GPGPUs

BITS Pilani
Pilani Campus

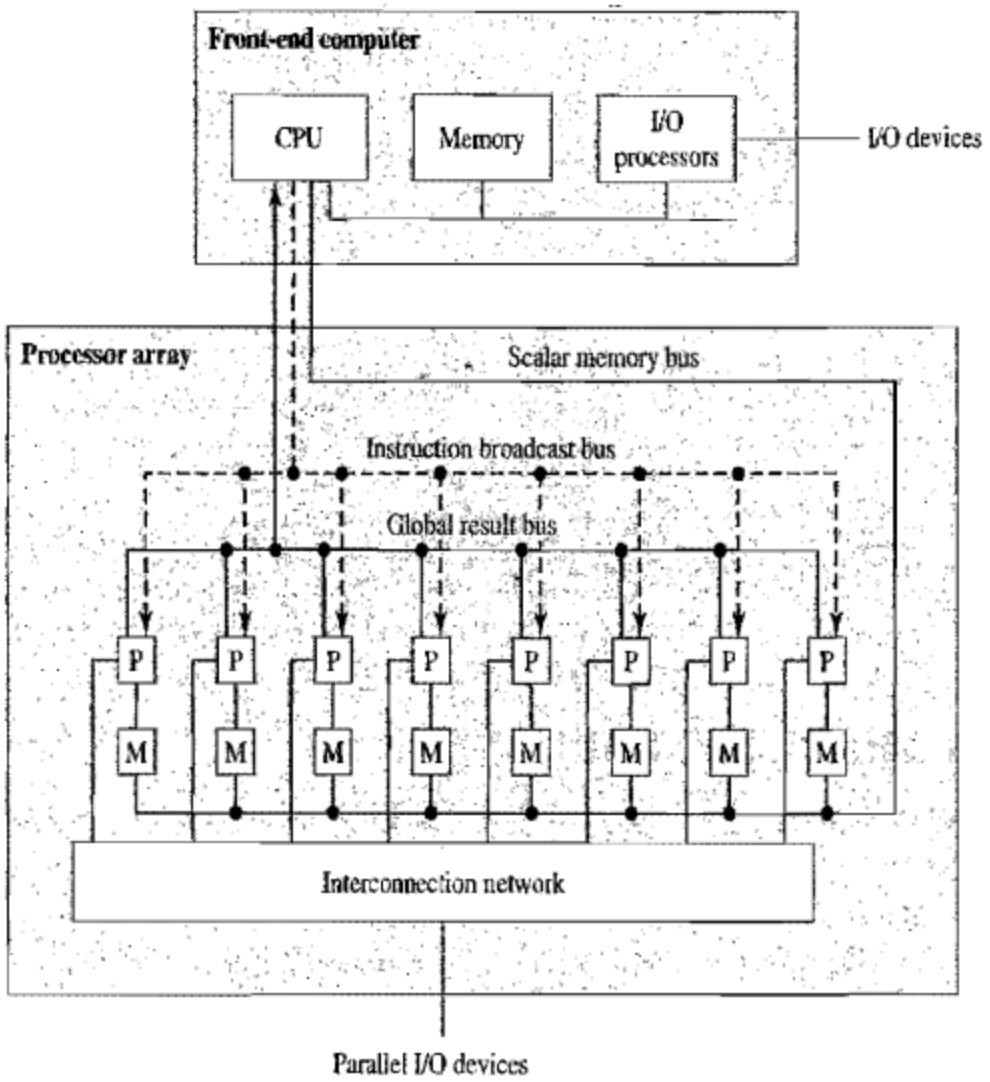
K Hari Babu
Department of Computer Science & Information Systems

Vector Processors

- Scalar processors
  - instructions operate on single data items only
- Vector computer
  - Vectors are large one-dimensional arrays of data values
  - A vector computer is a computer whose instruction set includes operations on vectors as well as scalars
- Two ways of implementing vector computer
  - Pipelined vector processor:
    - Streams vectors from memory to the CPU, where pipelined arithmetic units process them. E.g. Cray-1, Cyber-205
  - Processor arrays:
    - A sequential computer connected to a set of identical, synchronized processing elements capable of simultaneously performing the same operation on different data.

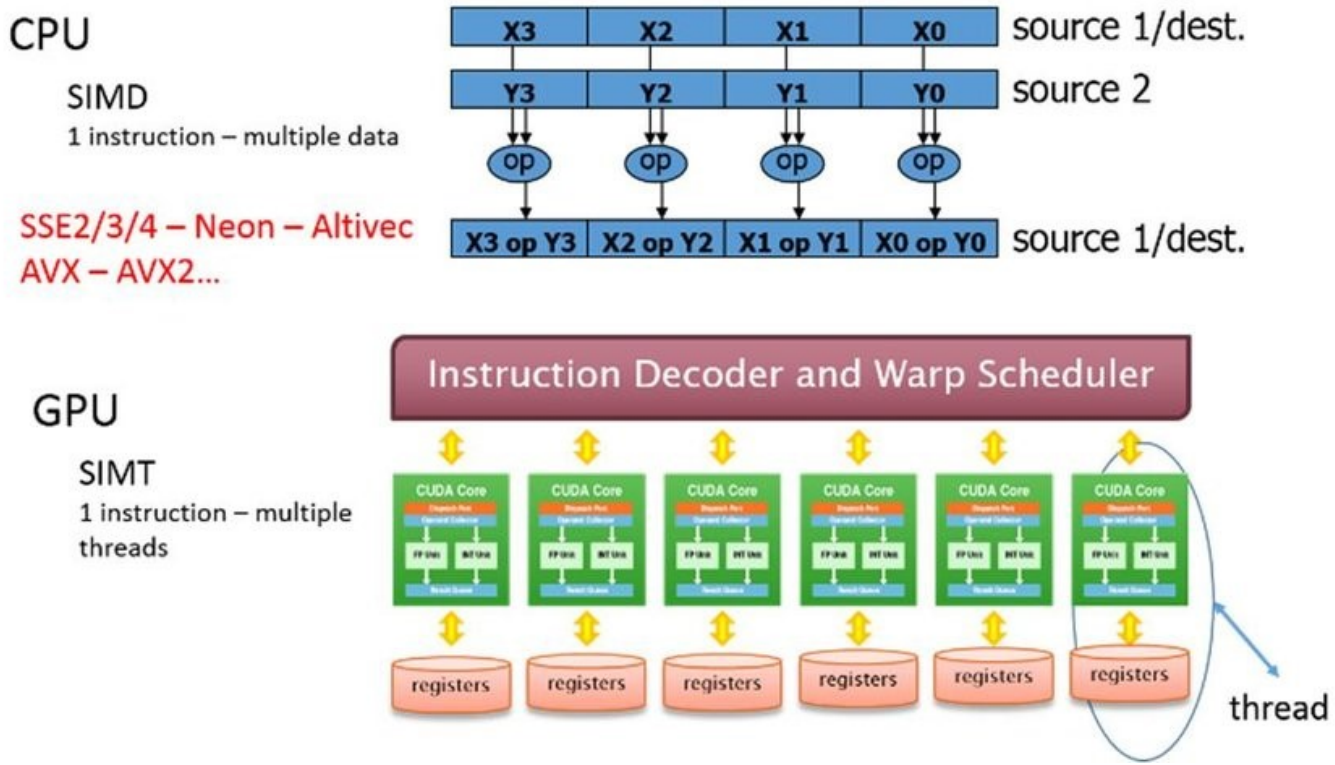
Processor Arrays

Suppose the processor array contains 1024 processors, labeled P0, P1…· Imagine two 1024-elemen vectors A and B are distributed among the processors such that a_i and b_i are in the memory of processor P_i, for all i in the range 0 <= i<= 1024. The processor array can perform the vector addition A + B in a single instruction, because each processor P_i fetches its own pair of values a_i and b_i and performs the addition in parallel with all the other processors.



SIMD vs SIMT

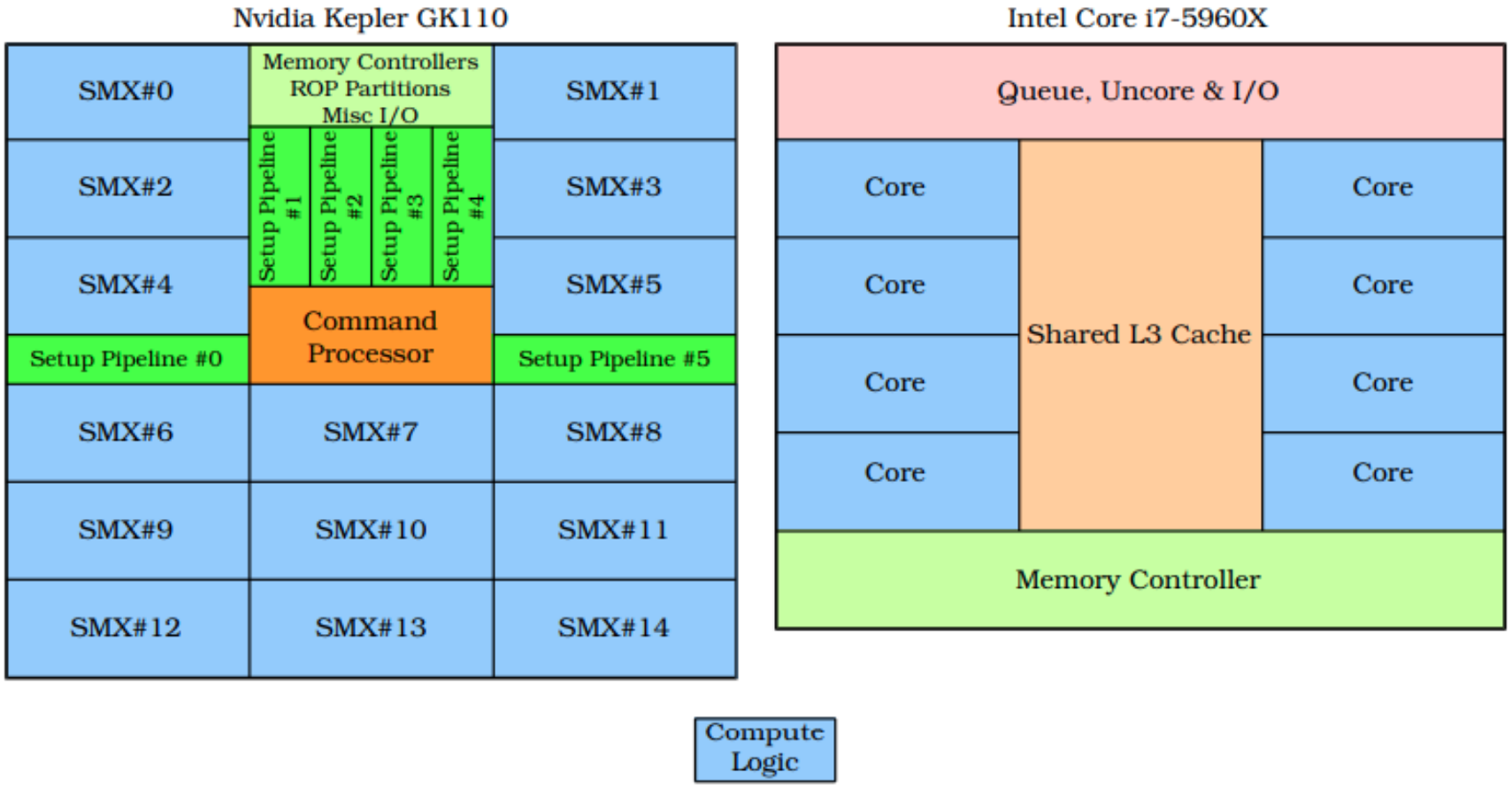
- SIMT (Single Instruction Multiple Threads) is analogous to SIMD
- The only major difference is that in SIMT the size of the “vector” on which the processing elements operate is determined by the software, i.e., the block size



GPU vs CPU

- CPU design
  - CPUs employ large on-chip memory caches, few complex (e.g., pipelined) arithmetic and logical processing units (ALUs), and complex instruction decoding and prediction hardware to avoid stalling
- Graphics Processing Units (GPUs) design
  - developed for processing massive amount of graphics data very quickly
    - their layout departed from the one used by conventional CPUs
  - GPU design: small on-chip caches with a big collection of simple ALUs
    - since data reuse is typically small for graphics processing
  - fast memory buses for fetching data from the GPU's main memory.

GPU vs CPU



- Block diagrams of the Nvidia Titan GPU and the Intel i7-5960X octa-core CPU.
  - clear that while memory cache dominates the die in the CPU case, compute logic dominates in the case of the GPU

Nvidia's KEPLER

- Kepler is the third GPU architecture of Nvidia
- The cores in a Kepler GPU are arranged in groups called Streaming Multiprocessors (abbreviated to SMX in Kepler)
- Each Kepler SMX contains 192 cores that execute in a SIMD fashion, i.e., they run the same sequence of instructions but on different data. Each SMX can run its own program.
  - The most powerful chip in the Kepler family is the GTX Titan, with a total of 15 SMXs
  - One of the SMXs is disabled in order to improve production yields, resulting in a total of $14 \cdot 192 = 2688$ cores!

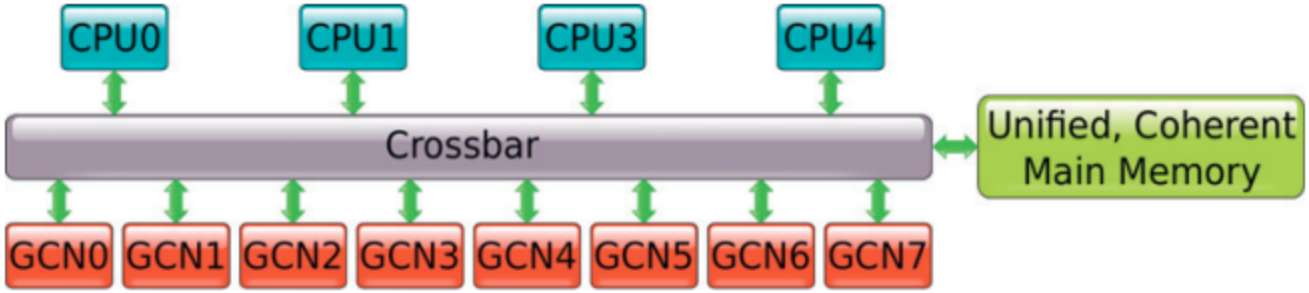
Programming GPU

- GPU is used as a coprocessor, assigned work items from the main CPU. The CPU is referred to as the host.
- *How do we generate 2688 threads?*
  - the GPU programming environment allows the launch of special functions called **kernels** that run with distinct, built-in variables
  - the number of threads that can be generated with a single statement/kernel launch comes to tens of thousands or even millions
    - Each thread will take its turn running on a GPU core
- The sequence can be summarized: (a) host sending data to the GPU, (b) host launching a kernel, and (c) host waiting to collect the results.

Heterogeneous System Architecture (HSA)

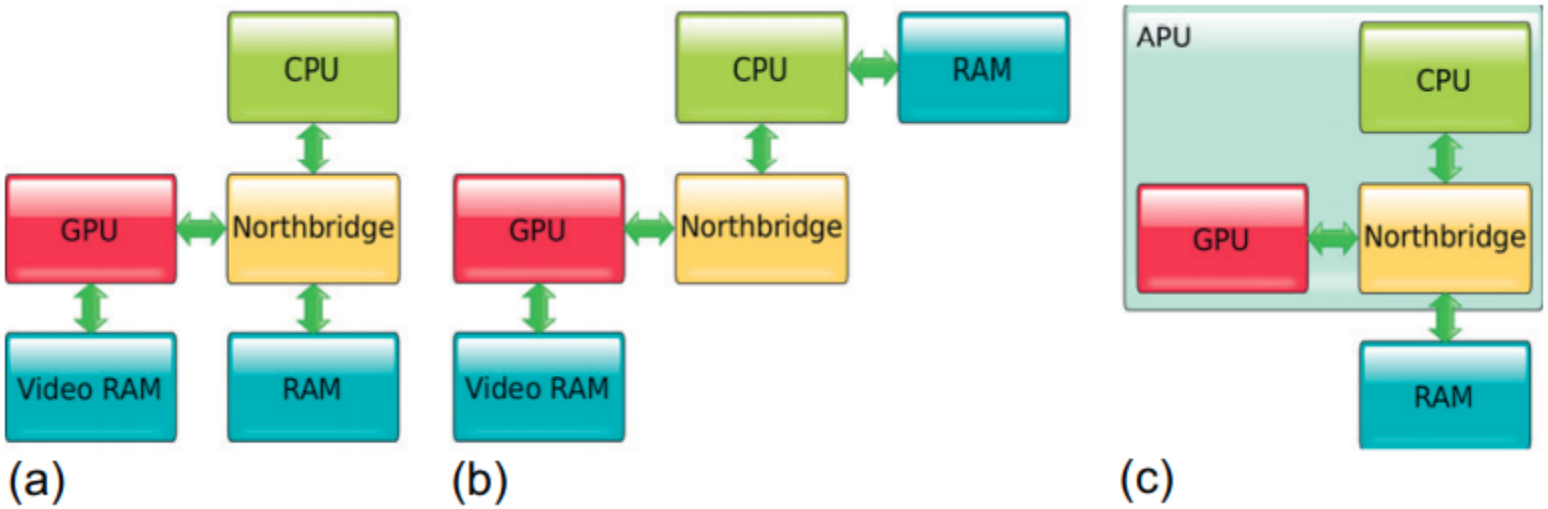
- AMD's line of APU (Accelerated Processor Unit) processors combine CPU and GPU cores on the same die
  - What is significant is the unification of the memory spaces of the CPU and GPU cores
  - no communication overhead associated with assigning workload to the GPU cores, nor any delay in getting the results back
- AMD's APU chips implement the Heterogeneous System Architecture (HSA)
  - Code written in HSAIL (intermediate language IL) is translated to a compute unit's native instruction set before execution. Because of this CPU cores can run GPU code and an HSA application can run on any platform.

AMD's Kaveri Architecture



- Block diagram of AMD's Kaveri architecture
  - GCN stands for Graphics Core Next, the designated name for AMD's next generation GPU core
- HSA is arguably the way forward, having the capability to assign each task to the computing node most suitable for it, without the penalty of traversing a slow peripheral bus.
  - Sequential tasks are more suitable for the CPU cores, while data-parallel tasks can take advantage of the high-bandwidth, high-computational throughput of the GPU cores.

Existing Architectures for CPU-GPU Systems



- (a) and (b) represent discrete GPU solutions, with a CPU-integrated memory controller in (b). Diagram (c) corresponds to integrated CPU-GPU solutions, such as the AMD's Accelerated Processing Unit (APU) chips.
  - sharing the same chip limits the amount of functionality that can be incorporated for both CPU and GPU alike

Challenges

- Having disjoint memories means that data must be explicitly transferred between the two whenever data need to be processed by the GPU or results collected by the CPU
  - communicating data over relatively slow peripheral buses like the PCIe (16GB/sec) is a major problem
- GPU devices may not adhere to the same floating-point representation and accuracy standards as typical CPUs
  - This can lead to the accumulation of errors and the production of inaccurate results

GPU Development Platforms

- CUDA: Compute Unified Device Architecture
  - broke free of the “code-it-as-graphics” approach used until then
- OpenCL: Open Computing Language
  - open standard for writing programs that can execute across a variety of heterogeneous platforms that include GPUs, CPU, DSPs, or other processors.
  - OpenCL is supported by both Nvidia and AMD. It is the primary development platform for AMD GPUs.
  - OpenCL’s programming model matches closely the one offered by CUDA

GPU Development Platforms

- OpenACC:
  - An open specification for an API that allows the use of compiler directives (e.g., #pragma acc, in a similar fashion to OpenMP) to automatically map computations to GPUs or multicore chips according to a programmer's hints.
- Thrust:
  - A C++ template library that accelerates GPU software development by utilizing a set of container classes and a set of algorithms to automatically map computations to GPU threads.
  - Thrust used to rely solely on a CUDA backend, but since version 1.6 it can target multiple device back-ends, including CPUs. Thrust has been incorporated in the CUDA SDK distribution since CUDA 4.1

innovate achieve lead

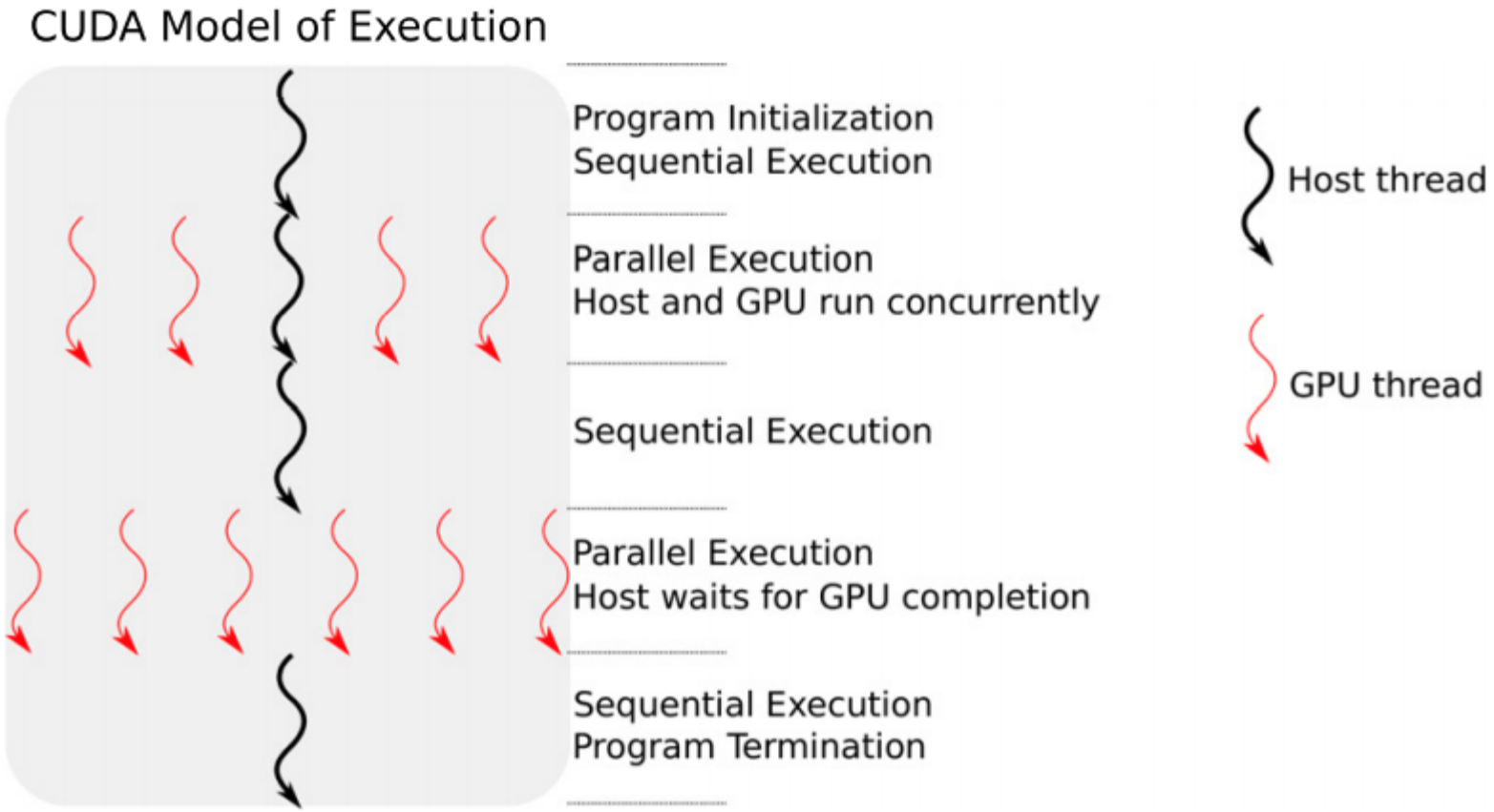
BITS Pilani

Pilani Campus

CUDA

CUDA'S Programming Model: Threads, Blocks, Grids

- GPUs are coprocessors that can be used to accelerate parts of a program
  - A CUDA program executes like a sequential program delegating whenever parallelism is required.



CUDA'S Programming Model: Threads, Blocks, Grids

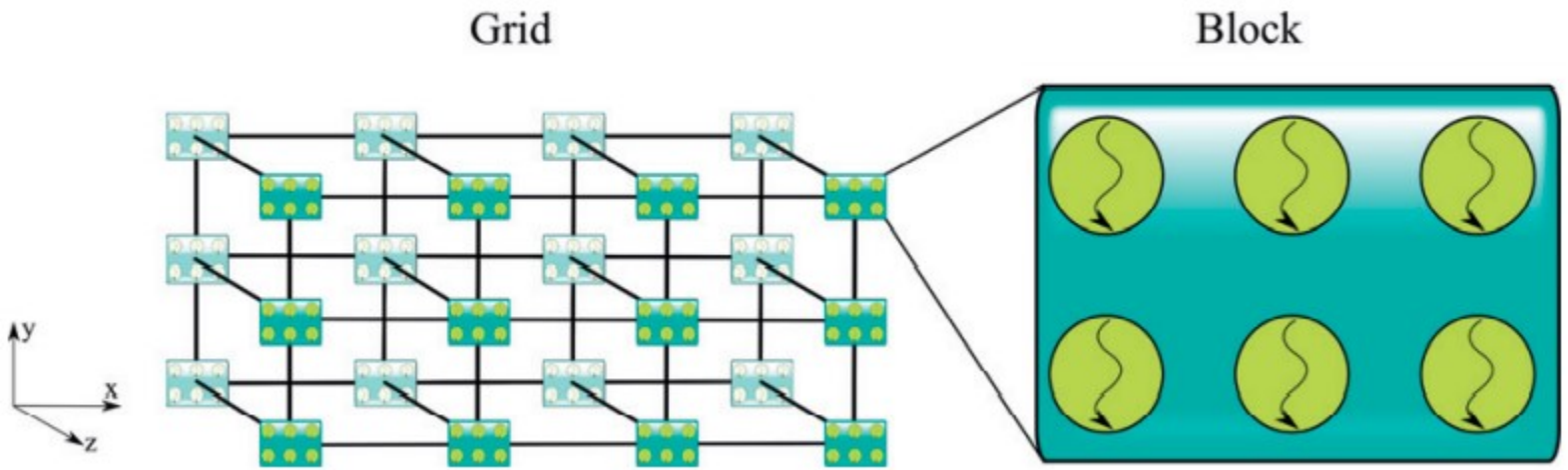
- To properly utilize a GPU, the program must be decomposed into a large number of threads that can run concurrently
  - GPU schedulers execute these threads with minimum switching overhead and under a variety of configurations based on the available device capabilities
- Challenges that must be overcome:
  - How do we spawn the hundreds or thousands of threads required?
    - CUDA's answer is to spawn all the threads as a group, running the same function with a set of parameters that apply to all
  - How do we initialize the individual threads so that each one does a different part of the work?

CUDA'S Programming Model: Threads, Blocks, Grids

- Challenges that must be overcome:
  - How do we spawn the hundreds or thousands of threads required?
    - CUDA's answer is to spawn all the threads as a group, running the same function with a set of parameters that apply to all
  - How do we initialize the individual threads so that each one does a different part of the work?
    - CUDA solves this problem by organizing the threads in a 6D structure (lower dimensions are also possible)! Each thread is aware of its position in the overall structure via a set of intrinsic structure variables. With this information, a thread can map its position to the subset of data to which it is assigned

CUDA'S Programming Model: Threads, Blocks, Grids

- Threads are organized in a hierarchy of two levels
  - At the lower level, threads are organized in blocks that can be of one, two or three dimensions.
  - Blocks are then organized in grids of one, two, or three dimensions.
  - The sizes of the blocks and grids are limited by the capabilities of the target device



CUDA'S Programming Model: Threads, Blocks, Grids

- Nvidia uses the compute capability specification to encode what each family/- generation of GPU chips is capable of. The part that is related to grid and block sizes is shown below:

	Compute Capability			
Item	**1.x**	**2.x**	**3.x**	**5.x**
Max. number of grid dimensions	2	3		
Grid maximum x-dimension	$2^{16} - 1$		$2^{31} - 1$	
Grid maximum y/z-dimension	$2^{16} - 1$			
Max. number of block dimensions	3			
Block max. x/y-dimension	512	1024		
Block max. z-dimension	64			
Max. threads per block	512	1024		
GPU example (GTX family chips)	8800	480	780	980

CUDA'S Programming Model: Threads, Blocks, Grids

- The programmer has to provide a function that will be run by all the threads in a grid
  - This function in CUDA terminology is called a **kernel**.
  - During the invocation of the kernel, one can specify the thread hierarchy with a special execution configuration syntax (<<< >>>)

```
1  dim3 block (3 ,2) ;
2  dim3 grid (4 ,3 ,2) ;
3  foo<<<grid , block >>>() ;
```

  - The CUDA-supplied dim3 type represents an integer vector of three elements. If less than three components are specified, the rest are initialized by default to 1.

CUDA'S Programming Model: Threads, Blocks, Grids

- In the special case where a 1D structure is desired, e.g., running a grid made up of five blocks, each consisting of 16 threads, the following shortcut is also possible:

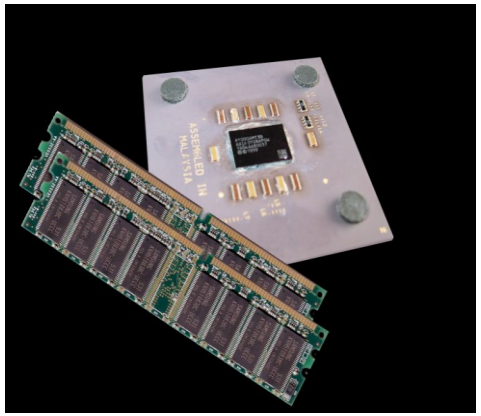
```
1  foo <<<5, 16>>>() ;
```

- Many different grid-block combinations are possible. Some examples are shown here:

```
1  dim3 b (3 ,3 ,3);
2  dim3 g (20 ,100);
3  foo<<<g, b>>>();//Run a 20 x100 grid made of 3x3x3 blocks
4  foo <<<10, b>>>();//Run a 10-block g rid, each block made by
5                       //3x3x3 threads
6  foo<<<g, 256>>>();//Run a 20 x100 g rid, made of 256 threads
7  foo<<<g, 2048>>>();//An invalid example: maximum blocksizeis
8                 //1024 th reads even for compute capability 5.x
9  foo <<<5, g>>>() ; //Another invalid example,that specifies
10                //a block size of 20x100=2000 threads
```

Hello World! In CUDA

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

Device

Hello World! In CUDA

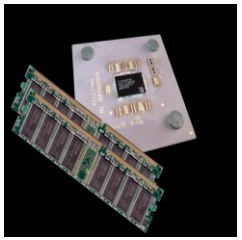
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

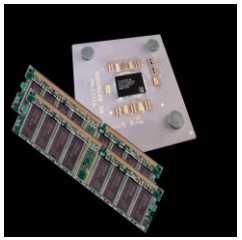
parallel fn

serial code

parallel code

serial code





Hello World! In CUDA

- Standard C that runs on the host
  - NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

```
1  int main(void) {
2          printf("Hello World!\n");
3          return 0;
4      }
```

- Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$
```

Hello World! In CUDA

```
2  #include <stdio . h>
3  #include <cuda . h>
4
5  __global__ void hello ( )
6  {
7    printf ( " Hello world \ n" ) ;
8  }
9
10 int main ( )
11 {
12  hello <<<1,10>>>() ;
13  cudaDeviceSynchronize () ;
14  return 1;
15 }
```

The program can be compiled and executed as follows (CUDA programs should be stored in files with a .cu extension):

```
$ nvcc -arch=sm_20 hello.cu -o hello
$ ./hello
```

- The “architecture” switch (-arch=sm_20) in the Nvidia CUDA Compiler (nvcc) driver command line above instructs the generation of GPU code for a device of compute capability 2.0. Compatibility with 2.0 and higher

Hello World! In CUDA

```
2  #include <stdio . h>
3  #include <cuda . h>
4
5  __global__ void hello ( )
6  {
7    printf ( " Hello world \ n" ) ;
8  }
9
10 int main ( )
11 {
12  hello <<<1,10>>>() ;
13  cudaDeviceSynchronize () ;
14  return 1;
15 }
```

- The __global__ directive specifies that the hello function is supposed to be called from the host and run on the device
- Kernels that are called from the host (i.e., __global__) are not supposed to return any value
- GPU execution is asynchronous i.e., program will not wait for the statement of line 12. An explicit barrier statement is needed (cudaDeviceSynchronize in line 13) to seethe output.

Thread's position

- CUDA threads are aware of their position in the grid/block hierarchy via the following intrinsic/built-in structures, all having 3D components x, y, and z.
  - blockDim: Contains the size of each block, e.g., (B_x, B_y, B_z)
  - gridDim: Contains the size of the grid, in blocks, e.g., (G_x, G_y, G_z).
  - threadIdx: The (x, y,z) position of the thread within a block, with $x \in [0, B_x - 1]$, $y \in [0, B_y - 1]$, and $z \in [0, B_z - 1]$.
  - blockIdx: The (b_x, b_y, b_z) position of a thread's block within the grid, with $b_x \in [0, G_x - 1]$, $b_y \in [0, G_y - 1]$, and $b_z \in [0, G_z - 1]$
- Knowing thread's position is to allow it to identify its workload.

Thread's position

- *threadIdx* is not unique among threads
  - there could be two or more threads in different blocks with the same index
- Deriving a unique scalar ID for each of the threads would require using block and grid info.
  - Each thread can be considered an element of a 6D array with the following definition

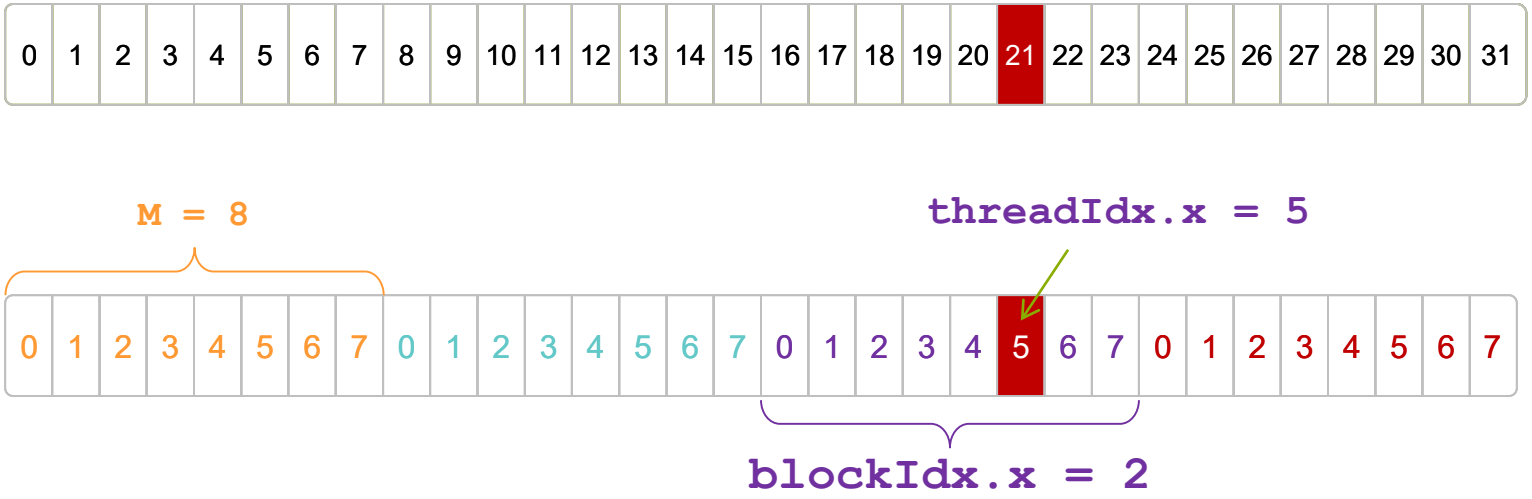
```
Thread t[gridDim.z][gridDim.y][gridDim.x][blockDim.z][blockDim.y][blockDim.x];
```

  - Getting the offset of a particular thread from the beginning of the array would produce a unique scalar ID

```
2  int myID = (blockIdx.z*gridDim.x*gridDim.y+
3  blockIdx.y*gridDim.x+
4  blockIdx.x)*blockDim.x*blockDim.y*blockDim.z+
5  threadIdx.z*blockDim.x*blockDim.y+
6  threadIdx.y*blockDim.x+
7  threadIdx.x;
```

Thread's Position in Single Dimension

- M= threads per block.
- blockIdx.x * blockDim.x + threadIdx.x gives thread position from the beginning.




```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2      * 8;
          = 21;
```

Thread's Position in 3 Dimensions

```
int altMyID = threadIdx.x + blockIdx.x * blockDim.x +
(blockIdx.y * blockDim.y + threadIdx.y) * gridDim.x *
blockDim.x+
(blockIdx.z * blockDim.z + threadIdx.z) * gridDim.x *
blockDim.x * gridDim.y * blockDim.y;
```

```
3  __global__ void hello ( )
4  {
5    int myID = ( blockIdx.z * gridDim.x * gridDim.y +
6    blockIdx.y * gridDim.x +
7    blockIdx.x ) * blockDim.x +
8    threadIdx.x ;
9
10   printf ( " Hello world from %i \ n" , myID ) ;
11 }
```

CUDA'S Execution Model: Streaming Multiprocessors and Warps

- When a kernel is run on a GPU core, the same instruction sequence is synchronously executed by processing units called streaming processors (SP)
- A group of SPs that execute under the control of a single control unit is called a streaming multiprocessor (SM)
- A GPU can contain multiple SMs, each running each own kernel
- Since each thread runs on its own SP, we will refer to SPs as cores
- The computational power of a GPU is largely determined by the number of SMs available

CUDA'S Execution Model: Streaming Multiprocessors and Warps

- Threads are scheduled to run on an SM as a block
  - The threads in a block do not run concurrently, instead they are executed in groups called warps
  - The size of a warp is hardware-specific
  - The current CUDA GPUs use a warp size of 32
    - At any time instance and based on the number of CUDA cores in an SM, we can have 32 threads active (one full active warp), 16 threads active (a half-warp active), or 8 threads active (a quarter-warp active).
    - The benefit of interleaving the execution of warps (or their parts) is to hide the latency associated with memory access, which can be significantly high.
  - An SM can switch seamlessly between warps (or half- or quarter-warps) as each thread gets its own set of registers.
    - This contradicts the arrangement used by expensive context-switching on CPUs

CUDA'S Execution Model: Streaming Multiprocessors and Warps

- Each SM can have multiple warp schedulers, e.g., in Kepler there are four.
  - This means that up to four independent instruction sequences from four warps can be issued simultaneously
  - Each warp scheduler can issue up to two instructions as long as they are independent leading to instruction-level parallelism (ILP)
    - Independent - the outcome of one does not depend on the outcome of the other

```
1  //independent instructions
2  a = a * b ;
3  d = b + e ;
4  //dependent instructions
5  a = a * b ;
6  d = a + e ; // needs the value of a
```

Block and Grid Design

- Once an SM completes the execution of all the threads in a block, it switches to a different block in the grid
- Assume a kernel that requires 48 registers per thread, and it is to be launched on a GTX 580 card, as a grid of 4x5x3 blocks, each 100 threads long
  - The registers demanded by each block are $100 * 48 = 4800$ (< 32k/SM available on this compute capability 2.0 card)
  - The grid is made of $4 * 5 * 3 = 60$ blocks that need to be distributed to the 16 SMs of the card
    - Assuming round-robin, there will be 12 SMs that will receive four blocks and four SMs that will receive three blocks. During the time the 12 SMs process the last of the 12 blocks they were assigned, the remaining four SMs are idle

SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8	SM9	SM10	SM11	SM12	SM13	SM14	SM15	SM16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60				

Block and Grid Design

- Each of the 100-thread blocks would be split into 4 warps, assuming warpSize=32 (ceil(100/32))
- The first three warps would have 32 threads and the last would have four threads!
- So, during the execution of the last warp of each block, (32−4)/32 = 87.5% of the multiprocessors would be idle
- These issues indicate that kernel design and deployment are critical for extracting the maximum performance from a GPU

Block and Grid Design

- Goal is to keep the available computing hardware "occupied/busy" as much as possible
- For two things must happen:
  - 1. Enough work should be assigned to the CUDA cores (deployment/execution configuration phase)
  - 2. The assigned work should allow execution with the minimum amount of stalling due to resource contention or slow memory access (execution phase)
- Block and grid designs influence the first aspect. The number of dimensions has no effect on the execution efficiency. The total number of threads per block and the total number of blocks in the grid do.

Block and Grid Design

```
2  int deviceCount = 0;
3  cudaGetDeviceCount (&deviceCount ) ;
4  if ( deviceCount == 0 )
5      printf ( "No CUDA compatible GPU exists.\n " ) ;
6  else
7  {
8      cudaDeviceProp pr ;
9      for ( int i =0; i<deviceCount ; i++)
10     {
11         cudaGetDeviceProperties (&pr , i ) ;
12         printf ( " Dev #%i is %s \n " , i , pr.name ) ;
13         printf ( " Dev #%i has %d SMs \n " , i , pr.multiProcessorCount ) ;
14     }
15 }
```

- The multiProcessorCount field can be used to derive the minimum number of blocks a grid should be made of
  - the block number can be a multiple of the SMs

Block and Grid Design

- The number of threads per block can be derived from the warpSize, the maxThreadsPerBlock, and the register and shared memory demands per thread of the kernel
  - The sharedMemPerBlock and regsPerBlock are the fields of the cudaDeviceProp structure
- Calculate an estimate for the number of threads per block

$$threadsPerBlock = min\left(\begin{array}{l} numWarpSchedulers \cdot warpSize, \\ \dfrac{regsPerBlock}{registersPerThread}, \\ \dfrac{sharedMem}{sharedPerThread}, \\ maxThreadsPerSM \end{array}\right)$$

$$threadsPerBlock = warpSize \cdot \left\lceil \frac{threadsPerBlock}{warpSize} \right\rceil$$

$$totalBlocks = \left\lceil \frac{numberOfThreads}{threadsPerBlock} \right\rceil$$

Kernel Structure

```
3   __global__ void foo ( )
4   {
5       int ID = threadIdx.y * blockDim.x + threadIdx.x ;
6       if ( ID % 2 == 0 )
7       {
8           doSmt ( ID ) ;
9       }
10      else
11      {
12          doSmtElse ( ID ) ;
13      }
14      doFinal ( ID ) ;
15  }
```

- A branching operation leads to the stalling of the threads that do not follow the particular branch
- A way around the stalling problem would be to modify the condition so that all the threads in a warp follow the same execution path, but they diversify across warps or blocks

Kernel Structure

```
3   __global__ void foo ( )
4   {
5       int ID = threadIdx.y * blockDim.x + threadIdx.x ;
6       if ( ID % 2 == 0 )
7       {
8           doSmt ( ID ) ;
9       }
10      else
11      {
12          doSmtElse ( ID ) ;
13      }
14      doFinal ( ID ) ;
15  }
```

New id is generated for each thread.

$$ID' = (ID - warpSize \cdot \lceil \frac{warpID}{2} \rceil) \cdot 2 + (warpID\%2) \qquad warpID = \lfloor \frac{ID}{warpSize} \rfloor$$

Threadid	warpId	warpId%2	warpId/2	warpsize*warpid/2	id-	*2	+
1	0	0	0	0	1	2	2
2	0	0	0	0	2	4	4
3	0	0	0	0	3	6	6
4	0	0	0	0	4	8	8
5	0	0	0	0	5	10	10
6	0	0	0	0	6	12	12
7	0	0	0	0	7	14	14
8	0	0	0	0	8	16	16
9	0	0	0	0	9	18	18
10	0	0	0	0	10	20	20
11	0	0	0	0	11	22	22
12	0	0	0	0	12	24	24
13	0	0	0	0	13	26	26
14	0	0	0	0	14	28	28
15	0	0	0	0	15	30	30
16	0	0	0	0	16	32	32
17	0	0	0	0	17	34	34
18	0	0	0	0	18	36	36
19	0	0	0	0	19	38	38
20	0	0	0	0	20	40	40
21	0	0	0	0	21	42	42
22	0	0	0	0	22	44	44
23	0	0	0	0	23	46	46
24	0	0	0	0	24	48	48
25	0	0	0	0	25	50	50
26	0	0	0	0	26	52	52
27	0	0	0	0	27	54	54
28	0	0	0	0	28	56	56
29	0	0	0	0	29	58	58
30	0	0	0	0	30	60	60
31	0	0	0	0	31	62	62
32	1	1	1	32	0	0	1
33	1	1	1	32	1	2	3
34	1	1	1	32	2	4	5
35	1	1	1	32	3	6	7
36	1	1	1	32	4	8	9
37	1	1	1	32	5	10	11
38	1	1	1	32	6	12	13
39	1	1	1	32	7	14	15
40	1	1	1	32	8	16	17
41	1	1	1	32	9	18	19
42	1	1	1	32	10	20	21
43	1	1	1	32	11	22	23
44	1	1	1	32	12	24	25
45	1	1	1	32	13	26	27
46	1	1	1	32	14	28	29
47	1	1	1	32	15	30	31
48	1	1	1	32	16	32	33
49	1	1	1	32	17	34	35
50	1	1	1	32	18	36	37
51	1	1	1	32	19	38	39
52	1	1	1	32	20	40	41
53	1	1	1	32	21	42	43
54	1	1	1	32	22	44	45
55	1	1	1	32	23	46	47
56	1	1	1	32	24	48	49
57	1	1	1	32	25	50	51
58	1	1	1	32	26	52	53
59	1	1	1	32	27	54	55
60	1	1	1	32	28	56	57
61	1	1	1	32	29	58	59

- warpId will be same for all threads in a warp.
- First term is always even as it is multiplied by 2. Second term adds remainder by making it odd or even.

```
3   int warpID = ID / warpSize ;
4   int IDprime = ( ID - ( warpID + 1) / 2 * warpSize ) *2+( warpID % 2) ;
```

Kernel Structure

```
8   __global__ void foo ( )
9   {
10      int ID = threadIdx.y * blockDim.x + threadIdx.x ;
11      int warpID = ID >> offPow ;
12      int IDprime = (( ID - ((( warpID + 1 ) >> 1 ) << offPow ) ) << 1 ) + (warpID & 1) ;
13
14      if ( ( warpID & 1) ==0 ) // modulo-2 condition
15      {
16         doSmt ( IDprime ) ;
17      }
18      else
19      {
20         doSmtElse ( IDprime ) ;
21      }
22      doFinal ( IDprime ) ;
23  }
```

Memory Heirarchy

- GPU memory is disjoint from the host's memory
  - passing chunks of data to a kernel in the form of a pointer to an array in the host's memory is not possible
  - CUDA allows us to copy data from one memory type to another.
  - This includes dereferencing pointers, even in the host's memory (main system RAM)
  - To facilitate this data movement CUDA provides cudaMemcpy()

```
cudaError_t cudaMemcpy ( void *                 dst,
                         const void *           src,
                         size_t                 count,
                         enum cudaMemcpyKind    kind
                       )
```

Parameters:

dst	- Destination memory address
src	- Source memory address
count	- Size in bytes to copy
kind	- Type of transfer

enum cudaMemcpyKind

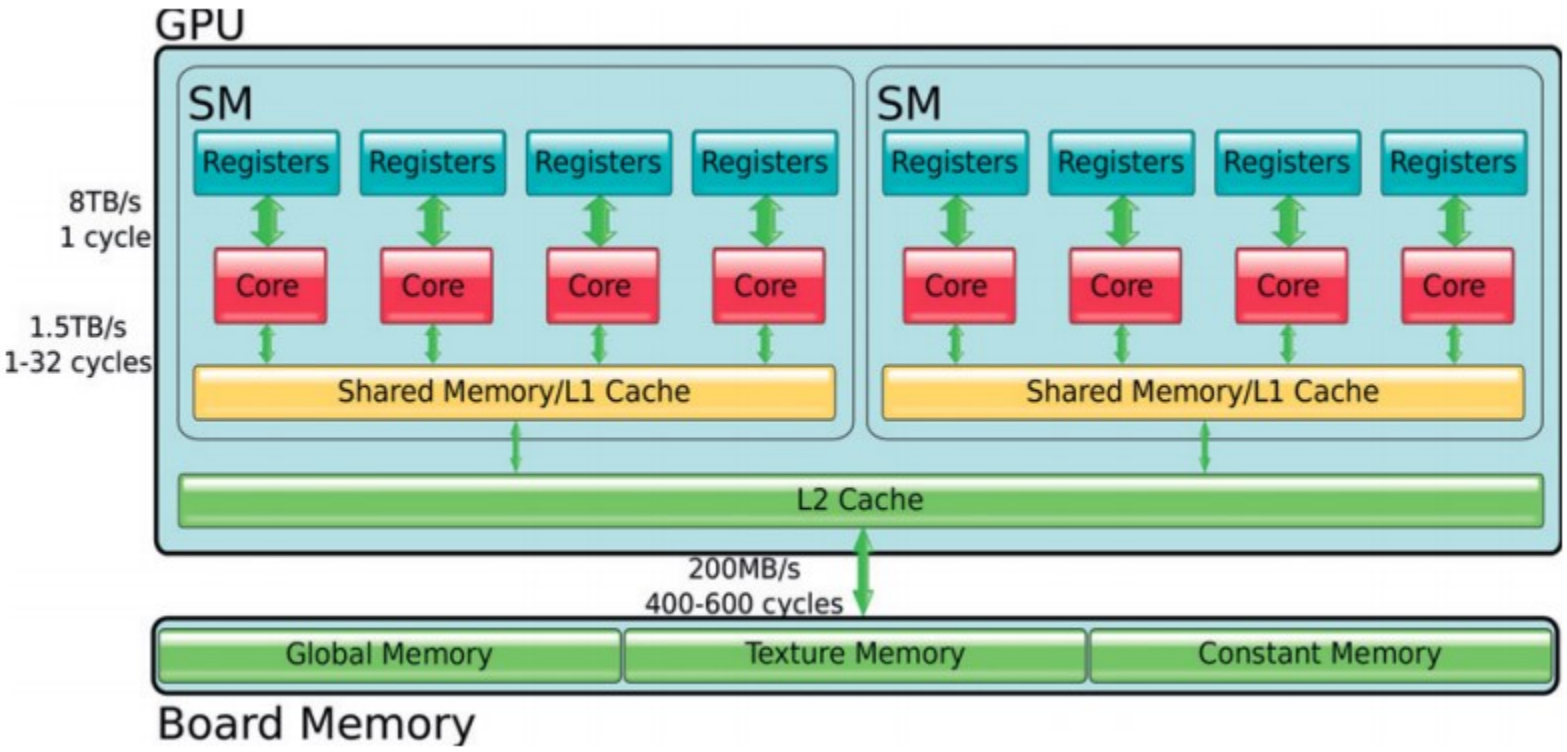
CUDA memory copy types

Enumerator:

cudaMemcpyHostToHost	Host -> Host.
cudaMemcpyHostToDevice	Host -> Device.
cudaMemcpyDeviceToHost	Device -> Host.
cudaMemcpyDeviceToDevice	Device -> Device.

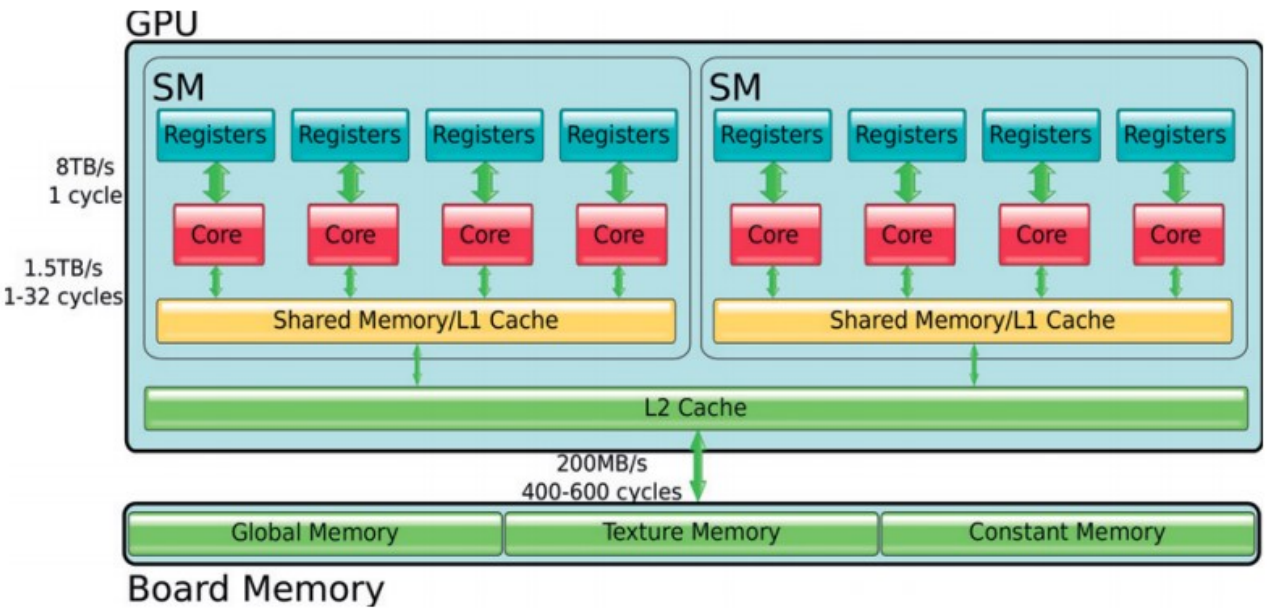
Memory Heirarchy

- The GPU memory hierarchy is not transparent to the programmer
  - GPUs have faster on-chip memory, which has separate address space than the off-chip one



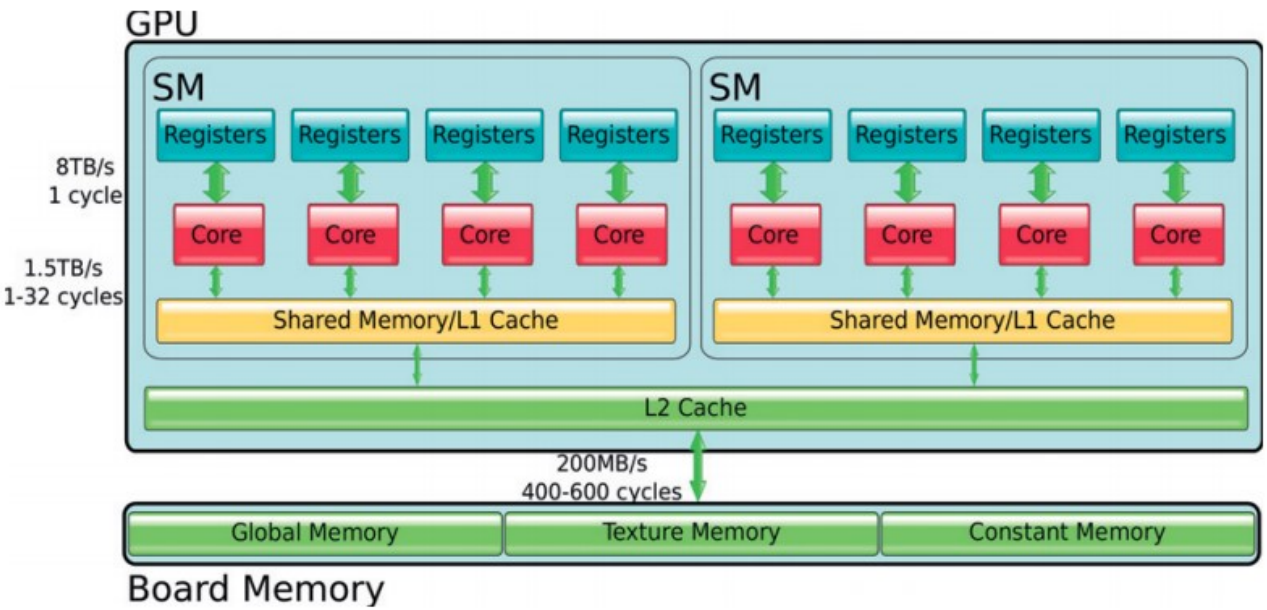
Memory Heirarchy

- Data movement between the host and the device can only involve global memory
- GPUs also employ other types of memory, most of them residing on-chip and in separate address spaces



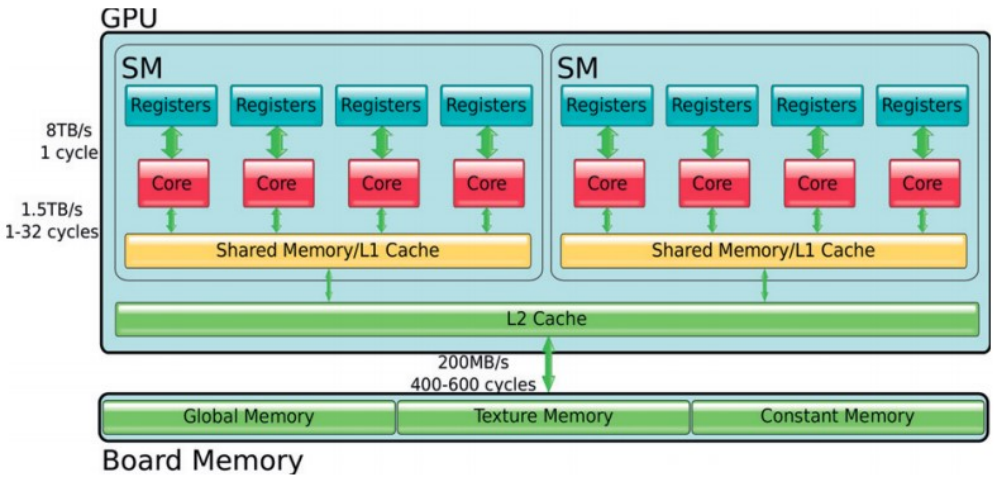
Memory Heirarchy

- Local memory/registers:
  - Used for holding automatic variables.
- Shared memory:
  - Fast on-chip RAM that is used for holding frequently used data. The shared on-chip memory can be used for data exchange between the cores of the same SM.



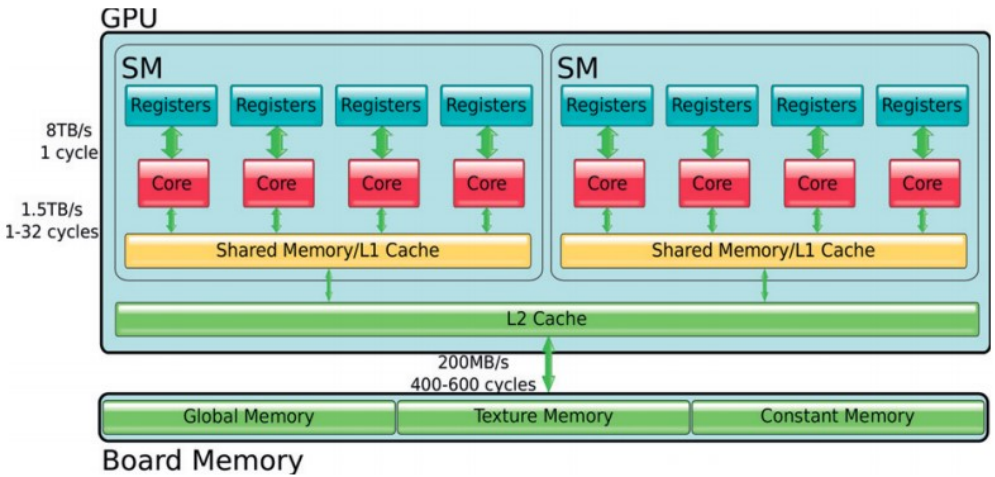
Memory Heirarchy

- Cache memory:
  - Cache memory is transparent to the programmer. In recent GPU generations (e.g., Fermi, Kepler), a fixed amount of fast on-chip RAM is divided between first-level cache (L1) and shared memory. The L2 cache is shared among the SMs
- Global memory:
  - Main part of the off-chip memory. High capacity, but relatively slow. The only part of the memory that is accessible to the host via the CUDA library functions.



Memory Heirarchy

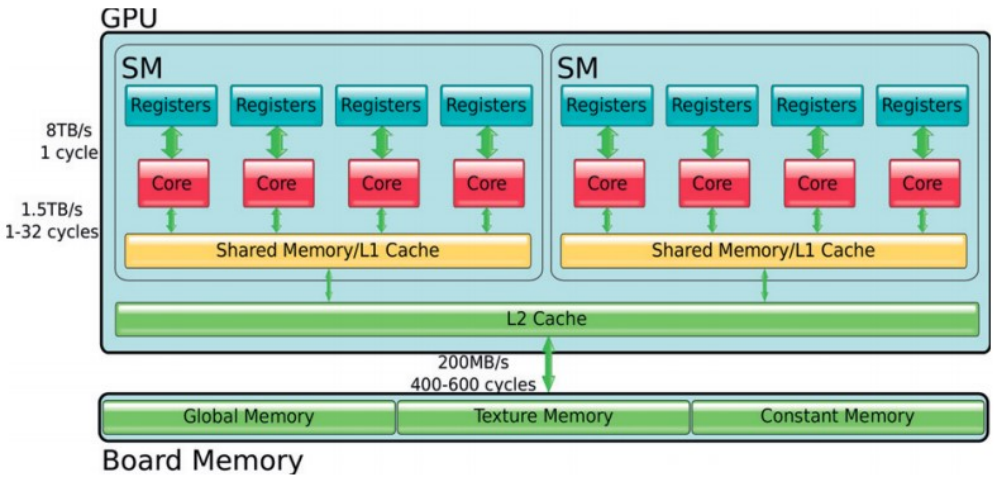
- Texture and surface memory:
  - Part of the off-chip memory. Its contents are handled by special hardware that permits the fast implementation of some filtering operations.
- Constant memory:
  - Part of the off-chip memory. As its name suggests, it is read-only. However, it is cached on-chip, which means it can provide a performance boost.



Memory Heirarchy

- Summary of the memory hierarchy characteristics

Type	Location	Access	Scope	Lifetime
Register	On-chip	R/W	Thread	Thread
Local	Off-chip	R/W	Thread	Thread
Shared	On-chip	R/W	Block	Block
Global	Off-chip	R/W	Grid	Controlled by host
Constant	Off-chip	R	Grid	Controlled by host
Texture	Off-chip	R	Grid	Controlled by host



Local Memory/Registers

- Each multiprocessor gets a set of registers that are split among the resident executing threads
    - used to hold automatic variables declared in a kernel, speeding up operations that would otherwise require access to the global or shared memories
- The number of registers used per thread influences the maximum number of threads that can be resident at an SM
    - For example, assume that a kernel is using 48 registers and it is invoked as blocks of 256 threads, which means each block requires $48 \cdot 256 = 12{,}288$ registers.

Local Memory/Registers

- For example, assume that a kernel is using 48 registers and it is invoked as blocks of 256 threads, which means each block requires $48 \cdot 256 = 12{,}288$ registers.
  - If the target GPU for running the kernel is a GTX 580, sporting 32k registers per SM, then each SM could have only two resident blocks (requiring $2 \cdot 12{,}288 = 24{,}576$ registers) as three would exceed the available register space ($3 \cdot 12{,}288 = 36{,}864 > 32{,}768$). This in turn means that each SM could have $2 \cdot 256 = 512$ resident threads running, which is well below the maximum limit of 1536 threads per SM. This undermines the GPU's capability to hide the latency of memory operations by running other ready warps.
- Nvidia calls occupancy the ratio of resident warps over the maximum possible resident warps

$$occupancy = \frac{resident\_warps}{maximum\_warps}$$

Local Memory/Registers

- In this example occupancy =

$$occupancy = \frac{resident\_warps}{maximum\_warps}$$

$$\frac{2 \cdot \frac{256 \text{ threads}}{32 \text{ threads/warp}}}{48 \text{ warps}} = \frac{16}{48} = 33.3\%$$

  - an occupancy close to 1 is desirable
- In order to raise the occupancy in our example, we could (a) reduce the number of required registers by the kernel, or (b) use a GPU with a bigger register file than GTX 580, such as GTX 680.
  - If the required registers per kernel fell to 40, then we could have three resident blocks (requiring a total of $3 \cdot 40 \cdot 256 = 30{,}720$ registers), resulting in an occupancy of 3·8 48 = 50% because each block is made up of eight warps
  - If a GTX 680 were used, the resident blocks would go up to five, resulting in an occupancy of 5·8 64 = 63%.

Shared Memory

- Shared memory is a block of fast on-chip RAM that is shared among the cores of an SM
- Each SM gets its own block of shared memory, which can be viewed as a user-managed L1 cache
- Shared memory can be used in the following capacities:
  - As a holding place for very frequently used data that would otherwise require global memory access
  - As a fast mirror of data that reside in global memory, if they are to be accessed multiple times
  - As a fast way for cores within an SM, to share data

Shared Memory

- How can we specify that the holding place of some data will be the shared memory of the SM and not the global memory of the device?
  - The answer is, via the __shared__ specifier

```
// histogramGPU computes the histogram of an input array on the
GPU
__global__ void histogramGPU(unsigned int* input,
       unsigned int* bins, unsigned int numElems) {
    int tx = threadIdx.x; int bx = blockIdx.x;

    // compute global thread coordinates
    int i = (bx * blockDim.x) + tx;

    // create a private histogram copy for each thread block
    __shared__ unsigned int hist[4096];
```

Shared Memory

- To speed up the update of the local counts, a “local” array is set up in shared memory. Because multiple threads may access its locations at any time, atomic addition operations have to be used for modifying its contents
  - The atomicAdd is an overloaded function that belongs to a set of atomic operations

```
// update private histogram
    if (i < numElems) {
        atomicAdd(&(hist[input[i]]), 1);
    }
    // wait for all threads in the block to finish
    __syncthreads();
```

- Because threads execute in blocks and each block executes warp by warp, explicit synchronization of the threads must take place between the discrete phases of the kernel, e.g., between initializing the shared-memory histogram array and starting to calculate the histogram, and so on
  - the __syncthreads() function can be called inside a kernel to act as a barrier for all the threads in a block

CUDA Streams

- In a CUDA program, first the data must be transferred from the CPU memory into the GPU memory. When the kernel execution is done, the processed data must be transferred back to CPU memory.
  - Data transfer rate between CPU and GPU is limited by PICe.
  - For, e.g. CPU→GPU takes 31% of the total time, kernel execution in GPU takes 39%, GPU→CPU takes 30% of total time.
- Can we start processing in GPU as soon as data starts arriving into GPU?
  - Yes. CPU←→GPU transfers can be overlapped with kernel execution because they differet hardware (PCI and GPU cores)

Asynchronous data transfers

- To get best overlap opportunities between data copy and kernel activity, it's necessary to
  - Host memory buffers that are pinned, e.g. via cudaMallocHost(void **, SIZE)
  - Usage of cudaMemcpyAsync (instead of cudaMemcpy)
- Asynchronous data transfers and pinned memory form the skeleton of CUDA streams
- To stream operations
  - First create pinned memory
    - needed for performing Direct Memory Access (DMA) transfers across the PCIe bus
  - Create multiple streams and partition task into multiple subtasks that can executed independently.
  - Assign each subtask into different stream

CUDA Streams

```
cudaMemcpyAsync ( void* dst, const void* src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0 )
/*Copies data between host and device.*/
```

- There is default stream in CUDA programs. Additional streams can be created.

```
cudaStream_t stream[MAXSTREAMS];
for(i=0;i<NumberOfStreams;i++){
    cudaStreamCreate(&stream[i]);
}
```

- There is one copy engine in each stream. By calling cudaMemcpyAsync copy operations are queued on copy engine of the streams which are executed whenever PCIe bus is available
- Execution moves onto the next line.
  - In the next line, execute a kernel or queue up another transfer on another stream.

CUDA Streams

- There is one kernel execution engine in each stream. Its job is to queue up the execution of different kernels for a given stream

```
10    additionKernel <<<g,b,0,stream[i]);
11
12  /*
13    Kernel_Name<<< GridSize,  BlockSize,  SMEMSize,   Stream  >>>(arg,..);
14    SMEMsize  : is the size of Shared Memory at Runtime .
15    Stream        : is a stream on which kernel will execute.
16    */
```

```
17  int main ()
18  {
19    cudaStream_t str[2];
20    int *h_data[2], *d_data[2];
21    int i;
22
23    for(i=0;i<2;i++)
24    {
25      cudaMallocHost((void **) &(h_data[i]), sizeof(int) * DATASIZE);
26      cudaMalloc((void ** )&(d_data[i]), sizeof(int) * DATASIZE);
27    }
28      // inititalize h_data[i]....
29
30    for(i=0;i<2;i++)
31    {
32      cudaStreamCreate(&(str[i]));
33      cudaMemcpyAsync(d_data[i], h_data[i], sizeof(int) * DATASIZE, cudaMemcpyHostToDevice, str[i]);
34
35      doSmt <<< 10, 256, 0, str[i] >>> (d_data[i]);
36
37      cudaMemcpyAsync(h_data[i], d_data[i], sizeof(int) * DATASIZE, cudaMemcpyDeviceToHost, str[i]);
38    }
39
40    cudaStreamSynchronize(str[0]);
41    cudaStreamSynchronize(str[1]);
42    cudaStreamDestroy(str[0]);
43    cudaStreamDestroy(str[1]);
44
45    for(i=0;i<2;i++)
46    {
47      cudaFree(h_data[i]);
48      cudaFree(d_data[i]);
49    }
50    cudaDeviceReset();
51    return 1;
52  }
```

```
6   __global__ void doSmt(int *data)
7   {
8     // Simplification of above
9     int myID = ( blockIdx.z * gridDim.x * gridDim.y  +
10                blockIdx.y * gridDim.x +
11                blockIdx.x ) * blockDim.x +
12                threadIdx.x;
13
14    printf ("Hello world from %i\n", myID);
15  }
```

Synchronizing CUDA Streams

```
9     cudaStreamSynchronize(stream[i]);
10
```

- For given stream, this function will block until all the queued up operations are complete

- Events
  - The time instance a command (and everything preceding it in a stream) completes can be captured in the form of an event. CUDA uses the cudaEvent_t type for managing events

```
cudaEventRecord (
cudaEvent_t event , // Event identifier (IN)
cudaStream_t stream =0) ; // Stream identifier( IN )
```

```
cudaEventElapsedTime (
float ms , // Storage for elapsed time in msec units (OUT)
cudaEvent_t start , // S t a r t event i d e n t i f i e r ( IN )
cudaEvent_t end ) ; // End event i d e n t i f i e r ( IN )
```

Events

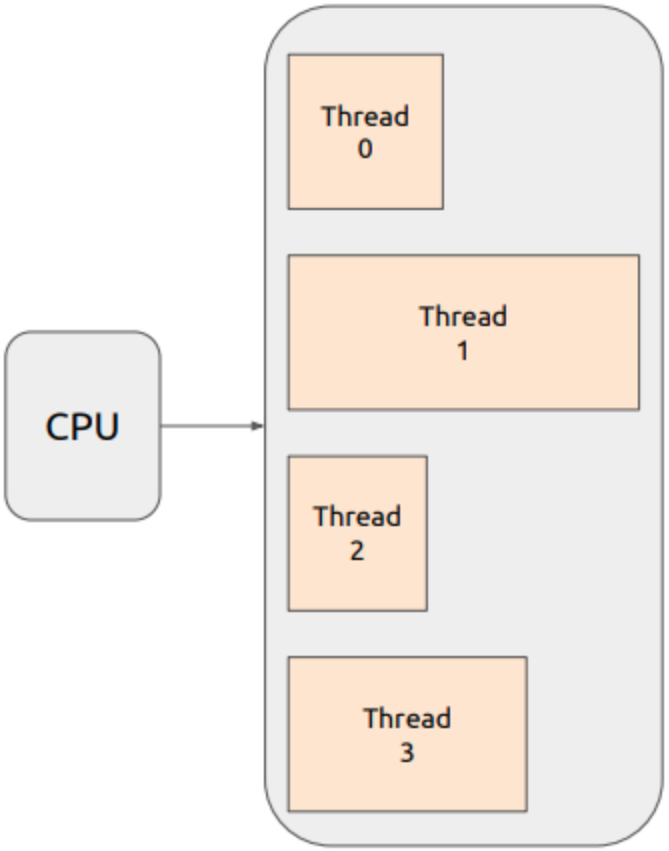
```
16  int main (int argc, char **argv)
17  {
18    int iter = atoi (argv[1]);
19    int step = atoi (argv[2]);
20    cudaStream_t str;
21    int *h_data, *d_data;
22    int i, dataSize;;
23    cudaEvent_t startT, endT;
24    float duration;
25
26    cudaMallocHost ((void **) &h_data, sizeof (int) * MAXDATASIZE);
27    cudaMalloc ((void **) &d_data, sizeof (int) * MAXDATASIZE);
28    for (i = 0; i < MAXDATASIZE; i++)
29      h_data[i] = i;
30
31    cudaEventCreate (&startT);
32    cudaEventCreate (&endT);
33    cudaStreamCreate (&str);
34
35    for (dataSize = 0; dataSize <= MAXDATASIZE; dataSize += step)
36      {
37        cudaEventRecord (startT, str);
38        for (i = 0; i < iter; i++)
39          {
40            cudaMemcpyAsync (d_data, h_data, sizeof (int) * dataSize, cudaMemcpyHostToDevice, str);
41          }
42        cudaEventRecord (endT, str);
43        cudaEventSynchronize (endT);
44        cudaEventElapsedTime (&duration, startT, endT);
45        printf ("%i %f\n", (int) (dataSize * sizeof (int)), duration / iter);
46      }
47
48    cudaStreamDestroy (str);
49    cudaEventDestroy (startT);
50    cudaEventDestroy (endT);
51
52    cudaFreeHost (h_data);
53    cudaFree (d_data);
54    cudaDeviceReset ();
```

CUDA Dynamic Parallelism

- An extension to the CUDA programming model which allows a thread to launch another grid of threads executing another kernel
- is supported in devices of Compute Capability 3.5 and above
- Uses for Dynamic Parallelism
  - Recursive algorithms
  - Processing at different levels of detail for different parts of the input (i.e. irregular grid structure)
  - Algorithms in which new work is “uncovered” along the way

Work Discovery With/Without Dynamic Parallelism

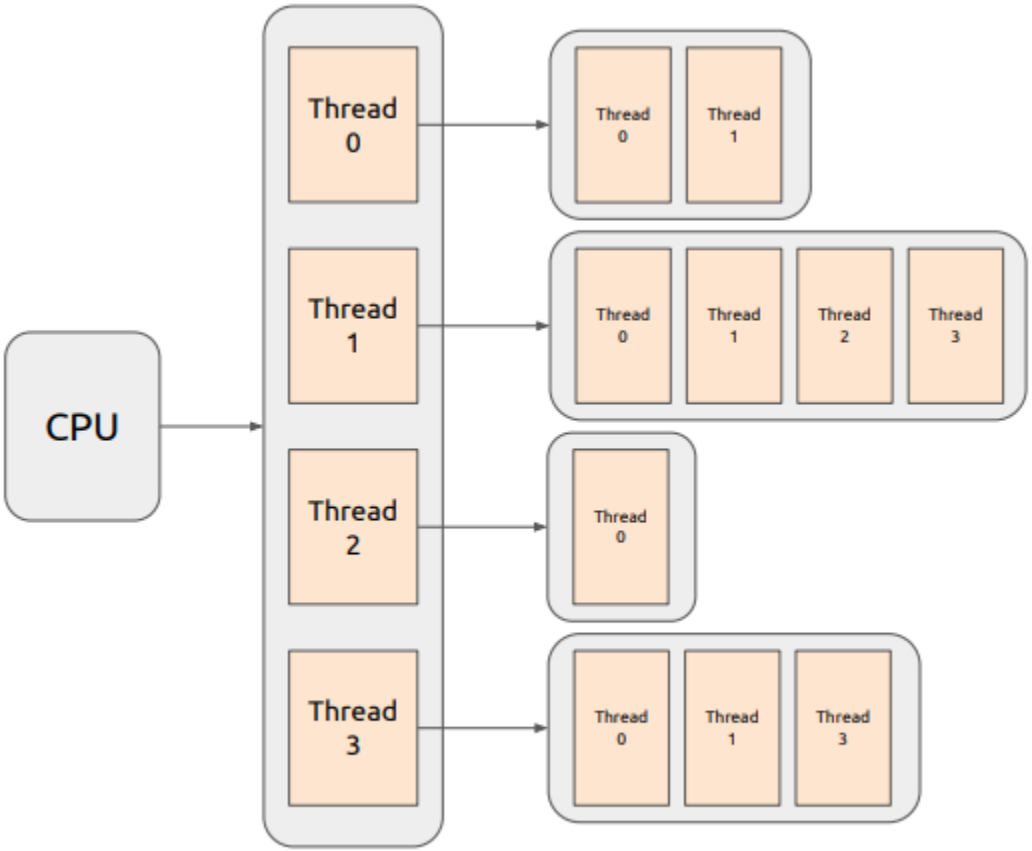
```
/*Without dynamic parallelism*/
__global__ void workDiscoveryKernel(const int * starts, const int * ends, float * data) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    for (int j = starts[i]; j < ends[i]; ++j) {
        process(data[j]);
    }
}
```



Without dynamic parallelism

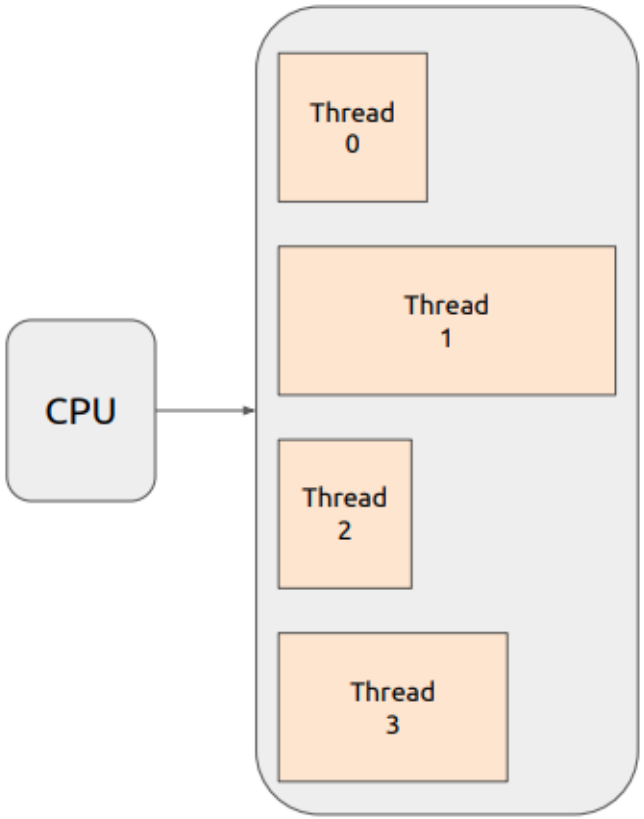
Work Discovery With/Without Dynamic Parallelism

```
10  /*With dynamic parallelism*/
11  __global__ void workDiscoveryKernel(const int * starts, const int * ends, float * data) {
12      int i = threadIdx.x + blockDim.x * blockIdx.x;
13      const int N = ends[i] - starts[i];
14      workDiscoveryChildKernel<<<(N-1)/128+1,128>>>(data + starts[i], N);
15  }
16  __global__ void workDiscoveryChildKernel(float * data, const int N) {
17      int j = threadIdx.x + blockDim.x * blockIdx.x;
18      if (j < N) {
19          process(data[j]);
20      }
21  }
```

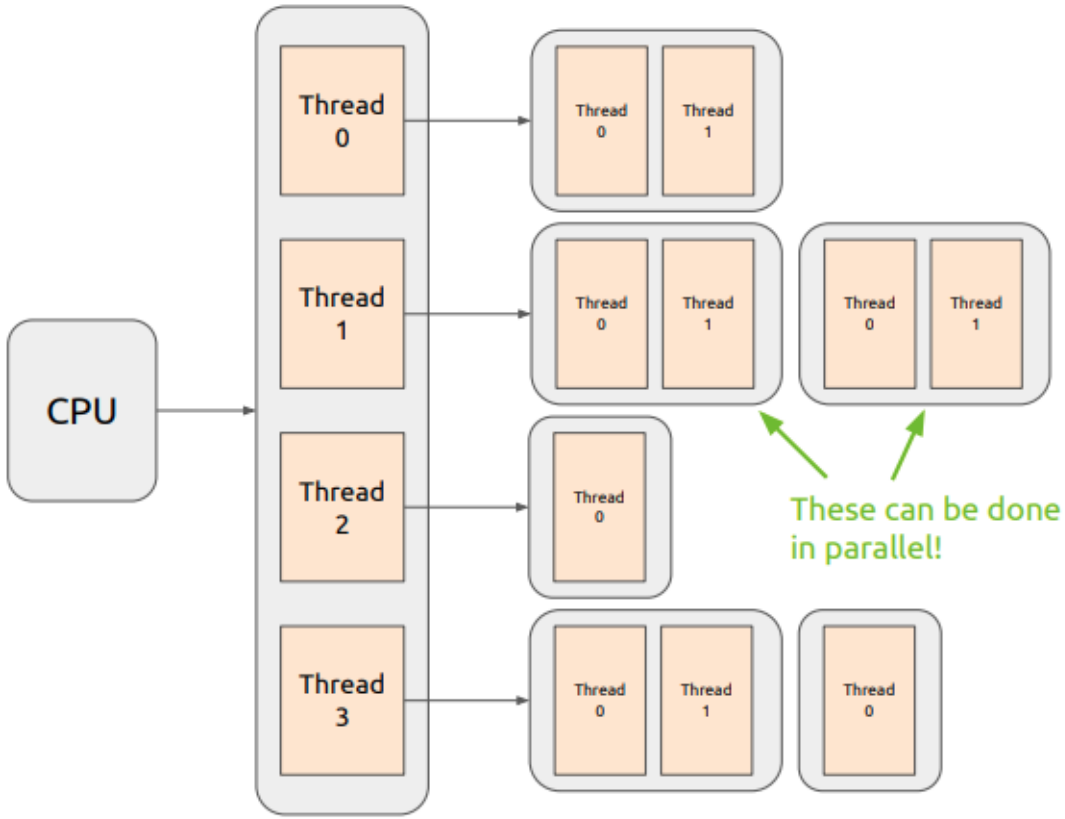


With dynamic parallelism

Work Discovery With/Without Dynamic Parallelism



Without dynamic parallelism



With dynamic parallelism

Parent grid & Child grid

- A grid that is launched by a CUDA thread is considered a child grid
  - The grid of the launcher is called the parent grid
- Child grids execute asynchronously, as if they were launched by the host
  - a parent grid is not considered finished, until all the child grids launched by its threads are also finished.
- Parent and child grids have two points of guaranteed global memory consistency:
  - When the child grid is launched by the parent; all memory operations performed by the parent thread before launching the child are visible to the child grid when it starts
  - When the child grid finishes; all memory operations by any thread in the child grid are visible to the parent thread once the parent thread has synchronized with the completed child grid

Parent grid & Child grid

- A kernel launched from within a kernel can launch a kernel, which can also launch a kernel, etc
- The total “nesting depth” allowed with dynamic parallelism is limited to 24
- Kernels launched from within a kernel cannot be executed on another GPU
- Memory management
  - a child grid launch can be passed references to global data, but not to shared memory, since shared memory is private to a specific SM
  - the same applies to local memory, as it is private to a thread.

QuickSort

The child grids are launched using two separate streams. Launching them into the default stream would force their sequential execution.

```
93   __global__ void QSort (int *data, int N, int depth)
94   {
95     if(depth == MAXRECURSIONDEPTH)
96     {
97       insertionSort(data, N);
98       return ;
99     }
100
101    if (N <= 1)
102      return;
103
104    // break the data into a left and right part
105    int pivotPos = partition (data, N);
106
107    cudaStream_t s0, s1;
108    // sort the left part if it exists
109    if (pivotPos > 0)
110      {
111        cudaStreamCreateWithFlags (&s0, cudaStreamNonBlocking);
112        QSort <<< 1, 1, 0, s0 >>> (data, pivotPos, depth+1);
113        cudaStreamDestroy (s0);
114      }
115
116    // sort the right part if it exists
117    if (pivotPos < N - 1)
118      {
119        cudaStreamCreateWithFlags (&s1, cudaStreamNonBlocking);
120        QSort <<< 1, 1, 0, s1 >>> (&(data[pivotPos + 1]), N - pivotPos - 1, depth+1);
121        cudaStreamDestroy (s1);
122      }
123  }
```

```
126  int main (int argc, char *argv[])
127  {
128    if (argc == 1)
129      {
130        fprintf (stderr, "%s N\n", argv[0]);
131        exit (0);
132      }
133    int N = atoi (argv[1]);
134    int *data;
135    cudaMallocManaged ((void **) &data, N * sizeof (int));
136
137    numberGen (N, 1000, data);
138
139    QSort <<< 1, 1 >>> (data, N, 0);
140
141    cudaDeviceSynchronize ();
142
143    dump (data, N);
144
145    // clean-up allocated memory
146    cudaFree (data);
147    return 0;
148  }
```

```
46   __device__ int partition (int *data, int N)
47   {
48     int i = 0, j = N;
49     int pivot = data[0];
50
51     do
52       {
53         do
54           {
55             i++;
56           }
57         while (pivot > data[i] && i < N);
58
59         do
60           {
61             j--;
62           }
63         while (pivot < data[j] && j > 0);
64         swap (data, i, j);
65       }
66     while (i < j);
67     // undo last swap
68     swap (data, i, j);
69
70     // fix the pivot element position
71     swap (data, 0, j);
72     return j;
73   }
```

Multi-GPU Programming with CUDA

- A single host thread can manage multiple devices
- In general, the first step is determining the number of CUDA-enabled devices available in a system with cudaGetDeviceCount()

```
2   #include <stdio.h>
3   #include <cuda.h>
4
5   int main ()
6   {
7     int deviceCount = 0;
8     cudaGetDeviceCount (&deviceCount);
9     if (deviceCount == 0)
10      printf ("No CUDA compatible GPU.\n");
11    else
12      {
13        cudaDeviceProp pr;
14        for (int i = 0; i < deviceCount; i++)
15          {
16            cudaGetDeviceProperties (&pr, i);
17            printf ("Dev #%i is %s\n", i, pr.name);
18          }
19      }
20    return 1;
21  }
```

Multi-GPU Programming with CUDA

- We select which GPU is the current target for all CUDA operations with cudaSetDevice()
  - This function sets the device with identifier id as the current device. device identifiers range from 0 to deviceCount-1
  - Current GPU can be changed while async calls (kernels, memcpy) are running

```
24    cudaSetDevice(0);
25    kernel<<<...>>>(...);
26    cudaMemcpyAsync(...);
27    cudaSetDevice(1);
28    kernel<<<...>>>(...);
```

Multi-GPU Programming with CUDA

1. Select the set of GPUs this application will use
2. Create streams for each device
3. Allocate device resources on each device (for example, device memory)
4. Launch tasks on each GPU through the stream (for example, data transfers or kernel executions)
5. Use the streams to wait for task completion

Multi-GPU Programming with CUDA

```
7   int deviceCount = 0;
8   cudaGetDeviceCount (&deviceCount);
9   if (deviceCount == 0)
10    printf ("No CUDA compatible GPU.\n");
11  else
12    {
13      for (int i = 0; i < deviceCount; i++)
14        {
15            cudaSetDevice(i);
16           cudaStreamCreate(&(str[i]));
17            h_data[i] = (int *)malloc(sizeof(int) * DATASIZE);
18            cudaMalloc((void ** )&(d_data[i]), sizeof(int) * DATASIZE);
19
20            // inititalize h_data[i]....
21
22            cudaMemcpyAsync(d_data[i], h_data[i], sizeof(int) * DATASIZE, cudaMemcpyHostToDevice, str[i]);
23
24            doSmt <<< 10, 256, 0, str[i] >>> (d_data[i]);
25
26            cudaMemcpyAsync(h_data[i], d_data[i], sizeof(int) * DATASIZE, cudaMemcpyDeviceToHost, str[i]);
27        }
28    }
```

Peer-to-peer transfer

- Many modern GPU systems are able to directly (without involving the host) transfer data
  - This is called peer-to-peer transfer.
- Data transfer speeds
  - Main memory: 900 GB/s
  - NVLink device-to-device: 300 GB/s
  - PCIe 3.0 16x: 16 GB/s

Peer-to-peer transfer

- Because not all GPUs support peer-to-peer access, we need to check if a device supports P2P using cudaDeviceCanAccessPeer()
- Peer-to-peer memory access must be explicitly enabled between two devices with cudaDeviceEnablePeerAccess()
- This function enables peer-to-peer access from the current device to peerDevice.
  - The flag argument is reserved for future use and currently must be set to 0.
  - The access granted by this function is unidirectional
  - this function enables access from the current device
  - to peerDevice but does not enable access from peerDevice.

```
3    int is_able;
4    cudaSetDevice(i);
5    cudaDeviceCanAccessPeer(&is_able, i, j);
6    if(is_able)
7    cudaDeviceEnablePeerAccess(j, 0);
```

Peer-to-peer transfer

- After enabling peer access between two devices, we can copy data between those devices asynchronously with cudaMemcpyPeerAsync()
- This transfers data from device memory on the device srcDev to device memory on the device dstDev. The function cudaMemcpyPeerAsync is asynchronous with respect to the host and all other devices.

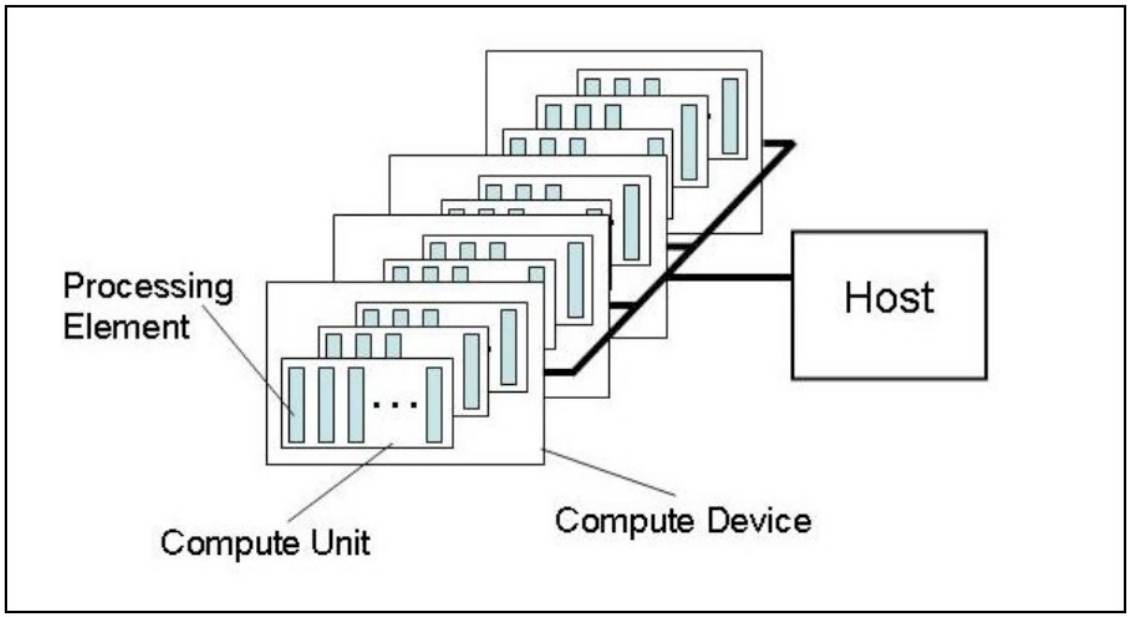
```
cudaMemcpyPeerAsync(dst, dst_device_id, src, src_device_id, size, stream[i]);
```

OpenCL

- OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of
  - central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators.

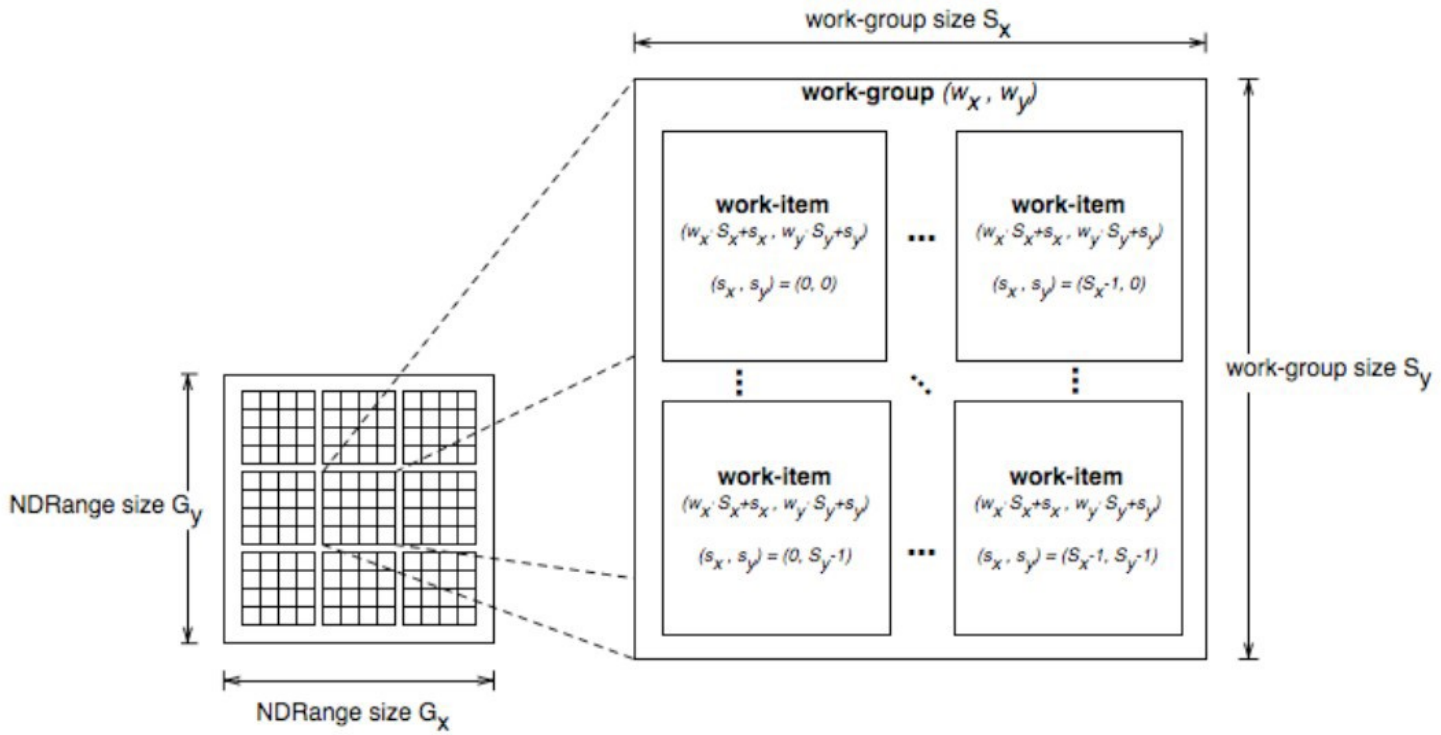
OpenCL

- OpenCL views a computing system as consisting of a number of compute devices, which might be central processing units (CPUs) or "accelerators" such as graphics processing units (GPUs), attached to a host processor (a CPU)
  - It defines a C-like language for writing programs. Functions executed on an OpenCL device are called "kernels".



OpenCL to CUDA Data Parallelism Model Mapping

OpenCL Parallelism Concept	CUDA Equivalent
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

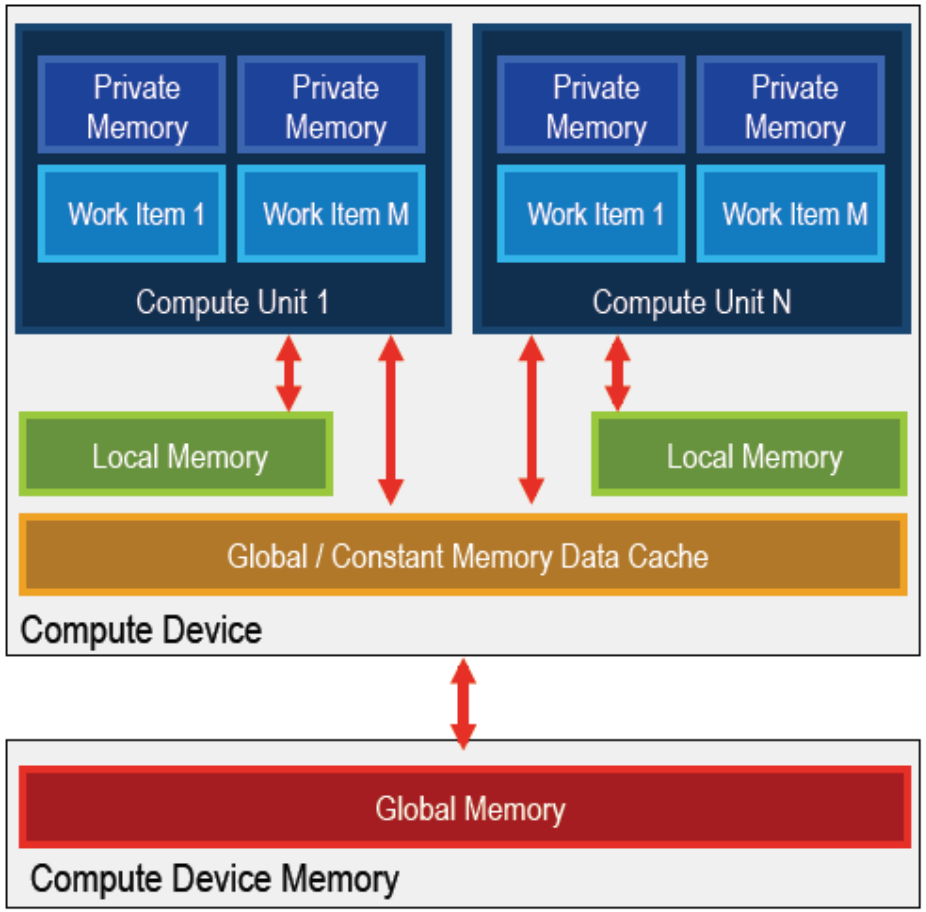


Mapping of OpenCL Dimensions and Indices to CUDA

OpenCL API Call	Explanation	CUDA Equivalent
get_global_id(0);	global index of the work item in the x dimension	blockIdx.x×blockDim.x+threadIdx.x
get_local_id(0)	local index of the work item within the work group in the x dimension	blockIdx.x
get_global_size(0);	size of NDRange in the x dimension	gridDim.x ×blockDim.x
get_local_size(0);	Size of each work group in the x dimension	blockDim.x

Mapping OpenCL Memory Types to CUDA

OpenCL Memory Types	CUDA Equivalent
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	Local memory



A Simple OpenCL Kernel Example

```
1. __kernel void  vadd(__global const float *a,
2.     __global const float *b, __global float *result) {

3.     int id = get_global_id(0);
4.     result[id] = a[id] + b[id];
5. }
```

A Simple OpenCL Kernel Example

```
1.  OpenCL:
2.  __kernel void clenergy( …) {

3.  unsigned int xindex=  (get_global_id(0) / get_local_id(0))*
    UNROLLX + get_local_id(0) ;
4.  unsigned int yindex= get_global_id(1);
5.  unsigned int outaddr= get_global_size(0) * UNROLLX
    *yindex+xindex;


6.  CUDA:
7.  __global__ void cuenergy(…) {
8.  Unsigned int xindex= blockIdx.x *blockDim.x +threadIdx.x;
9.  unsigned int yindex= blockIdx.y *blockDim.y +threadIdx.y;
10. unsigned int outaddr= gridDim.x *blockDim.x *
    UNROLLX*yindex+xindex
```

```c
#include <stdio.h>
#include <stdlib.h>
#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif
#define VECTOR_SIZE 1024
//OpenCL kernel which is run for every work item
created.
const char *saxpy_kernel =
"__kernel                                   \n"
"void saxpy_kernel(float alpha,     \n"
"                  __global float *A,       \n"
"                  __global float *B,       \n"
"                  __global float *C)       \n"
"{                                          \n"
"    //Get the index of the work-item       \n"
"    int index = get_global_id(0);          \n"
"    C[index] = alpha* A[index] + B[index]; \n"
"}                                          \n";
int main(void) {
  int i;
  // Allocate space for vectors A, B and C
  float alpha = 2.0;
  float *A = (float*)malloc(sizeof(float)*VECTOR_SIZE);
  float *B = (float*)malloc(sizeof(float)*VECTOR_SIZE);
  float *C = (float*)malloc(sizeof(float)*VECTOR_SIZE);
  for(i = 0; i < VECTOR_SIZE; i++)
  {
    A[i] = i;    B[i] = VECTOR_SIZE - i;    C[i] = 0;
  }
  // Get platform and device information
  cl_platform_id * platforms = NULL;
  cl_uint     num_platforms;
  //Set up the Platform
  cl_int clStatus = clGetPlatformIDs(0, NULL,
&num_platforms);
  platforms = (cl_platform_id *)
  malloc(sizeof(cl_platform_id)*num_platforms);
  clStatus = clGetPlatformIDs(num_platforms, platforms,
NULL);
```

```c
//Get the devices list and choose the device you want to run
on
  cl_device_id     *device_list = NULL;
  cl_uint           num_devices;

  clStatus = clGetDeviceIDs( platforms[0],
CL_DEVICE_TYPE_GPU, 0,NULL, &num_devices);
  device_list = (cl_device_id *)
  malloc(sizeof(cl_device_id)*num_devices);
  clStatus = clGetDeviceIDs(
platforms[0],CL_DEVICE_TYPE_GPU, num_devices, device_list,
NULL);

  // Create one OpenCL context for each device in the
platform
  cl_context context;
  context = clCreateContext( NULL, num_devices, device_list,
NULL, NULL, &clStatus);

  // Create a command queue
  cl_command_queue command_queue =
clCreateCommandQueue(context, device_list[0], 0, &clStatus);

  // Create memory buffers on the device for each vector
  cl_mem A_clmem = clCreateBuffer(context,
CL_MEM_READ_ONLY,VECTOR_SIZE * sizeof(float), NULL,
&clStatus);
  cl_mem B_clmem = clCreateBuffer(context,
CL_MEM_READ_ONLY,VECTOR_SIZE * sizeof(float), NULL,
&clStatus);
  cl_mem C_clmem = clCreateBuffer(context,
CL_MEM_WRITE_ONLY,VECTOR_SIZE * sizeof(float), NULL,
&clStatus);

  // Copy the Buffer A and B to the device
  clStatus = clEnqueueWriteBuffer(command_queue, A_clmem,
CL_TRUE, 0, VECTOR_SIZE * sizeof(float), A, 0, NULL, NULL);
  clStatus = clEnqueueWriteBuffer(command_queue, B_clmem,
CL_TRUE, 0, VECTOR_SIZE * sizeof(float), B, 0, NULL, NULL);
```

```
// Create a program from the kernel source
  cl_program program =
clCreateProgramWithSource(context, 1,(const
char **)&saxpy_kernel, NULL, &clStatus);

  // Build the program
  clStatus = clBuildProgram(program, 1,
device_list, NULL, NULL, NULL);

  // Create the OpenCL kernel
  cl_kernel kernel = clCreateKernel(program,
"saxpy_kernel", &clStatus);

  // Set the arguments of the kernel
  clStatus = clSetKernelArg(kernel, 0,
sizeof(float), (void *)&alpha);
  clStatus = clSetKernelArg(kernel, 1,
sizeof(cl_mem), (void *)&A_clmem);
  clStatus = clSetKernelArg(kernel, 2,
sizeof(cl_mem), (void *)&B_clmem);
  clStatus = clSetKernelArg(kernel, 3,
sizeof(cl_mem), (void *)&C_clmem);

  // Execute the OpenCL kernel on the list
  size_t global_size = VECTOR_SIZE; //
Process the entire lists
  size_t local_size = 64;           //
Process one item at a time
  clStatus =
clEnqueueNDRangeKernel(command_queue,
kernel, 1, NULL, &global_size, &local_size,
0, NULL, NULL);
```

```
  // Read the cl memory C_clmem on device to the host
variable C
  clStatus = clEnqueueReadBuffer(command_queue, C_clmem,
CL_TRUE, 0, VECTOR_SIZE * sizeof(float), C, 0, NULL, NULL);

  // Clean up and wait for all the comands to complete.
  clStatus = clFlush(command_queue);
  clStatus = clFinish(command_queue);

  // Display the result to the screen
  for(i = 0; i < VECTOR_SIZE; i++)
    printf("%f * %f + %f = %f\n", alpha, A[i], B[i], C[i]);

  // Finally release all OpenCL allocated objects and host
buffers.
  clStatus = clReleaseKernel(kernel);
  clStatus = clReleaseProgram(program);
  clStatus = clReleaseMemObject(A_clmem);
  clStatus = clReleaseMemObject(B_clmem);
  clStatus = clReleaseMemObject(C_clmem);
  clStatus = clReleaseCommandQueue(command_queue);
  clStatus = clReleaseContext(context);
  free(A);
  free(B);
  free(C);
  free(platforms);
  free(device_list);
  return 0;
}
```

References

- Chapter 6, Multicore and GPU Programming An Integrated Approach, Gerassimos Barlas, Morgan Kaufmann

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You