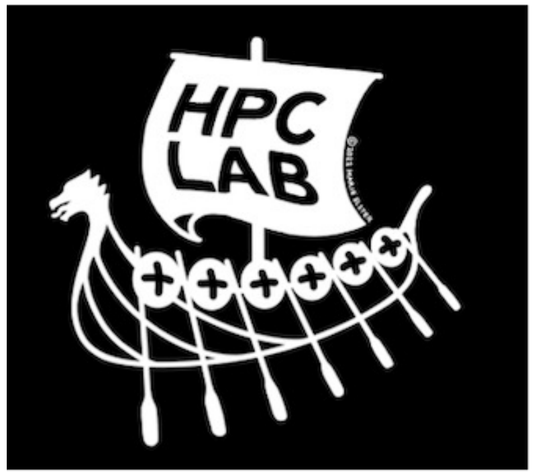


TDT 4200 Fall 2024
Parallel Computing

CUDA Architectures, more on mem, synch and ILP
(based partially on CalTech Lect 4-5 – spring 2024)

Prof. Anne C. Elster, PhD

Dept. of Computer Science (IDI)

Norwegian Univ. of Science & Technology

(NTNU Trondheim, Norway)

&

Univ. of Texas at Austin (Senior Visiting Scientist)

TDT4200 Fall 2024 Student Reference Group

Please contact for praise and suggestions on how we can make this course better:

- Ingar Asheim – ingara@stud.ntnu.no

- Elin Skålund Berntsen – elinsbe@stud.ntnu.no
- Hanne Marie Haakaas – hannamhaa@stud.ntnu.no
- Suzanne Maduga – suzannm@stud.ntnu.no

- Simone Deidier – simondei@stud.ntnu.no
- Davide Esposito – davidees@stud.ntnu.no
- Diego Velasco – diegove@stud.ntnu.no
- ???
- ???

+ Need someone that are from NTNU, but not CS….

TDT4200 F2024 Course Information

– See also https://www.ntnu.edu/studies/courses/TDT4200/

- **PS 5 on CUDA intro out this morning, due Oct 21**
- **Will accept until Oct 28, but then cannot be late on PS6!**

NOTE: Compulsory assignments:

– **You need to do and pass ALL Problem Sets/Exercises in order to take the final**

… also those that are Pass/Fail!!

Updates Lecture schedule. Next week, Tues lecture @ 4pm!

Week 10 (43)			
Lab hrs	Mon Oct 21, 16:00-18:00	Lab hrs with Student TAs in Cybele	
Lect	Tues Oct 22	**MOVED TO RECITATION slot due to Elster´s conflict**	
Lect	Tues Oct 22	**NO CLASS during 10-12 slot**	
Lect 28-29/ Recit	**Tues Oct 22**	**Presentation of PS 6 CUDA Graded** **If time: More on CUDA -- CUDA algorithms -- CalTech/Lect 5 + some of 6?**	Hand out PS 6??
Lect 30	Wed Oct 23	**CUDA algorithms -- CalTech/NVIDA lect on prefix sum (lect 7)**	
Week 11 (44)			
Lab hrs	Mon Oct 28, 16:00-18:00	Lab hrs with Student TAs in Cybele	
Lect	Tues Oct 29	Lecuture moved to 4pm	Elster conflict - NTNU Board meeting
Lect	Tues Oct 29	Lecture moved to 4pm	Elster conflict - NTNU Boad meeting
Recit	Tues Oct 29	**Lect on CuFFT and Elster´s Bit-reversal +** **PS6: CUDA graded Q&A!!**	May have to cancel if Elster has to attend longer NTNU Board meeting
Lect 31?	Wed Oct 30	TBD / CuBLAS /other parallel environments	
Week 12 (45)			
Lab hrs	**Mon Nov 4,** 16:00-18:00	Lab hrs with student TAs in Cybele	**PS 6 CUDA Graded due Monday Nov. 4 at 10pm!! Loss of points if handed in after that. No score/fail after Nov 11, even w/ note.**
Lect 32?	Tues Nov 5	ARM Guest lecture!	
Lect 33?	Tues Nov 5	Advanced CUDA -- pointers to resources, warp-level functions ++	
Recit 13	Tues Nov 5	Final Student TA hour at 16-17 in Cybele instead of recitation lecture at 16:15 in R5	

Outline

- **CUDA Intro**
  - **– review of multiplication example**

-- CUDA memory system

Based on

-- CalTech Slides:

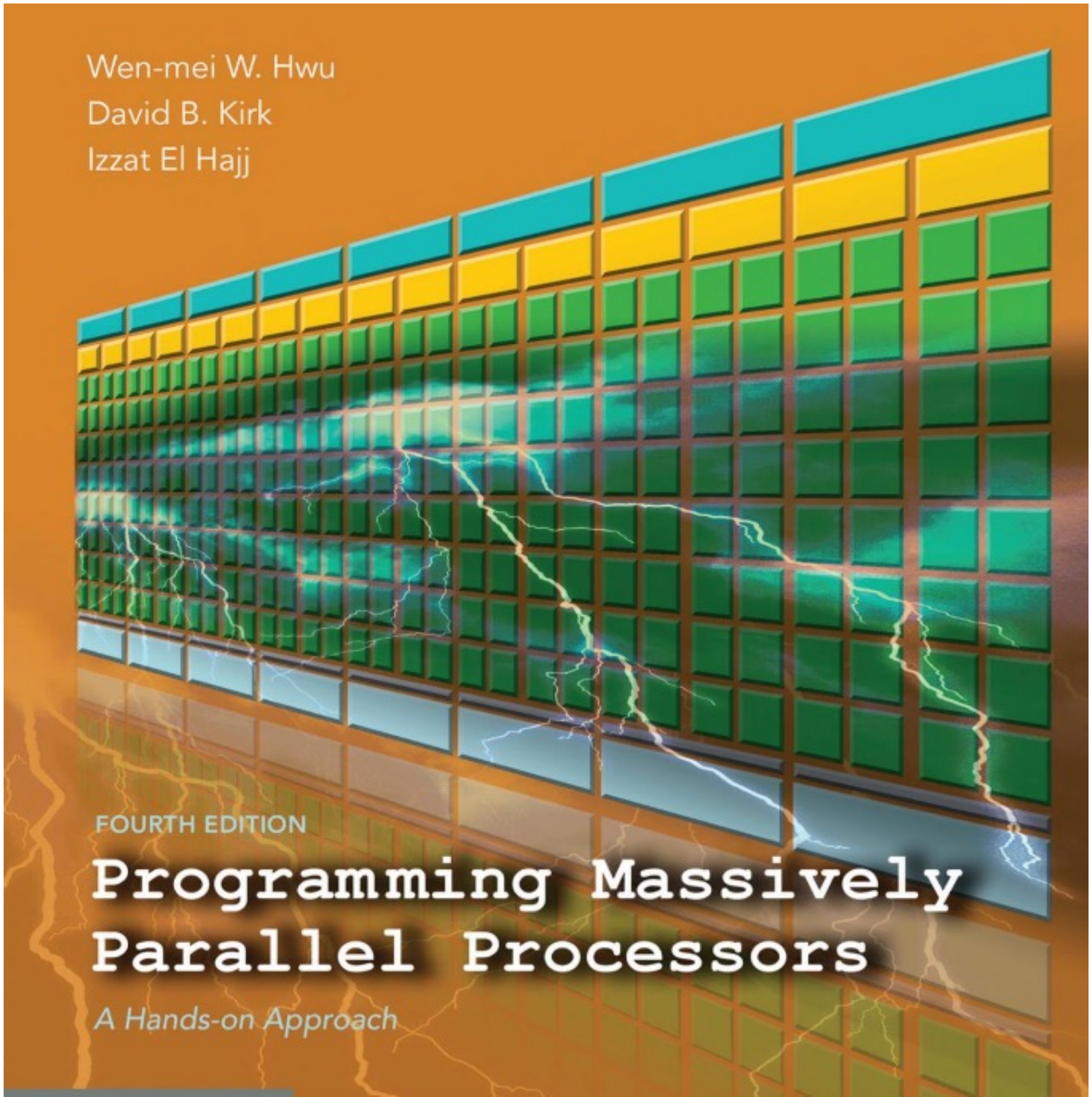
http://courses.cms.caltech.edu/cs179/

(taught Spring 2024)

– Elster CUDA draft

– Kirk CUDA Book

CUDA Book by Hwu, Kirk & Haji, **4th edition** (2022): Chapters 1-8 preview online!



CUDA Book – old version! (v2013)

Bedrift? Klikk her

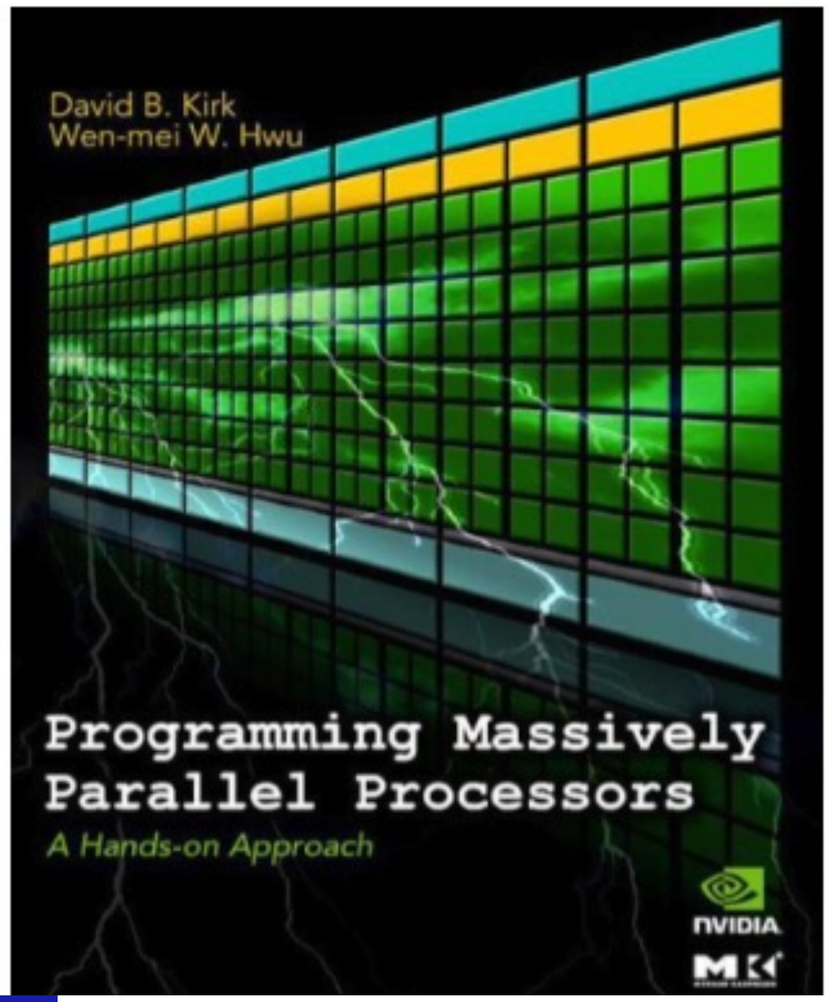
akademika

Meny

Søk på ISBN, boktittel, eller forfatter

Butikk Logg inn Han

Hjem / Teknologi / Data- og informasjonsteknologi / Informatikk / Dataarkitektur og logisk design / Programming Massively Parallel Processors



Programming Massively Parallel Processors

Kirk, David B.

Digital bok / 2013 / Engelsk

Nettpris
Laveste pris siste 30 dager: 616

Dette er en digital bok som er bundet til bokleseren Bookshelf. Du vil få en aktiveringslenke på e-post når du har fullført kjøpet.
Klikk her for mer info.

○ **Lifetime access** **624,-**

PRODUKTBESKRIVELSE

Programming Massively Parallel Processors discusses the basic concepts of parallel programming and GPU architecture. Various techniques for

CUDA Book by Elster & Khan v2014-draft:

May update – provided on BlackBoard later this week

- Includes PS 5 solution. ☺

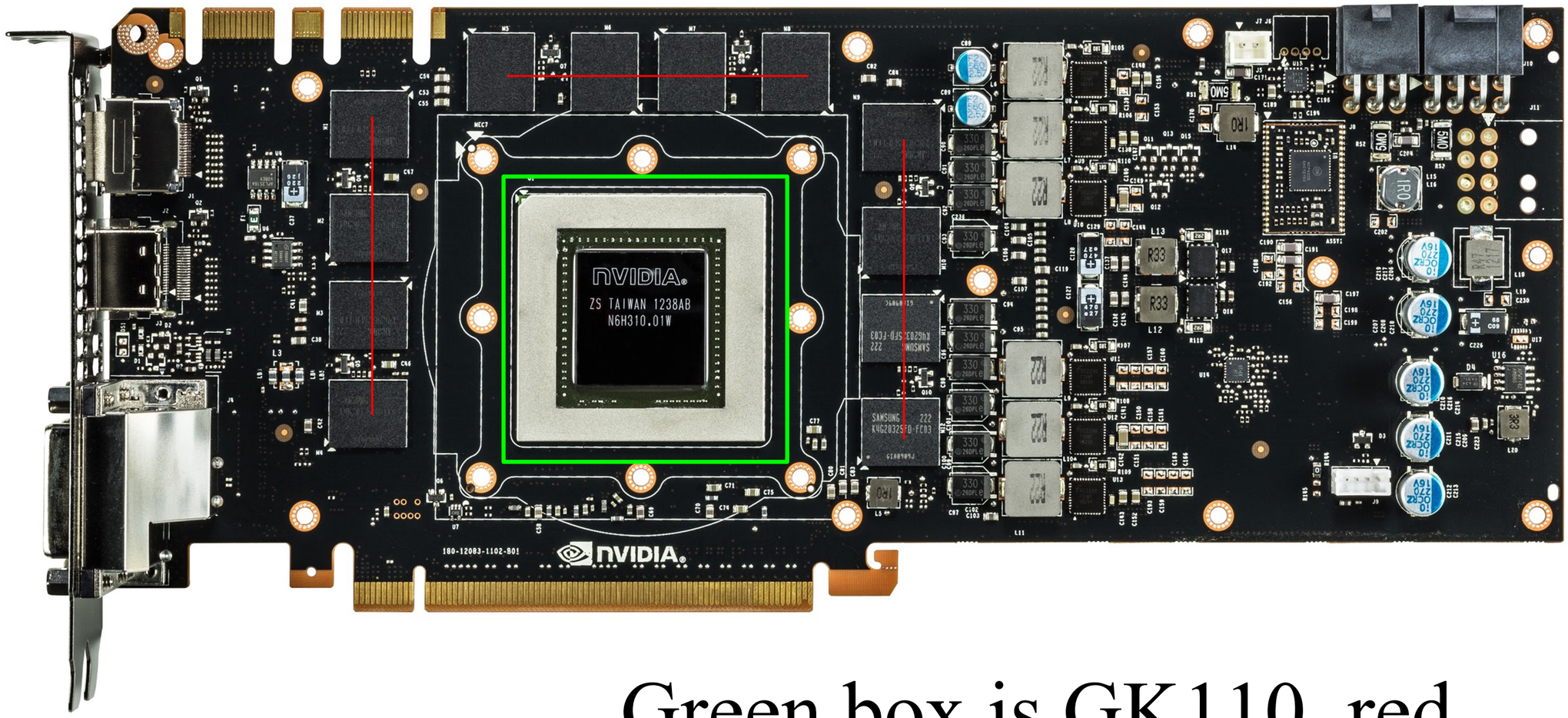
Note Old (3rd editon and older) of Kirk book is available on-line as PDF

CUDA program – step by step

– very similar to PS4!!

1) Setup inputs on the host (CPU-accessible memory)
   - Allocate memory for outputs on the host CPU
   - Create the kernal that performs the calculations
2) Set up device (GPU) memory
   incl. Copy inputs from host to GPU (slow)
3) Execute kernal on device (GPU) - fast!
4) Transfer result from device (GPU) to host (slow)
5) Free the device memory

NOTE: Copying can be asynchronous,
and unified memory management is available



Green box is GK110, red lines are global memory

Nvidia GeForce GTX 780

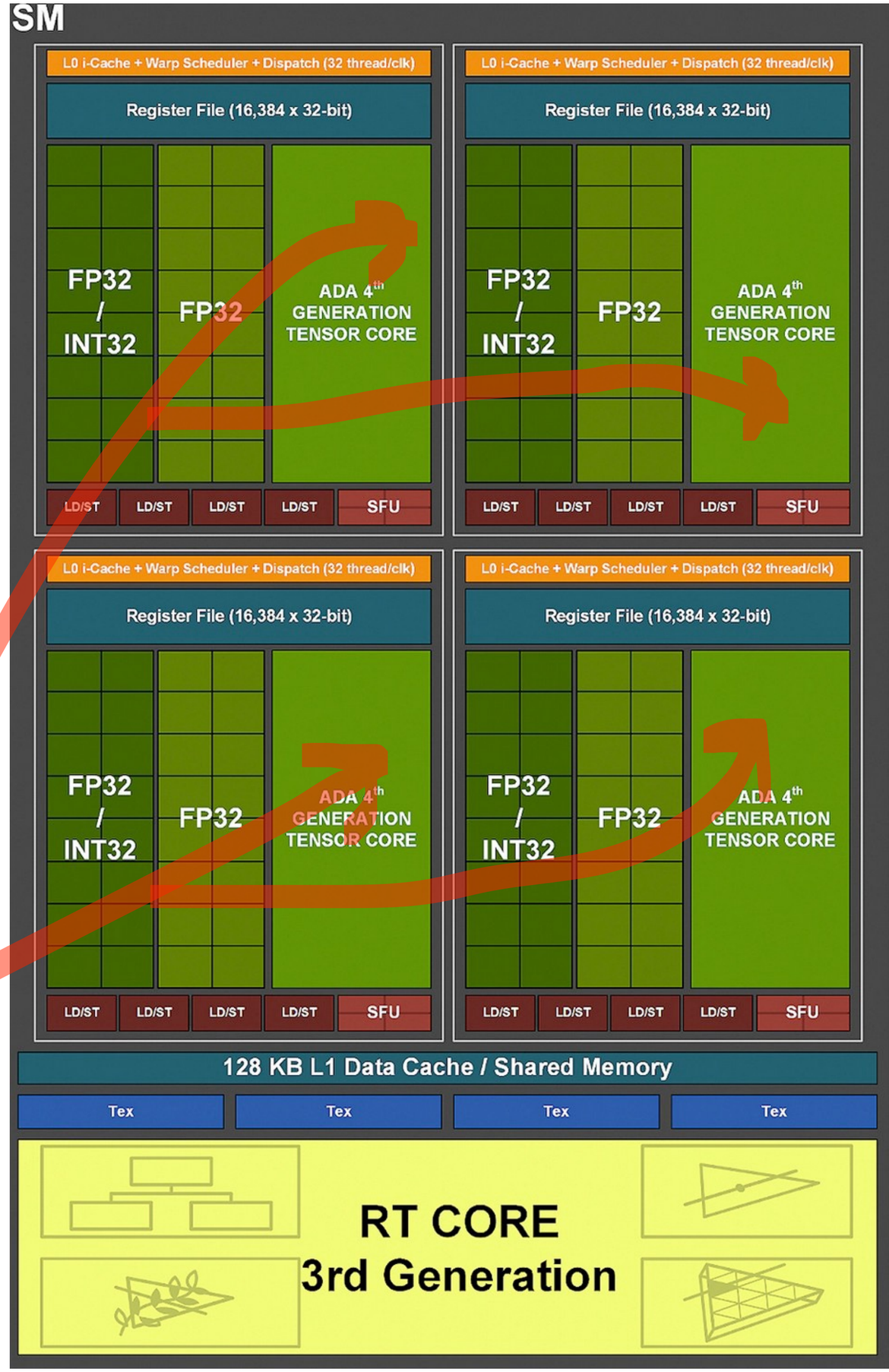
– GK110 is based on Nvidia´s Kepler architecture

CUDA Kernel – indexing Thread Hierarchy
(based on Bart VB slides)

An SM can only have a certain number of threads active at the same time (usually 2048)

- Our problems are usually much larger, and we tend to use many threads

→ Solution: thread hierarchy



NVIDIA Architectures:

- **Blackwell Architecture (March 2024) -** targeting generative AI **Read More**
- **Hopper Architecture (+ ARM-based Grace CPU). Read More . H100**
- **Ada Lovelace Architecture (September 2022) Read More CC8.9,RTX4090**

Previous Architectures:

- **Ampere Architecture (2020) – Tesla A100s (in Betzy), RTX 3050-3090s**
- **Turing Architecture (2018) – T4s in Snotra**
- **Volta Architecture (2017) – Tesla V100**
- **Pascal Architecture (2016) – Tesla P100**
- **Maxwell Architecture (2014) – Tesla M40**
- **Kepler Architecture (2012) – Tesla K20, GTX 780**
- **Fermi Architecture (2010) –**
- **Tesla Architecture (2006) – Tesla 1080,** GeForce 8,9,100-405, Quadro FX++
- Curie Architecture 2004)
- Rankine (2003)
- Kelvin (2001)
- Celsius (1999)

NVIDIA Architecture GH combo:

Grace Hopper GH200	
Designed by	Nvidia
Manufactured by	TSMC
Fabrication process	TSMC 4N
Codename(s)	Grace Hopper
Specifications	
Compute	GPU: 132 Hopper SMs CPU: 72 Neoverse V2 cores
Shader clock rate	1980 MHz
Memory support	GPU: 96 GB HBM3 or 144 GB HBM3e CPU: 480 GB LPDDR5X

More on NVIDIA Grace and Hopper:

A. C. Elster and T. A. Haugdahl, "Nvidia Hopper GPU and Grace CPU Highlights," in *Computing in Science & Engineering*, vol. 24, no. 2, pp. 95-100, 1 March-April 2022, doi: 10.1109/MCSE.2022.3163817.

https://ieeexplore.ieee.org/document/9789536

Last time...

- Global Memory access is not that fast
  - Tends to be the bottleneck in many GPU programs
  - Especially true if done stupidly
    - We'll look at what "stupidly" means

- Optimize memory access by utilizing hardware specific memory access patterns

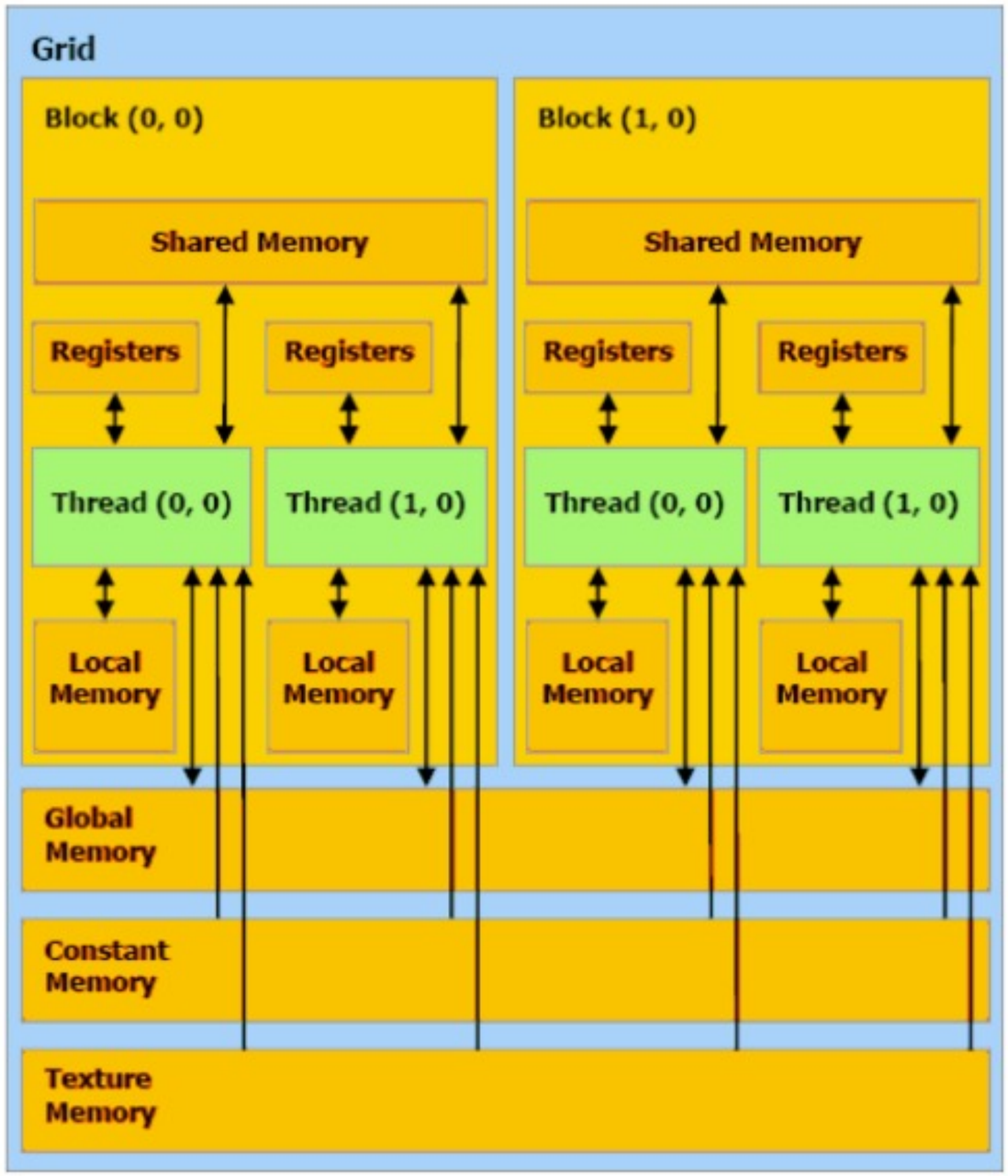
- Optimize memory access by utilizing different caches that come with the GPU

From CalTech CS 179: GPU Programming

LECTURE 4: GPU MEMORY SYSTEMS

GPU Memory Breakdown

- Registers
- Local memory
- Global memory
- Shared memory
- L1/L2/L3 cache
- Constant memory
- Texture memory
- Read-only cache (CC 3.5+)



Accessing global memory efficiently

- Global memory IO is the slowest form of IO on GPU
  –except for accessing host memory (duh...)

- Because of this, we want to access global memory as little as possible

- Access patterns that play nicely with GPU hardware are called **coalesced memory accesses.**

Memory Coalescing

- Memory accesses are done in large groups setup as

  Memory Transactions

  – **Done per warp**

  – Fully utilizes the way IO is setup at the hardware level

- Coalesced memory accesses minimize the number of cache lines read in through these memory transactions

  –GPU cache lines are 128 bytes and are aligned

- Memory coalescing is much more complicated in reality

  –Will cover some examples later, but fairly simple strategies used in this class.

A common pattern in kernels

(1) copy from global memory to shared memory

(2) `__syncthreads()`

(3) perform computation, incrementally storing output in shared memory, `__syncthreads()` as necessary

(4) copy output from shared memory to output array in global memory

Padding to avoid bank conflicts

To fix the stride 32 case, we'll waste a byte on padding and make the stride 33 :)

Don't store any data in slots 32, 65, 98, ...

Now we have
thread 0 ⇒ index 0 (bank 0)
thread 1 ⇒ index 33 (bank 1)
thread `i` ⇒ index `33 * i` (bank `i`)

Registers

- A **Register** is a piece of memory used directly by the processor
  - Fastest "memory" possible, about 10x faster than shared memory
  - There are tens of thousands of registers in each SM
    - Generally works out to a maximum of 32 or 64 32-bit registers per thread

- Most stack variables declared in kernels are stored in registers
  - example: `float x; (duh...)`

- Statically indexed arrays stored on the stack are sometimes put in registers

Local Memory

- **Local memory** is everything on the stack that can't fit in registers

- The scope of local memory is just the thread.

- Local memory is stored in global memory
  –much slower than registers

Questions?

- Global memory
- Local memory
- Shared memory
- Registers

Lecture on Oct 15, 2024 stopped here

Part 2

- L1/L2/L3 cache
- Constant memory
- Texture memory
- read-only cache (CC 3.5)

L1 Cache

- Fermi - caches local & global memory
- Kepler, Maxwell - only caches local memory
  - same hardware as shared memory
  - Nvidia used to allow a configurable size (16, 32, 48KB), but dropped that in recent generations
  - each SM has its own L1 cache

L2 cache

- caches all global & local memory accesses
- ~1MB in size
- shared by all SM's

L3 Cache

– Another level of cache above L2 cache

– Slightly slower (increased latency) than L2 cache but also larger.

Constant Memory

Constant memory is global memory with a special cache

- Used for constants that cannot be compiled into program
- Constants must be set from host before running kernel.

~64KB for user, ~64KB for compiler

- kernel arguments are passed through constant memory

Constant Cache

- 8KB cache on each SM specially designed to broadcast a single memory address to all threads in a warp (called static indexing)

  – Can also load any statically indexed data through constant cache using “load uniform” (LDU) instruction

  – Go to http://www.nvidia.com/object/sc10_cuda_tutorial.html for more details

Constant memory syntax

- In global scope (outside of kernel, at top level of program):
  - `__constant__ int foo[1024];`

- In host code:
  - `cudaMemcpyToSymbol(foo, h_src, sizeof(int) * 1024);`

Texture Memory

Complicated and only marginally useful for general purpose computation
Useful characteristics:

- 2D or 3D data locality for caching purposes through “CUDA arrays”. Goes into special texture cache.
- fast interpolation on 1D, 2D, or 3D array
- converting integers to “unitized” floating point numbers

Use cases:

(1) Read input data through texture cache and CUDA array to take advantage of spatial caching. This is the most common use case.

(2) Take advantage of numerical texture capabilities.

(3) Interaction with OpenGL and general computer graphics

Texture Memory

And that’s all we’re going to say on texture memory.
It’s a complex topic, you can learn everything you want to know about it
from online resources.

Read-Only Cache (CC 3.5+)

- Many CUDA programs don't use textures, but we should take advantage of the texture cache hardware.

- CC ≥ 3.5 makes it much easier to use texture cache.
  - Many `const restrict` variables will automatically load through texture cache (also called read-only cache).

  - Can also force loading through cache with `__ldg` intrinsic function

- Differs from constant memory because doesn't require static indexing

NTNU

Questions?

- Registers
- Global memory
- Local memory
- Shared memory
- L1/L2/L3 cache
- Constant memory
- Texture memory
- Read-only cache (CC 3.5)

Based on:
CalTech CS 179: GPU Programming

LECTURE 5: SYNCHRONIZATION AND ILP

So far...

- **GPU Memory System**
  - Global Memory: the slowest and largest form of memory on the GPU, shared by all grids
    - Coalesced memory access minimizes the number of cache lines read
  - Shared Memory: very fast memory, located on the SM and shared by the block
    - Setup as 32 banks that can be accessed in parallel. Each successive 32-bit words are assigned to successive banks
    - A bank conflict occurs when 2 threads in a warp access different elements in the same bank
  - Registers: fasted memory possible, located on the SM, scope is the thread
  - Local Memory: located on the Global Memory, scope is the thread
  - **L1/L2/L3 Cache**
  - **Texture and Constant Cache**

Rest of this lecture:

- Synchronization and Deadlock
  - -- Digression on Floating Point calculations
  - -- Digression on seeing Compiler optimizations (https://godbolt.org/)

- Atomic Operations

- Instruction Dependencies

- Instruction Level Parallelism (ILP)

- Warp Scheduler

- Occupancy

Synchronization

Ideal case for parallelism:

- no resources shared between threads
- no communication needed between threads

→ PS 5! ☺

However, many algorithms that require shared resources can still be accelerated by massive parallelism of the GPU.

• Need to avoid **Deadlock**! Processes can depend on each other and get stuck!

• Each member of a group can wait for another member, including itself, to take action.

•https://en.wikipedia.org/wiki/Deadlock

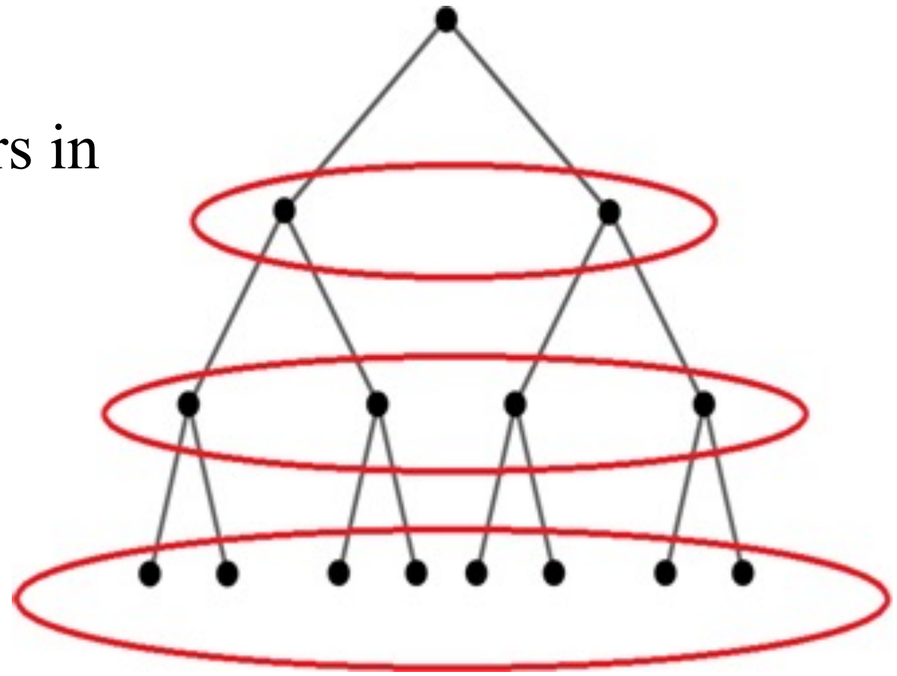
Synchronization -- examples

– Parallel BFS (Breadth First Search)
  - see https://en.wikipedia.org/wiki/Parallel_breadth-first_search
  - (assuming thread touches elements in shared memory that were loaded by other threads. Can't start processing until all threads have finished loading data into shared memory.)

– Accurately summing a list of floating point numbers in parallel -- Many issues! See http://ic.ese.upenn.edu/pdf/parallel_fpaccum_tc2016.pdf

– Loading parallel data into a GPU's shared memory

– Dining philosophers problem –
  - https://en.wikipedia.org/wiki/Dining_philosophers_problem



Digression: Accuracy Issues for Floating Point Calculations

- Finite precision arithmetic! (Use double precision for numerical accuracy. Slower.)

$$(a + b) + c \neq a + (b + c)$$

- Addition is NOT associative, so

For instance, there is a small enough positive number, ε such that $1 + \varepsilon = 1$.

Consider $(\varepsilon + 1) - 1$ which is equal to $(1) - 1$ which is zero.
Re-associate, and compare to $\varepsilon + (1 - 1)$ which is $\varepsilon + (0)$ which is ε .

The two different results are very different!
Not enough bits to retain full accuracy!

Don't add large list of finite precision numbers in uncontrolled order.
May as well be a random number generator!
Add from smallest to largest, to preserve "bits."

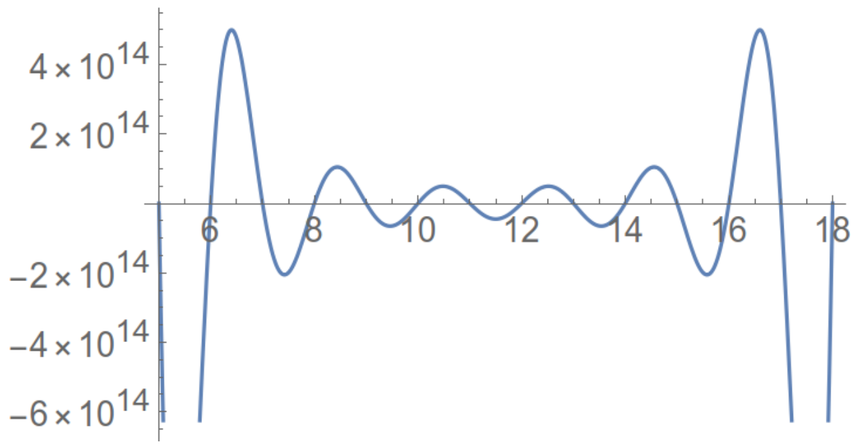
Earliest C compilers assumed addition was associative, and ruined the calculations. Couldn't easily be used for numeric calculations!

How bad can floating point get?

```
f[x_] = Product[(x - i), {i, 1, 22}]
```

```
(-22 + x) (-21 + x) (-20 + x) (-19 + x) (-18 + x) (-17 + x) (-16 + x)
 (-15 + x) (-14 + x) (-13 + x) (-12 + x) (-11 + x) (-10 + x) (-9 + x)
 (-8 + x) (-7 + x) (-6 + x) (-5 + x) (-4 + x) (-3 + x) (-2 + x) (-1 + x)
```

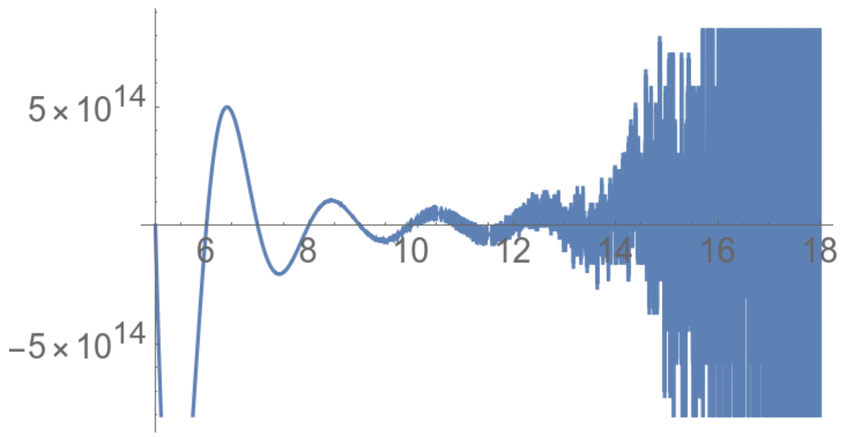
```
Plot[f[x], {x, 5, 18}]
```



Expanded Symbolically :

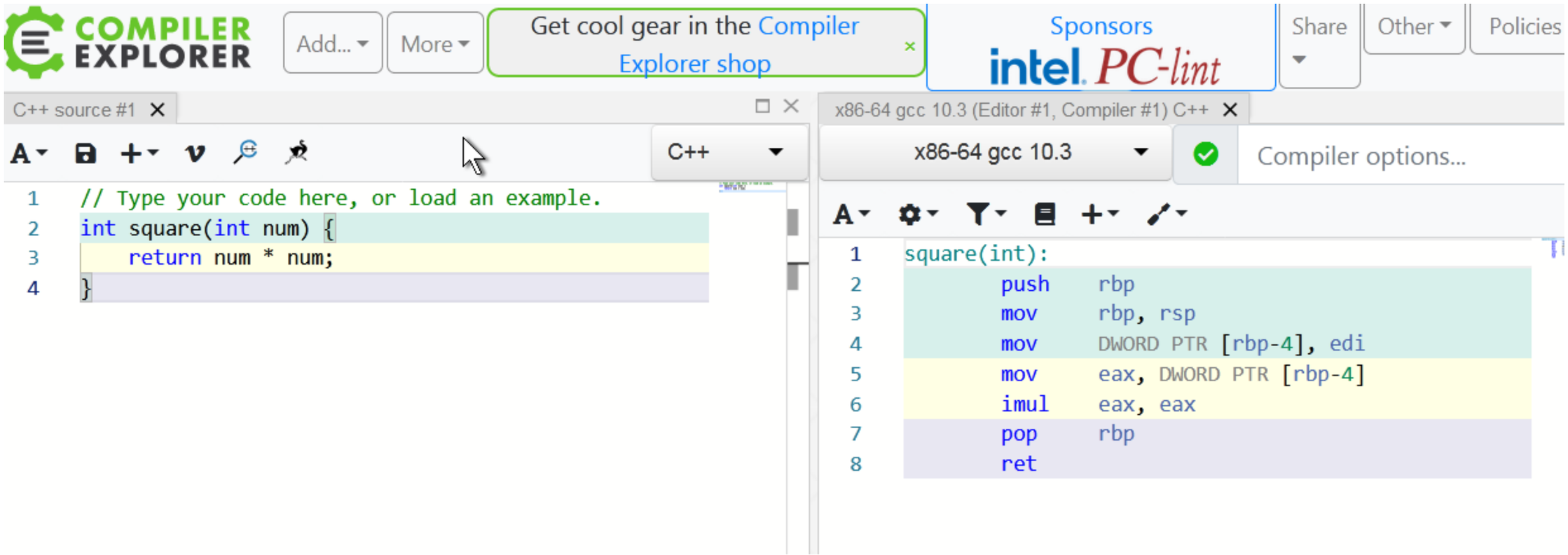
$$1\,124\,000\,727\,777\,607\,680\,000 - 4\,148\,476\,779\,335\,454\,720\,000\,x + 6\,756\,146\,673\,770\,930\,688\,000\,x^2 - |\ldots + 62\,382\,416\,421\,941\,x^{14} - 3\,256\,091\,103\,430\,x^{15} + 136\,717\,357\,942\,x^{16} - 4\,546\,047\,198\,x^{17} + 116\,896\,626\,x^{18} - 2\,240\,315\,x^{19} + 30\,107\,x^{20} - 253\,x^{21} + x^{22}$$

```
Plot[expandfExact[x], {x, 5, 18}]
```



Digression on code from compilers …

- See https://godbolt.org/ for useful site to see result from different compilers
- Sample source code is on the left
- Resulting compiled code is on the right, where you can choose which compiler you're going to examine.



Synchronization

- On a CPU, you can solve synchronization issues using Locks, Semaphores, Condition Variables, etc.
  - Locks -- https://en.wikipedia.org/wiki/Computer_lock
  - Semaphores -- https://en.wikipedia.org/wiki/Semaphore_(programming)
  - Condition Variables -- https://en.wikipedia.org/wiki/Monitor_(synchronization)#Condition_variables_2

- On a GPU, these solutions can frequently introduce too much memory and process overhead
  - Simpler solutions available, many times better suited for parallel programs

CUDA Synchronization

- Usually use the **__syncthreads() function** to sync threads **within a block**

  – Only works at the block level
    - SMs are separate from each other so can't do “better” than this

  – Similar to barrier() function in C/C++or MPI_Barrier
    https://en.wikipedia.org/wiki/Barrier_(computer_science)

  – This __synchthreads() call is very useful for kernels using **shared memory**.

Atomic Operations

- **Atomic Operations** are operations that ONLY happen **in sequence, independent of other processes**
  - For example, if you want to add up N numbers by adding the numbers to a variable that starts in 0, you must add one number at a time
    - Don't do this though. We'll talk about better ways to do this in the next lecture. Only use when you have no other options

"Atomic operations" in concurrent programming are program operations that run completely independently of any other processes.

See https://www.techopedia.com/definition/3466/atomic-operation

CUDA Built-in Atomic Operations

atomic<op>(float *address, float val);

- Replace <op> with one of:
  - Add, Sub, Exch, Min, Max, Inc, Dec, And, Or, Xor
  - e.g. atomicAdd(float *address, float val) for atomic addition

- These functions are all implemented using a function called atomicCAS(int *address, int compare, int val)
  - CAS stands for compare and swap.
  - The function compares *address to compare and swaps the value to val if the values are different
  - Double precision more accurate, but can be much slower!

Instruction Dependencies

An **Instruction Dependency** is a requirement relationship between instructions that force a sequential execution

- In the example on the right, each summation call must happen in sequence because the value of acc depends on the previous summation as well

```
acc += x[0];
acc += x[1];
acc += x[2];
acc += x[3];
...
```

Can be caused by direct dependencies or requirements set by the execution order of code

- I.e. You can't start an instruction until all previous operations have been completed in a single thread

Instruction Level Parallelism (ILP)

- **Instruction Level Parallelism** is when you avoid performances losses caused by instruction dependencies

  – Idea: we do not have to wait until instruction *n* has finished to start instruction *n + 1*

    → *pipelining /latency hiding*

  – In CUDA, also removes performances losses caused by how certain operations are handled by the hardware

ILP Example

```
z0 = x[0] + y[0];
z1 = x[1] + y[1];
```

COMPILATION

```
x0 = x[0];
y0 = y[0];
z0 = x0 + y0;

x1 = x[1];
y1 = y[1];
z1 = x1 + y1;
```

- The second half of the code can't start execution until the first half completes

ILP Example

```
z0 = x[0] + y[0];
z1 = x[1] + y[1];
```

COMPILATION

```
x0 = x[0];
y0 = y[0];
z0 = x0 + y0;

x1 = x[1];
y1 = y[1];
z1 = x1 + y1;
```

- The second half of the code can't start execution until the first half completes

ILP Example

```
z0 = x[0] + y[0];
z1 = x[1] + y[1];
```

COMPILATION

```
x0 = x[0];
y0 = y[0];
x1 = x[1];
y1 = y[1];
z0 = x0 + y0;
z1 = x1 + y1;
```

- Sequential nature of the code due to instruction dependency has been minimized.
- Additionally, this code minimizes the number of memory transactions required

Warp Schedulers

- Warp schedulers find a warp that is ready to execute its next instruction and available execution cores and then start execution
- GPU has at least **one warp scheduler per SM**.

  Each scheduler has **2 dispatchers**.
- The scheduler picks an eligible warp and executes all threads in the warp .
- If any of the threads in the executing warp stalls (uncached memory read) the scheduler makes it inactive.

  If there are no eligible warps left, then GPU **idles**

Context switch between warps is fast – About 1 or 2 cycles (1 nano-second on 1 GHz GPU)

–The whole thread block has resources allocated on an SM by the compiler ahead of time

GK110:

- 4 warp schedulers in each SM and 2 dispatchers in each scheduler
- Can start instructions in up to 4 warps each clock and up to 2 **subsequent, independent** instructions in each warp. Up to 80 warp instructions to hide latency of warp add (10 cycles)

Occupancy

Idea: Need enough independent threads per SM to hide latencies

- Instruction latencies
- Memory access latencies

Occupancy: number of concurrent threads per SM

- Occupancy = active warps per SM / max warps per SM

Number of threads that fit per SM (max warps per SM) is determined by the hardware resources of the GPU.

Threads/block matters because (combined with the number of blocks) let's us know how many warps there are on the SM.

Occupancy

The number of active warps per SM is determined by the limiting resources

- **Registers per thread**
  -- SM registers are partitioned among the threads
- **Shared memory per thread block**
  -- SM shared memory is partitioned among the blocks
- **Threads per thread block**
  -- Threads are allocated at thread block granularity

Needed occupancy depends on the code

- More independent work per thread -> less occupancy is needed
- Memory-bound codes tend to need more occupancy
  Higher latency than for arithmetic, need more work to hide it

Don’t need 100% occupancy for maximum performance

GK110 (Kepler) numbers

- max threads / SM = 2048 (64 warps)
- max threads / block = 1024 (32 warps)
- 32 bit registers / SM = 64k
- max shared memory / SM = 48KB

The number of blocks that run concurrently on a SM depends on the resource requirements of the block!

GK110 Occupancy

100% occupancy

- 2 blocks of 1024 threads
- 32 registers/thread
- 24KB of shared memory / block

50% occupancy

- 1 block of 1024 threads
- 64 registers/thread
- 48KB of shared memory / block

GK110 systems – Nvdia Kepler architecture

E.g. Tesla K20 /K20X
Quadro K6000
GTX Titan (&GB)
- 7.1 billion transistors
- TSMC 28 nm process

Review of Terms …

- Synchronization
- Atomic Operations
- Instruction Dependencies
- Instruction Level Parallelism (ILP)
- Warp Scheduler
- Occupancy

Other Resources

ILP

- https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

Warps and Occupancy

- http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf

Next week...

- NO class @ 10-12 on Tuesday (Elster busy)
- 4pm Tues in Kjel 5: PS 6 presentation

- GPU based algorithms

NTNU

CUDA applications
(covered in CalTech course)

- Solving PDEs on GPUs
- GPU vs CPU fluid mechanics
- Ray Traced Quaternion fractals and Julia Sets
- Deep Learning and GPUs
- Real-Time Signal Processing with GPUs

TDT4200 Fall 2024 Student Reference Group

Please contact for praise and suggestions on how we can make this course better:

Ingar Asheim – ingara@stud.ntnu.no

Elin Skålund Berntsen – elinsbe@stud.ntnu.no
Hanne Marie Haakaas – hannamhaa@stud.ntnu.no
Suzanne Maduga – suzannm@stud.ntnu.no

Simone Deidier – simondei@stud.ntnu.no
Davide Esposito – davidees@stud.ntnu.no
Diego Velasco – diegove@stud.ntnu.no
???
???

Updates Lecture schedule. Next week, Tues lecture @ 4pm!

Week 10 (43)			
Lab hrs	Mon Oct 21, 16:00-18:00	Lab hrs with Student TAs in Cybele	
Lect	Tues Oct 22	**MOVED TO RECITATION slot due to Elster´s conflict**	
Lect	Tues Oct 22	**NO CLASS during 10-12 slot**	
Lect 28-29/ Recit	**Tues Oct 22**	**Presentation of PS 6 CUDA Graded** **If time: More on CUDA -- CUDA algorithms -- CalTech/Lect 5 + some of 6?**	Hand out PS 6??
Lect 30	Wed Oct 23	**CUDA algorithms -- CalTech/NVIDA lect on prefix sum (lect 7)**	
Week 11 (44)			
Lab hrs	Mon Oct 28, 16:00-18:00	Lab hrs with Student TAs in Cybele	
Lect	Tues Oct 29	Lecuture moved to 4pm	Elster conflict - NTNU Board meeting
Lect	Tues Oct 29	Lecture moved to 4pm	Elster conflict - NTNU Boad meeting
Recit	Tues Oct 29	**Lect on CuFFT and Elster´s Bit-reversal +** **PS6: CUDA graded Q&A!!**	May have to cancel if Elster has to attend longer NTNU Board meeting
Lect 31?	Wed Oct 30	TBD / CuBLAS /other parallel environments	
Week 12 (45)			
Lab hrs	**Mon Nov 4,** 16:00-18:00	Lab hrs with student TAs in Cybele	**PS 6 CUDA Graded due Monday Nov. 4 at 10pm!! Loss of points if handed in after that. No score/fail after Nov 11, even w/ note.**
Lect 32?	Tues Nov 5	ARM Guest lecture!	
Lect 33?	Tues Nov 5	Advanced CUDA -- pointers to resources, warp-level functions ++	
Recit 13	Tues Nov 5	Final Student TA hour at 16-17 in Cybele instead of recitation lecture at 16:15 in R5	

Parallel Computing is Fun!