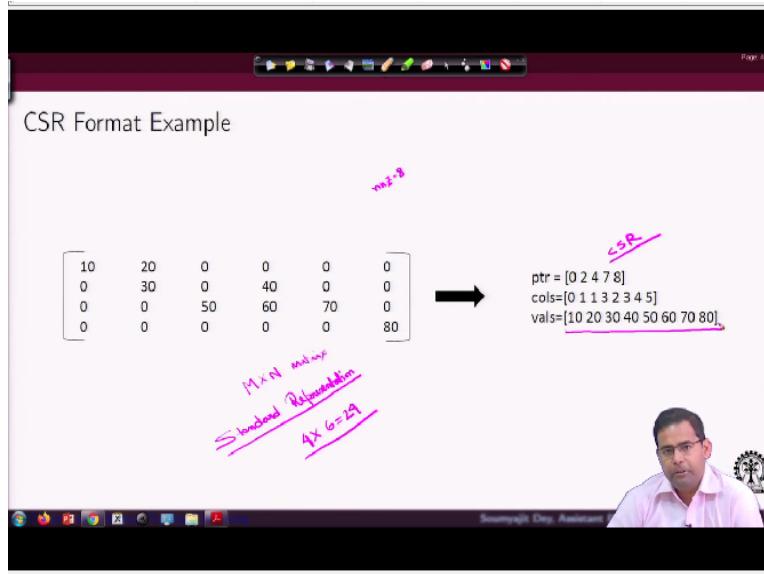
And the next thing that we would like to use for our purpose is optimizing the matrix multiplication further. So the compressed sparse row format, stores us parts across an matrix in a row form where it uses three 1-D area. So let us call the arrays as val, col, and ptr. Now the arrays are will refer to them as value, column and ptr. The arrays val and col are of length nnz, which represents the number of nonzero values in the matrix.

So let us say inside $M \times N$ matrix , I have got only nnz number of non-zero values and everything else is set to zero right. So, essentially we are just interested in storing these nnz number of entries. And, we are interested in figuring out what are the locations in which these non-zero values are stored, because we can safely say that all other locations are just having zeros. Now, the col or column stores the column index and the val stores the non-zero value.

So, we have one 1-D array, which is storing all these nnz number of non-zero values, we have another 1-D array called col, which is storing the column indices for each of these nonzero values i.e. this nonzero value is actually positioned in this column of the sparse matrix. And then, I have the other array called ptr, which is storing the cumulative number of nonzero elements. So essentially $\text{ptr}[i]$ will represent the total number of nonzero elements observed up till the ith row.

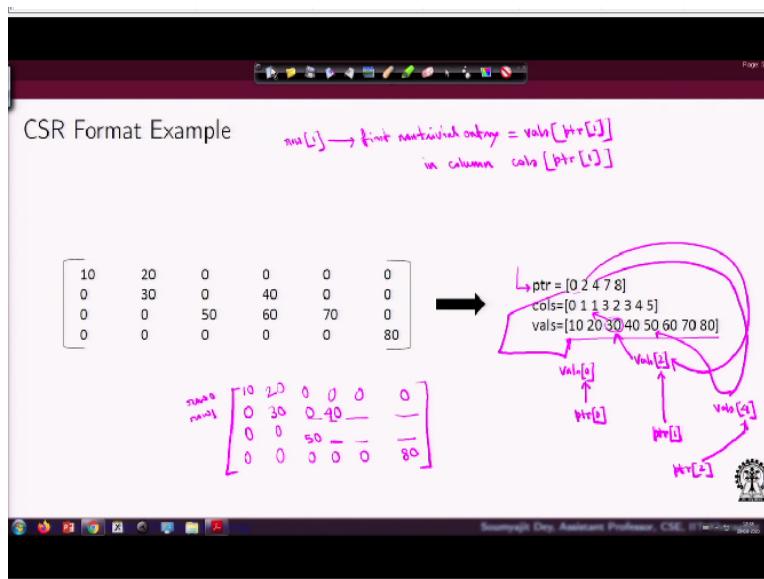
We will see what it means. The value of the array derived from the recursive equation will let us get into that. Rather, maybe we will have a better idea by looking into an example.

(Refer Slide Time: 06:20)



So let us take a simple example. So on the left side, we have our CSR format. We have a normal $M \times N$ matrix. So in this case, as you can see the amount of storage I require is 4×6 . So that is 24 and here on the right hand side, I am having the storage in CSR format. And we are just using three 1-D arrays. So first of all, what are the number of values? 1 2 3 4 5 6 7 8, right. So, these are all stored in the val array.

(Refer Slide Time: 07:43)



So, let us try to figure out how this thing can be constructed back. So, look at this ptr array. This essentially is containing information related to the rows. Now, I am interested in constructing the 0th row. So what we will do is that we will see that the entry for the 0th row here is 0. Now this 0 means well you start looking for the entry in val 0.

This means you start looking for the entry in $\text{val}[2]$ and so on so forth. Then 4 means you start looking for the entry in $\text{val}[4]$ like this. So if we just now try and construct this thing, for the 0th row, we get that $\text{vals}[0]$ and for ptrs 0th position I have 0. So let us see what is this 0? This is nothing but the content in ptrs 0th position, right. This is the content in ptrs first position. This is the content in ptrs second position. So essentially from this, for each row, I am looking into ptr . And I am getting the corresponding index in val . So this is essentially telling me what is the location of the first non trivial element in that row? So let us take an example for the first row. I get an index 2. Now, I look into it. So I follow this index 2, and then look into $\text{val}[2]$. So essentially, what I am doing is for the first row I am looking into $\text{ptr}[1]$, and then I am looking into vals of $\text{ptr}[1]$.

That is what I am doing. So if I just write it like this. First non trivial entry of row 1 equal to $\text{vals}[\text{ptr}[1]]$ i.e. row 1's first non trivial entry is located in column $\text{cols}[\text{ptr}[1]]$. So for 1, you go to $\text{ptr}[1]$. So, that is true, right. So, you know just look into the location for it. That is how it is getting done So it is here in this the column i.e. 30. So, I hope that is clear. So that tells me that for row 1. Just remember this.

So ptr 's, position 1 gives me some information about some element in row 1. So, row 1 is fine. Now then in row 1, what do we have? So, you use ptr 1's position. And since it contains 2, it tells me that the value, the first non trivial entry in row 1 is located at this position, which is $\text{val}[\text{ptr}[1]]$. So, whatever is the content of ptr 1, you would now look into that location of val .

So, that contains 30. So, the corresponding column position is what? That is simply $\text{cols}[\text{ptr}[1]]$ Since, $\text{ptr}[1]$ is 2, and columns $\text{ptr}[1]$ is containing 1. So that is column 2 that is containing 1, right. So now I have got overall information. The overall information is the first non trivial entry of row 1 which is of value 30. And it is located in column 1. And I already know the row.

So this is going to be 0. And this is 30. Well, in that way, I can say that using val , I will be able to figure out all the non trivial entries in s in each row. But what about other entries? Well, let us

first use this idea to figure out all the first non trivial entries in each row. For example, 0, so let us go here. So that is 10, 0, right. So that means it is right here.

Similarly for 4, you just follow it right. So location 4 here is 0, 1, 2, 3 ,4 i.e. 50 and for 50 what is the column? 0 1 2 3 4 i.e. 2 right. So that is a 50. And the last one is 8. So after 4 what do you really have? 7, right. So go to 7 here 0 1 2 3 4 5 6 7. It is 80 right. So where is 80? Look at it. As it is saying it is in the fifth position. And there will be 2 more positions here, and 80 will come here in the fifth position 0, 1, 2, 3, 4, 5 right.

So this is clear because this is the last location. So all the previous entries must be trivial. So these are all trivial entries. But we know up to this, but what about the rest? Well, the rest is easy to form. As you can see here, it is 0 and the next is 2 that means before this in the first row, I have got 2 values. Well, that means the value next to 10 i.e. the next location is telling you to look into $\text{vals}[2]$.

So that means the value previous to $\text{vals}[2]$ is also in the first row and the column wise location of that, can be got from here. So that tells me 20 is there and then immediately after 2 what do I have? 4 and that takes me right here to 50. And for 50 I know where to go. So essentially what we are following is, if we just subtract them, you know that in each row, how many do you really have.

So, essentially, this 2 minus 0 is telling you how many are there two in the first row, 4 minus this. So that is 2. So you have 2 elements in the second row, or the sorry, the 0th row has 2 elements. The first row has 2 elements. Since the first row has 2 elements, how do you get the second element. Well, if you follow 4, you get to 50. Before that you have 40. If you have followed 2, you got to 30. So whatever is next to 30, that is 40.

And before this one, where do you go from 4? So this is also in the same row. That is the information. Now of course, the location you can get right from here, this is third. So here it must be a 0 and here it must be 40. And since there only 2 elements here, all the other locations has to

be 0 right. So, I hope now, it is clear that if we just keep on following this process, I can just simply keep on building the CSR.

(Refer Slide Time: 17:14)

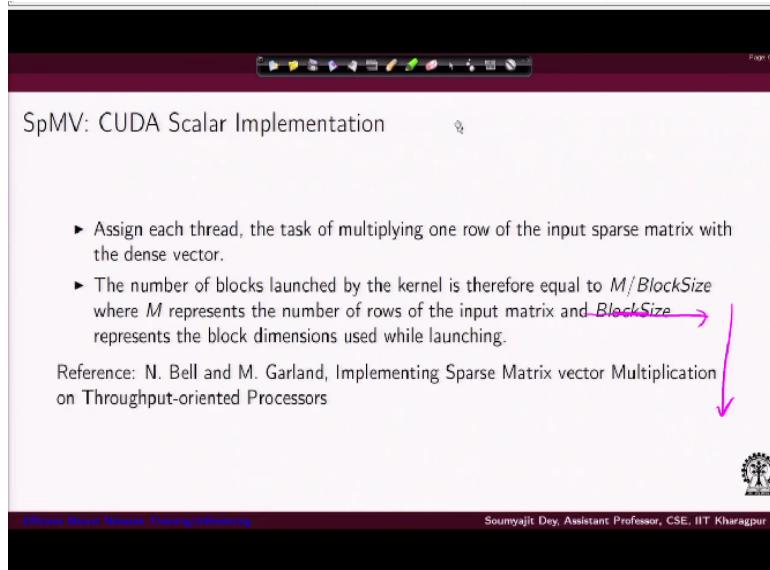
The screenshot shows a presentation slide with the title "SpMV: CPU Implementation". The slide contains a block of C++ code for a sparse matrix-vector multiplication function. The code uses templates and pointers to access matrix elements. It includes nested loops to iterate through rows and columns, and a summation loop to calculate the result. The slide is part of a larger presentation, with a navigation bar at the top and a footer at the bottom.

```
1 template <typename T>
2 void spmv_cpu(T *val, T *vec, int *cols, int *ptr, int N, T *out)
3 {
4     for (int i = 0; i < N; i++){
5         T t = 0;
6         for (int j = ptr[i]; j < ptr[i + 1]; j++){
7             int col = cols[j];
8             t += val[j] * vec[col];
9         }
10        out[i] = t;
11    }
12 }
```

Well, the next thing that we will just shortly touch upon is how really this is going to help? Assuming that you have the storage in the CSR format, you are saving on the storage. Well, how do you really do the multiplication? Well, that is easy. All you need to do is that you already have the vector. All you need to do is that you have to figure out the elements in the corresponding column. You have just required to find out the elements located in the corresponding column because of the vector that is already there.

So how do you just do it? So you run this outer loop. Using this outer loop you are scanning the ptr elements right. And then so for every i, so essentially this outer loop is telling you which row to access. Now for the regular matrix. Now that row is that you have to go here. So suppose for any ith row, you have to start from $\text{ptr}[i]$, and how much really is this inner loop going to iterate? Well, that is easy to get. Just observe our previous definition.

(Refer Slide Time: 18:25)



So all that we said was that this location 0 and this $\text{ptr}[2]$, so what is the difference? That is the number of elements in the 0th row. From 2 to 4 the difference is the number of elements in row 1, For 4 to 7, there is a difference is the number of elements in row 2 and like that. So that is why you start from $\text{ptr}[i]$ and you have to up till $\text{ptr}[i+1]$. How do you get the value? You just get the value in this way, since you have set $\text{ptr}[i]$ to $\text{ptr}[j]$. You will just look into $\text{val}[\text{ptr}[j]]$.

Because that is the val, and the column you will just get because you have already got $\text{ptr}[i]$'s value in j. Just sample from the column matrix $\text{cols}[j]$. That is giving you the column. And once you have got the column, you know how to get the vector values. So, that is how you are going to do the matrix vector multiplication by sampling these vectors' corresponding locations' value. Because this column whatever you are getting, is also telling you which value to sample from the vector.

(Refer Slide Time: 19:31)

```

1  template <typename T>
2  __global__ void spmv_csr_scalar_kernel(T * d_val,T * d_vector,int * d_cols,int
3  * d_ptr,int N, T * d_out)
4  {
5  int tid = blockDim.x * blockIdx.x + threadIdx.x;
6  for (int i = tid; i < N; i += blockDim.x * gridDim.x)
7  {
8      T t = 0;
9      int start = d_ptr[i]; int end = d_ptr[i+1];
10     // One thread handles all elements of the row assigned to it
11     for (int j = start; j < end; j++)
12     {
13         int col = d_cols[j];
14         t += d_val[j] * d_vector[col];
15     }
16     d_out[i] = t;
17 } //Kernel Launch parameters: <<<M/BlockSize),BlockSize>>>

```

I hope that is clear. So with each new iteration, you are going to change the j , i.e. you are simply incrementing j . That means you are just proceeding through the row,. So you are just proceeding through the value matrix. You are going to the next element in the value matrix, because ptr has told you in value matrix at which point to start, right. So that is $\text{val}[j]$, and then you are just incrementing following $\text{val}[j]$ right, I hope this is clear.

So if we just go back in this example, let us say you have started here from 0. You are here, then you are just following the value matrix row wise right. So, all that you need is the start position. Where to start? For example, 50 60 70 is there in a row. You get the idea that have to start here, and then you are just following the elements in the val matrix. And while you follow them, you are also hopping over the corresponding columns in the row matrix.

I hope this is clear. So, again to just give an example, consider that this is your vector. Now, when you are hopping through this row, instead of our normal matrix multiplication, where you start from here and scan through up to here, you are doing it in an optimized way. From ptr you get to val , you get 50, you get the position of 50 which would be 0 1 2 3 4 0 1 2 3 4. That is 2, so you know that you have to start from here.

And you know that you have to go up to this because the position next just up to the next ptr position. So you just scan through this. While you are scanning to whatever you were doing for

each of these values, you also get to know what are their column indice? And then you just scan through those column indices only in the vector because the others are not required because they will be getting multiplied by the 0, right.

So that is how it is getting optimized. So that I hope now is clear from the $\text{ptr}[i]$ to $\text{ptr}[i+1]$ this loop is going to run. You get the starting point j in your val you do $j++$ to skip through the val, for each of the j locations. You also know what is the corresponding column index, and that gives you the which vector component to multiply. And that is how this CPU implementation will work.

Well, how will the CUDA implementation work? Let us do a simple thing first. We will assign each thread the task of multiplying one row of the input sparse matrix with the dense vector. So unlike the CPU implementation, where we had this outer loop now, we will not have this outer loop because every thread will know which row to work with. And accordingly it will just do the job. So, every thread is put in charge of multiplying one row of the input matrix with the vector.

The number of blocks that will be launched will therefore be equal to M divided by the block size. So let us say M represents the number of rows and rows of the matrix and block size represents the block dimension that I want to use while launching right. So every block has got that many threads. So, the number of blocks will simply be $M / \text{BLOCK_SIZE}$ and each block has BLOCK_SIZE number of threads.

So overall, I have got this as many number of threads as there are going to be rows right. And that is as a naive implementation for the corresponding CUDA kernel.

(Refer Slide Time: 22:54)

```

1 template <typename T>
2 __global__ void spmv_csr_scalar_kernel(T * d_val,T * d_vector,int * d_cols,int
3 * d_ptr,int N, T * d_out)
4 {
5     int tid = blockDim.x * blockIdx.x + threadIdx.x;
6     for (int i = tid; i < N; i += blockDim.x * gridDim.x)
7     {
8         T t = 0;
9         int start = d_ptr[i]; int end = d_ptr[i+1];
10        // One thread handles all elements of the row assigned to it
11        for (int j = start; j < end; j++)
12        {
13            int col = d_cols[j];
14            t += d_val[j] * d_vector[col];
15        }
16        d_out[i] = t;
17    } //Kernel Launch parameters: <<M/BlockSize>>,BlockSize>>>

```

So have a look. So this would be my corresponding CUDA kernel. As you can see the I am writing a template enabled code because we are not trying to specify what kind of data type will be there for the matrix. So this will work for any of them right. It can be float, it can be int etc. So this is my kernel. So, since I am charging each thread to multiply one row of the matrix with a column with the vector, the dense vector, we will just find out the tid, the global ID of the thread.

And then inside this loop, we will start from tid. And we will go up to the end actually or whatever is the number of threads. I am going to have the total number of rows, right. And then inside I have this thread that is handling all the elements of the row that have been assigned to it. So as you can see, I am running for this thread this outer loop from tid and am hopping over with block dimension multiplied by grid dimension. Any idea why we are really doing it?

So let us just chop it out. So I want to row wise multiplication. But again, let us remember this is the column major format. So the values I am really interested in, they would be like this right. So the thread in each of its iterations has to get values like this. I mean, from which location of the matrix that i index will be found by incrementing i by block dimension multiplied by the grid dimension.

As you can understand that this multiplication gives me the number of rows. This multiplication is going to be the number of rows of the mathematical matrix. So I will just hop over to get to the next i like this and then with those i's. What am I doing is once am I am inside the row, I have figured out which value to get, then that thread handles all the elements of the row that have been assigned to it.

So, the rest of it is very simple. It is similar to the CPU on implementation. So you start from the `d_ptr[i]` i.e. the ptr matrix position and you are going to end of course at the `d_ptr[i+1]` and for each of these locations, you are sampling the column and you are getting to what location from the vector to get and also you are accumulating here the value in `t` and you are multiplying this vector with the `val` matrix's corresponding location that you have in `j`.

So that is very simple similar to the CPU implementation right. So, in that way you are going to continue and all you are doing is you have to figure out what is the `i` value from where to start in the `ptr` array. So, to get to the different `i` values in the `ptr`, since you have a different representation here, you are having this outer loop. So, I hope that would be okay with the scalar implementation. Now, what about the vector implementation?

(Refer Slide Time: 26:39)

SpMV: CUDA Vector Implementation

- ▶ Each warp of 32 threads take care of one row in the matrix.
- ▶ Shared memory of size equal to the block dimensions used to launch the kernel is leveraged for storing the products of the input sparse matrix and the dense vector.
- ▶ Once all the partial dot-products are finished for a block, the warps perform a partial reduction.
- ▶ The first thread of each warp finally writes to the output vector.

Reference: N. Bell and M. Garland, Implementing Sparse Matrix vector Multiplication on Throughput-oriented Processors

Soumyajit Dey, Assistant Professor

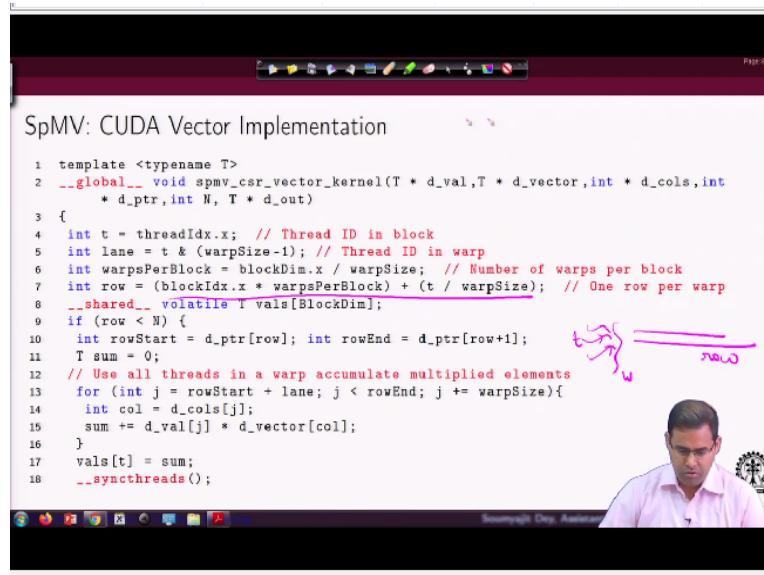
So, it is simple. Let us just optimize like this. We bring back our previous ideas of reduction. So let each warp of 32 threads, together take care of one row in the matrix. So that means we will apply our previous reduction ideas here. So a shared memory whose size is equal to the block

dimension that is used to launch the kernel can be leveraged for storing the products of the input sparse matrix and the dense vector.

So, in some way we will use that. First of all, where are we really doing this? Again, we are considering very large matrices i.e. lot of rows. But again, we also want to optimize. So for that, we want the threads to cooperate. That is why we want each warp to take care of one row of the matrix. And also, while each warp is taking care of one row of the matrix, I want to have shared memory based partial loads.

Where the shared memory size is equal to block dimension, that many threads will be loading the data to the shared memory in a cooperative way. Then they will perform partial dot products. Once the partial dot products are performed, the warp of 32 threads will perform a row wise partial reduction. Then again, I will bring some more data into the shared memory base tile for the next set of points. Again, the warp will perform partial products and partial reduction like that.

(Refer Slide Time: 28:26)



```
1  template <typename T>
2  __global__ void spmv_csr_vector_kernel(T * d_val,T * d_vector,int * d_cols,int
3  * d_ptr,int N, T * d_out)
4  {
5  int t = threadIdx.x; // Thread ID in block
6  int lane = t & (warpSize-1); // Thread ID in warp
7  int warpsPerBlock = blockDim.x / warpSize; // Number of warps per block
8  int row = (blockIdx.x * warpsPerBlock) + (t / warpSize); // One row per warp
9  __shared__ volatile T vals[BlockDim];
10 if (row < N) {
11     int rowStart = d_ptr[row]; int rowEnd = d_ptr[row+1];
12     T sum = 0;
13     // Use all threads in a warp accumulate multiplied elements
14     for (int j = rowStart + lane; j < rowEnd; j += warpSize){
15         int col = d_cols[j];
16         sum += d_val[j] * d_vector[col];
17     }
18     vals[t] = sum;
19     __syncthreads();
20 }
```

So let us see how it works. So in our CUDA vector style implementation, we get the thread id of each block. But then I need to figure out what is the location of this thread inside the warp. So what is that? Let us say that my warp size is 32. 32 minus 1 gives me an array of all 1's. We do

a bitwise and operation. So that gives me the offset that is the thread id in the warp. So essentially, I am getting the thread id.

What is this location inside the wrap? So it is basically a modular 32 operation, I would say. Now, the next thing I figured out is, how many warps do I have inside a block? So I know my block dimension. I am considering one dimensional blocks. And then I just divide the block dimension by the warp size. So that gives me that inside one thread block how many warps I have. Why am I really computing this?

Because I want warps to each of the wraps to do the reduction for one data points, right. So for each block, I figure out how many warps are there per block, and I am really going to use this later on. We will see, first we have computed this warps per block. But the issue is, I want to figure out the threads inside the wraps. They are going to do compute or reduction for which row. That is being taken care of by the next line. I just like to say that this is a bit advanced, but I hope this will be clear.

So you see figured out the position of the thread inside the warps, then I found that this is the total number of threads launched per block. And I want to figure it out by figuring that what is the total block dimension. And by dividing by the warp size, that gives me how many warps do I really have for each block of data.

Now, since I have figured out how many warps I do really have, and I multiply it by the block size, that gives me how many blocks are there already, i.e. how many warps are already done. I am only considering the current block. So this multiplication gives me how many warps I have already got covered by the previous blocks, and then I add with this offset of t divided by the warp size.

So essentially, what does this tell me? First of all, I have figured out, well, what is my thread?. And then let us understand that since a warp of threads are going to work for a row, I have essentially got multiple threads working for the same row. So they all are elements of the same

warp. So by using this formula, I am trying to figure out this in general, any threat t, it is going to work for which row?

That is what I am trying to figure out. So, what do we really do is that we have already figured out what is warps per block, you multiply the block ID. That gives me that how many warps are already done. And then you just edit with the offset. And that tells you that this current thread is going to work on this row. And of course, you have for each block you have got this t which will be there for this block dimension.

The next point that comes is well, I am going to load the data from the row, right. So for that every thread will load the corresponding data as long as this is less than n. So you what you get is with this row value, you figure out what is the position to start from $d_ptr[row]$ and then from $dptr[row+1]$, you know, what is the position to n . I hope this is clear that this transformation is required.

Because every thread is not working for each row, a set of threads inside the warp are going to for work for a row. So every thread by doing this calculation is figuring out for which rows it is going to work. And then I am going to figure out to load data from which point to which point right. Well, once that is done, we use all the threads in our warp to accumulate the multiplied elements. That is what I am going to do. So first, you see I have already figured out from where to start and where to end.

(Refer Slide Time: 32:59)

```

1 // Reduce partial sums
2 if (lane < 16) vals[t] += vals[t + 16];
3 if (lane < 8) vals[t] += vals[t + 8];
4 if (lane < 4) vals[t] += vals[t + 4];
5 if (lane < 2) vals[t] += vals[t + 2];
6 if (lane < 1) vals[t] += vals[t + 1];
7 __syncthreads();
8 // Write result
9 if (lane == 0)
10 d_out[row] = vals[t];
11 }
12 }

```

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant

And that is (as you can see that some part we are skipping here for brevity) basically how the loads will be done into the vals and all that. I hope you can figure it out. But we are just talking about how data from these input matrices dvals and dvectors are actually going to be accumulated here. Then you are going to do is this part, which is the important part.

That is once I have these partial sums calculated, as you can see, I activate the lanes. So this is very much like the reduction optimization we taught in our reduction class where we did reduction inside a warp. So, that is what you can see. We are assigning for each thread which parts of the val is going to add right. So, once the partial products are calculated, after that all of them are written back to this shared memory right.

Now, this is where the importance of the shared memory is coming because now I want the warp to do cooperative reduction. And of course for that, I will need the data to be in shared memory. So first every thread has figured out, which row it is belonging to, and this is non trivial, because as we discussed earlier a set of threads inside a warp are going to work for the same row. Once that is figured out, I just figured out what is my row start and row end.

And I use them for the normal loop for doing the multiplication. After the multiplication for where the accumulation is done, I just stored back the value in the shared memory, where I will do reduction using my warps. And now, using these consecutive if statements, (we are not

getting into these details, this is something we have already covered) we are able to do the final reduction. And that gives me the final result, which is of course, written back by only the thread which has the lane value equal to 0s, right. Because we will have one thread per warp with this value equal to 0 and that gives me the final output.

(Refer Slide Time: 36:00)

Teaching Assistants

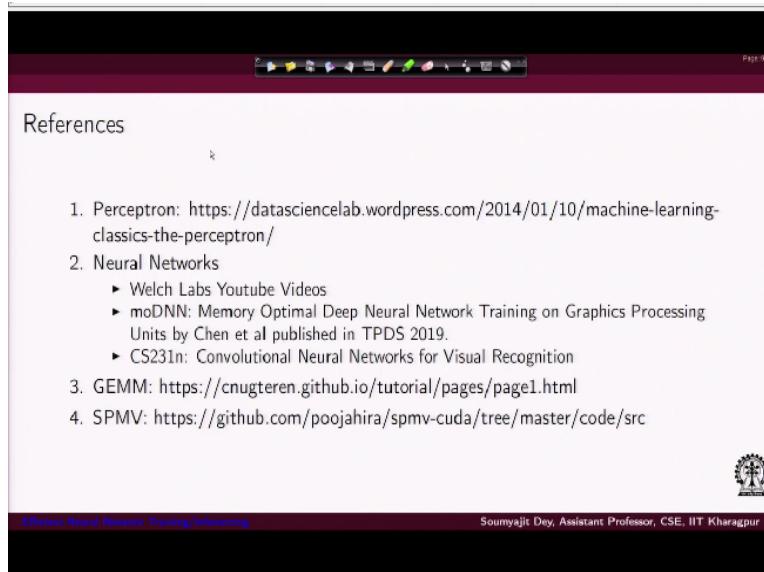
	Anirban Ghose
PhD scholar in Department of Computer Science and Engineering	
IIT Kharagpur	
Research interests: Intelligent Scheduling and Compiler Optimization Techniques for Heterogeneous Architectures	
	Srijeeta Maity
PhD scholar in Department of Computer Science and Engineering	
IIT Kharagpur	
Research interests: Real time scheduling on heterogeneous embedded architectures	

Efficient Neural Network Training/Inference

Soumyajit Dey, Assistant Professor

So with this, we will also like to say that this is an end to the coverage for the course, I hope you had a nice exposure to GPU architectures and programming. And just I would also like to thank 2 of my TAs who are Anirban Ghose and Srijeeta Maity. They are doing PhD under my supervision in the Department of Computer Science, and without their cooperation and help with respect to assignments and this slide creation, this course would really not have been possible.

(Refer Slide Time: 36:35)



And finally, these are the important references that are there for the last set of slides. Of course, for perceptron part we refer to this link, for the neural networks part, there were this nice Welch labs videos. And you can look into this a recent paper called moDNN, where they discuss about optimizing memory consumption while doing neural network training.

Now this is really something that is useful. I mean, that is something we could not cover, but it is really useful. And there were several other sources from which we borrowed material for convolution neural networks, GEMM and the SPMV. So, I hope in general, this was a nice exposure for you. And that would be all from our side. I thank you for attending the course.

**THIS BOOK IS
NOT FOR SALE
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in