Introduction to CUDA
CIRC Summer School 2014

Baowei Liu

Center of Integrated Research Computing
University of Rochester

October 20, 2014

What will you learn on this class?

- Start from "Hello World!"
- Write and execute Parallel code on the GPU
- Basic concepts and technologies witu CUDA programming
- Hands-experience on BlueHive with Labs

Prerequisites

- You (probably) need experience with C or C++
- You don't need GPU experience
- You don't need parallel programming experience
- You don't need graphics experience

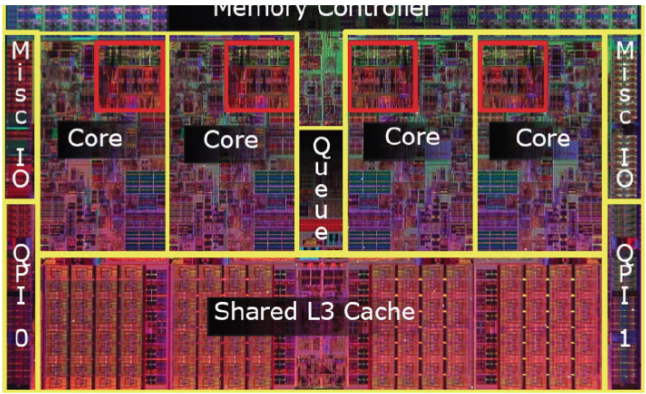
Multicore vs. Manycore

Thursday, June 26, 2014

BlueHive specifications:

- 178 compute nodes each with
  - 2 Intel Xeon E5-2695 v2 processors with 12 cores each
  - Between 64 and 512 GB of physical memory
- FDR10 Infiniband interconnect
- 2 PB of raw storage
- 18 TB of physical memory
- 60 NVidia Tesla K20X cards each with
  - 6 GB of physical memory
  - 2688 cores @ 732 MHz
- 16 Intel 5110P Phi coprocessors each with
  - 60 cores @ 1.053 GHz
  - 8 GB physical memory
  - 30 MB Cache
- 4 Hadoop nodes with a total of 448 TB of local storage

Multicore vs. Manycore



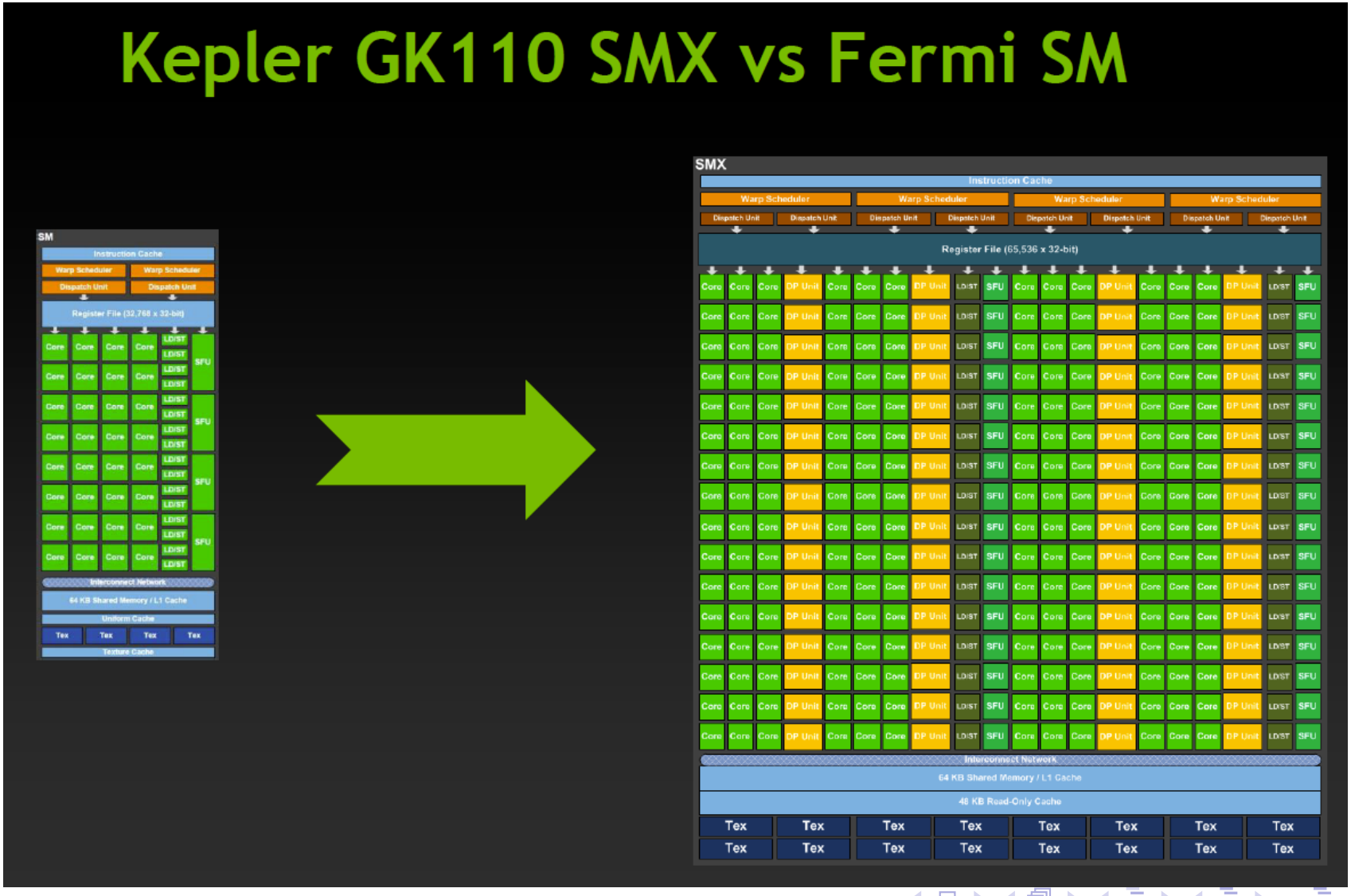
(a) Intel's Xeon processor includes 12 CPU cores @ 2.4 GHz with simultaneous multithreading, 30MB of L3 cache, and on-chip DRAM controllers. Made with 22 nm process Technology.

(b) The Tesla K20 GPU includes 2688 CUDA cores @ 732 MHz, 1.5MB of L2 cache.Made with 28 nm process Technology.

Streaming Multiprocessors(SMs)

- Perform the actual computations
- Each SM has its own control units, registers, execution pipelines, caches
- Each Tesla Kepler GPU has 15 SMX Units.
- Each SMX has 192 single-precision CUDA cores, 64 double-precision units and 32 special function units (SFU) and 32 load/store units (LD/ST).
- A Fermi SM has 32 CUDA cores.

Streaming Multiprocessors(SMs)



GPU

Shift to GPUs

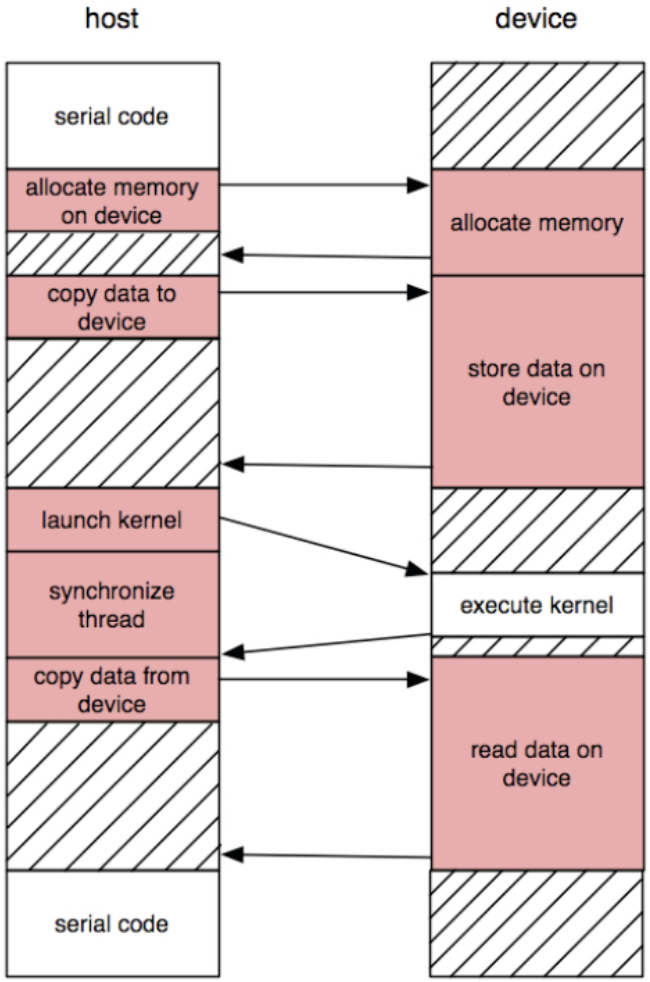
- Single core processors have hit performance wall due to heat dissipation and power requirements
- Hardware industry created multicore CPUs to overcome these limitations
- Science community has taken notice of GPU performance and have ported codes to use GPUs
- GPU programming model different, and programmers need tools to provide unified view of application's performance

CUDA

- Early work used the original interfaces developed for Graphics (eg OpenGL)
- Non-intuitive, high learning curve $\rightarrow$ NVIDIA developed CUDA
- Developed specifically for general computation
- Abstracts the hardware into the CUDA Model
- CUDA is a language extension
- Library calls for your C/Fortran code
- A runtime system to handle data movement, GPU scheduling, etc.
- Open Source Alternative: OpenCL(Available on a very wide range of devides. Flexibility costs: complexity)

GPU Programming

http://www.geek.com/wp-content/uploads/2009/03/xeon-processor.jpg



http://cc.doc.ic.ac.uk/projects/prj_axel/images/Tesla_c1060.png

GPU Programming with CUDA

```
1 int main(int argc, char** argv) {
2     int a_host;
3     int b_host;
4     int answer;
5
6     a_host = thread_id;
7     b_host = 3;
8
9     answer = a_host + b_host;
10
11    return 0;
12 }
```

```
1 __global__ void add(int *a, int *b, int *answer_dev) {
2   *answer_dev = *a + *b;
3 }
4
5 int main(int argc, char** argv) {
6   int a_host;
7   int *a_dev;
8   int b_host;
9   int *b_dev;
10  int answer;
11  int *answer_dev;
12
13  a_host = thread_id;
14  b_host = 3;
15
16  cudaMalloc((void **) &answer_dev, sizeof(int));
17  cudaMalloc((void **) &a_dev, sizeof(int));
18  cudaMalloc((void **) &b_dev, sizeof(int));
19
20  cudaMemcpy(a_dev, &a_host, sizeof(int), cudaMemcpyHostToDevice);
21  cudaMemcpy(b_dev, &b_host, sizeof(int), cudaMemcpyHostToDevice);
22
23  dim3 dimGrid(1);
24  dim3 dimBlock(1,32);
25  add<<<dimGrid, dimBlock, 0>>>(a_dev, b_dev, answer_dev);
26
27  cudaThreadSynchronize();
28
29  cudaMemcpy(&answer, answer_dev, sizeof(int), cudaMemcpyDeviceToHost);
30
31  return 0;
32 }
```

define kernel

serial code

allocate memory

move data

launch kernel

move data

serial code

Review on Terminology

- SM: Streaming Multiprocessor, the component which performs the actual computations on GPU. One GPU has Multi-SMs and each SM has many cores
- Host: The CPU and its memory (host memory)
- Device: The GPU and its memory (device memory)
- Kernel: functions that execute on GPU devices.
- Kernel Launch: call device code from a host

Module,Compiler and Queue

- CUDA Module on BlueHive

```
$module load cuda/5.5/b1
```

- Compiler nvcc (VS. gcc, g++ etc.)

```
$which nvcc
```

- GPU Queue: graphics, standard

```
$salloc -p graphics --gres=gpu:1 -t 00:30:00
$srun --pty $SHELL -I
```

Hello World!

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

```
1 $ nvcc hello_world.cu
2 $ ./a.out
3 Hello World!
4 $
```

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

Hello World! with Device Code

```
1 __global__ void kernel(void) {
2 }
3
4 int main(void) {
5 kernel<<<1,1>>>();
6 printf("Hello␣World!\n");
7 return 0;
8 }
```

- CUDA C/C++ keyword __global__ indicates a function that:
  - Runs on the device
  - Is called from the host code

Hello World! with Device Code

```
1 __global__ void kernel(void) {
2 }
3
4 int main(void) {
5 kernel<<<1,1>>>();
6 printf("Hello␣World!\n");
7 return 0;
8 }
```

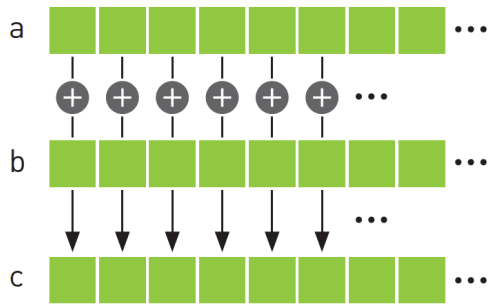
- Triple angle brackets mark a call from host code to device code
  - Also called a "kernel launch"
  - We'll return to the parameters (1,1) in a moment
- That's all that is required to execute a function on the GPU!

Hello World! with Device Code

```
1 __global__ void kernel(void) {
2 }
3
4 int main(void) {
5 kernel<<<1,1>>>();
6 printf("Hello␣World!\n");
7 return 0;
8 }
```

- kernel() does nothing!
- No data move needed!
- Not a typical "Hello World!" example: Can we put the printf line in the kernel?

Vector Addition on the Device



- Ask the device to do more: add two vectors and send the results back
- Big picture: Need to do memory allocation and data copy between CPU and GPU
- Will use the code in Lab 1

Serial Code for Vector Addition

```
1 for (int i=0; i<N; i++)
2   C[i] = A[i] + B[i];
```

- How to parallel it with MPI or OpenMP?

Vector Addition on the Device

```
1 C[thread_id] = A[thread_id] + B[thread_id];
```

- Suppose you have enough threads to use and the sizes of the vectors are small.
- Each thread computes one element of the vector
- Still need serial code to allocate memory and do data tranfer

Vector Addition on the Device

```
1  int main() {
2    int* a;       //input arrays (on host)
3    int* b;
4    int* res;     //output array (on host)
5
6    int* a_dev;  //input arrays (on GPU)
7    int* b_dev;
8    int* res_dev; //output array (on GPU)
9
10   //allocate memory
11   a = (int*) malloc(N*sizeof(int));
12   b = (int*) malloc(N*sizeof(int));
13   res = (int*) malloc(N*sizeof(int));
```

Vector Addition on the Device

Allocate memory on device:

```
1  cudaMalloc((void**) &a_dev, N*sizeof(int));
2  cudaMalloc((void**) &b_dev, N*sizeof(int));
3  cudaMalloc((void**) &res_dev, N*sizeof(int));
```

Move data to device:

```
1  //transfer a and b to the GPU
2  cudaMemcpy(a_dev, a, N*sizeof(int),
       cudaMemcpyHostToDevice);
3  cudaMemcpy(b_dev, b, N*sizeof(int),
       cudaMemcpyHostToDevice);
```

GPU Programming with CUDA

Define kernel:

```
1 __global__ void kernel(int* res, int* a, int* b) {
2   //function that runs on GPU to do the addition
3   //sets res[i] = a[i] + b[i]; each thread is
        responsible for one value of i
4
5   int thread_id = threadIdx.x + blockIdx.x*blockDim.x
        ;
6   if(thread_id < N) {
7     res[thread_id] = a[thread_id] + b[thread_id];
8   }
9 }
```

GPU Programming with CUDA

Launch kernel:

```
1 //call the kernel
2   int threads = 512;                    //# threads per
        block
3   int blocks = (N+threads-1)/threads; //# blocks (N/
        threads rounded up)
4   kernel<<<blocks,threads>>>(res_dev, a_dev, b_dev);
```

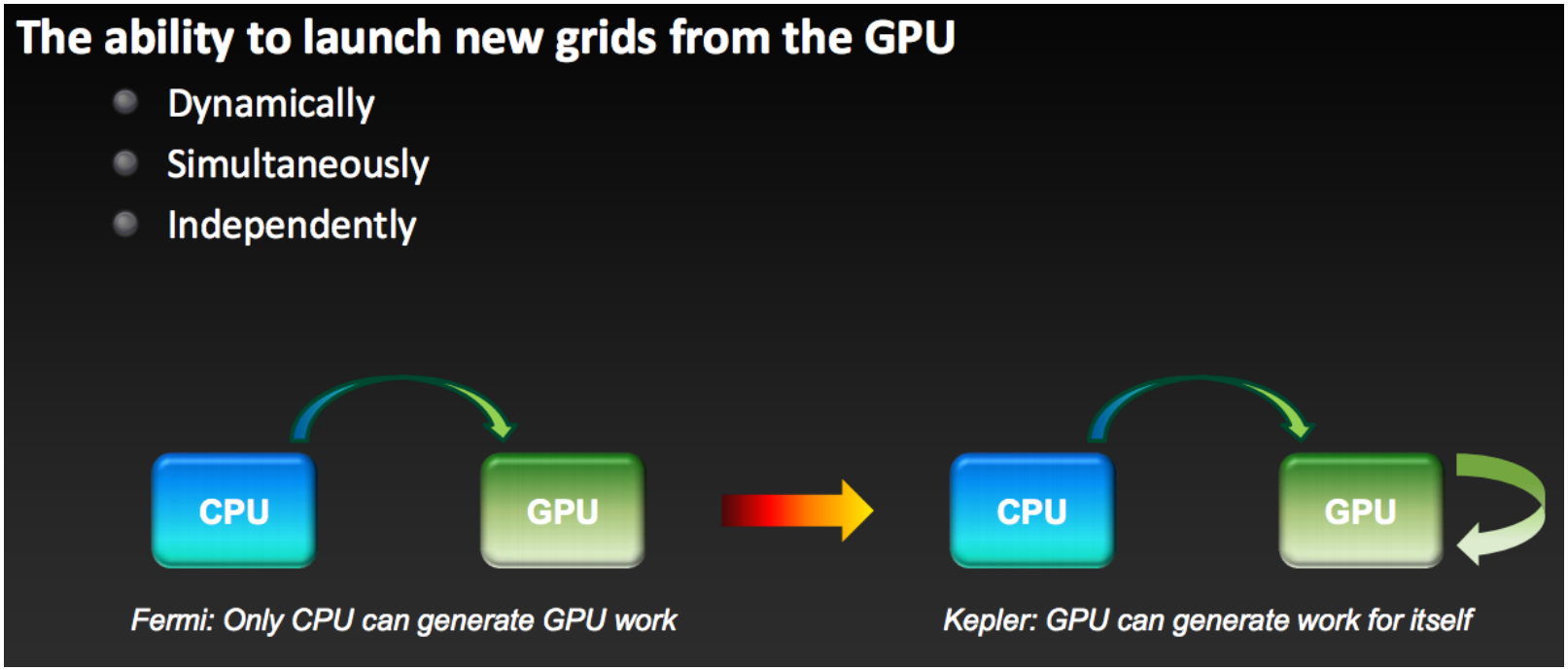
Move data:

```
1 cudaMemcpy(res, res_dev, N*sizeof(int),
      cudaMemcpyDeviceToHost);
```

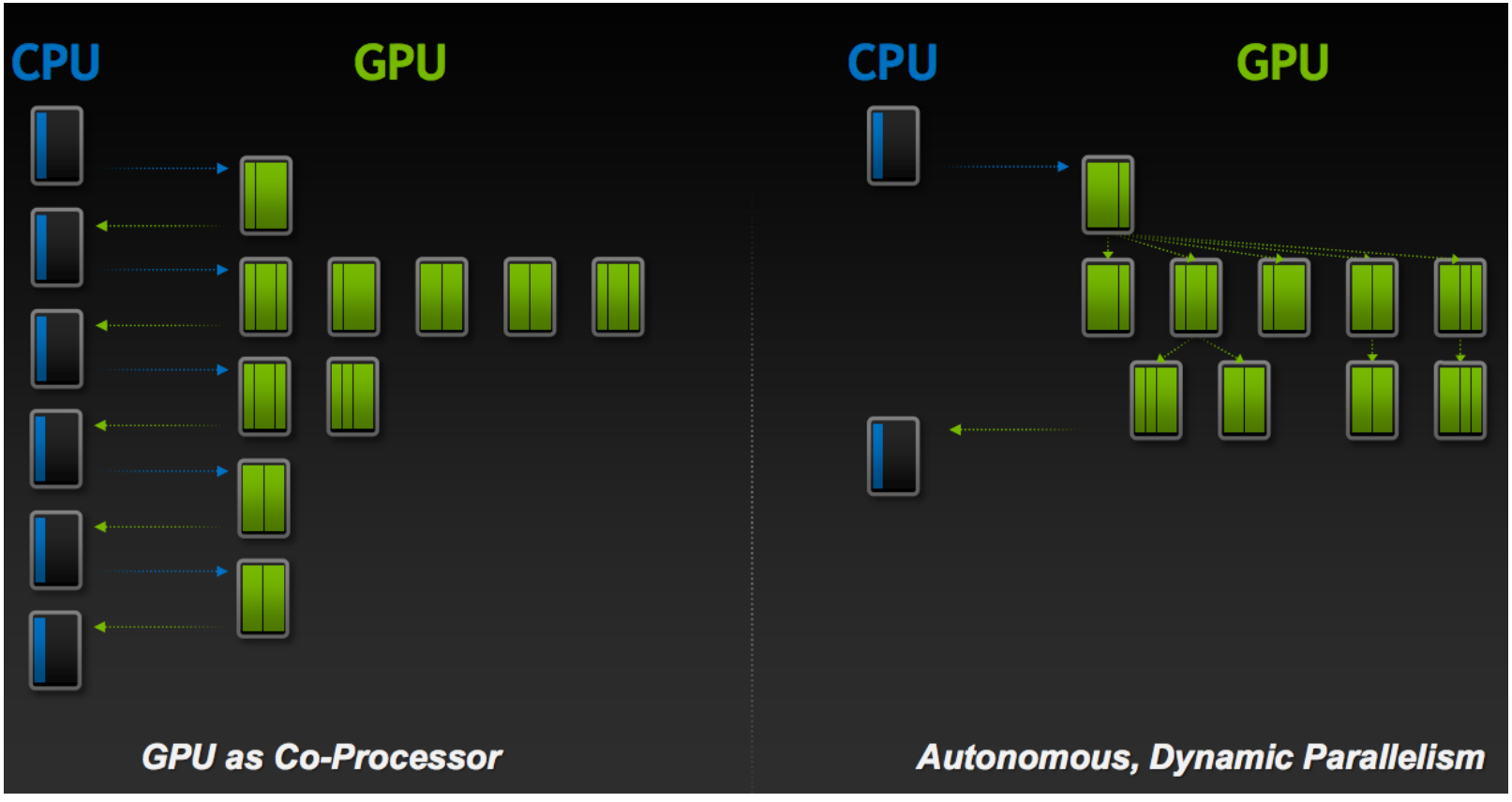
Review on GPU Programming with CUDA

- Define kernel
- Allocate memory
- Copy data from host to device
- Launch kernel
- Copy data from device to host

New Technology on Tesla K20



New Technology on Tesla K20



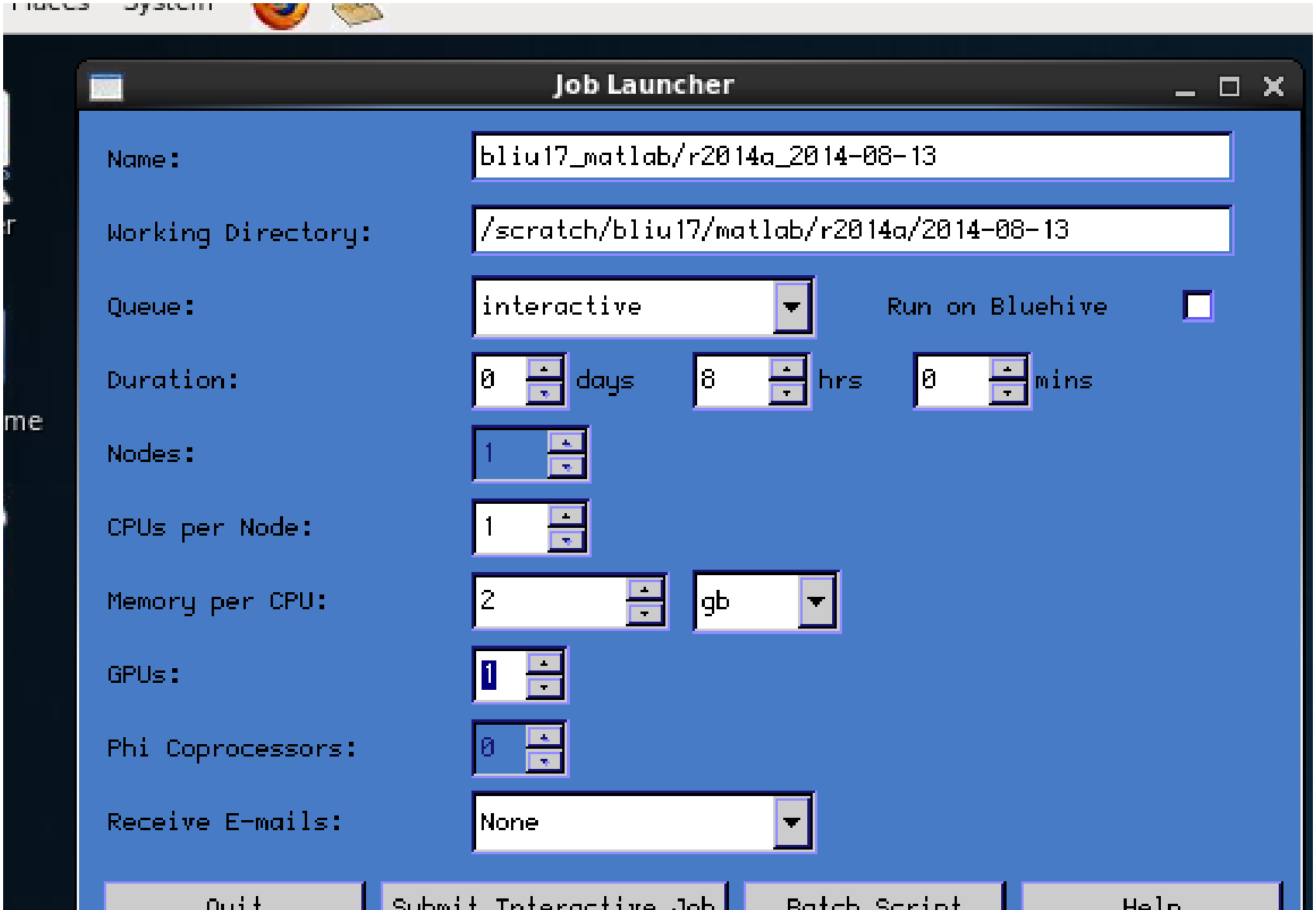
Text Editors

- Linux editors: vi, emacs
- X2Go GUI for BlueHive: gedit
- info.circ.rochester.edu, Getting Started

CUDA Fortran

- PGI Fortran Compile `https://www.pgroup.com/resources/cudafortran.htm`. Not installed on BlueHive
- Call C functions from Fortran code? .o file can be generated from .cu but fortran compiler on BlueHive would recognize.

Matlab with GPU on BlueHive



Matlab with GPU on BlueHive

CUDA: Test if there's a device availale.

```
1 cudaGetDeviceCount(&deviceCount);
```

Questions about 1st Day Class

- Other Questions?
- Try Lab 1 by yourself

Kernel Launch

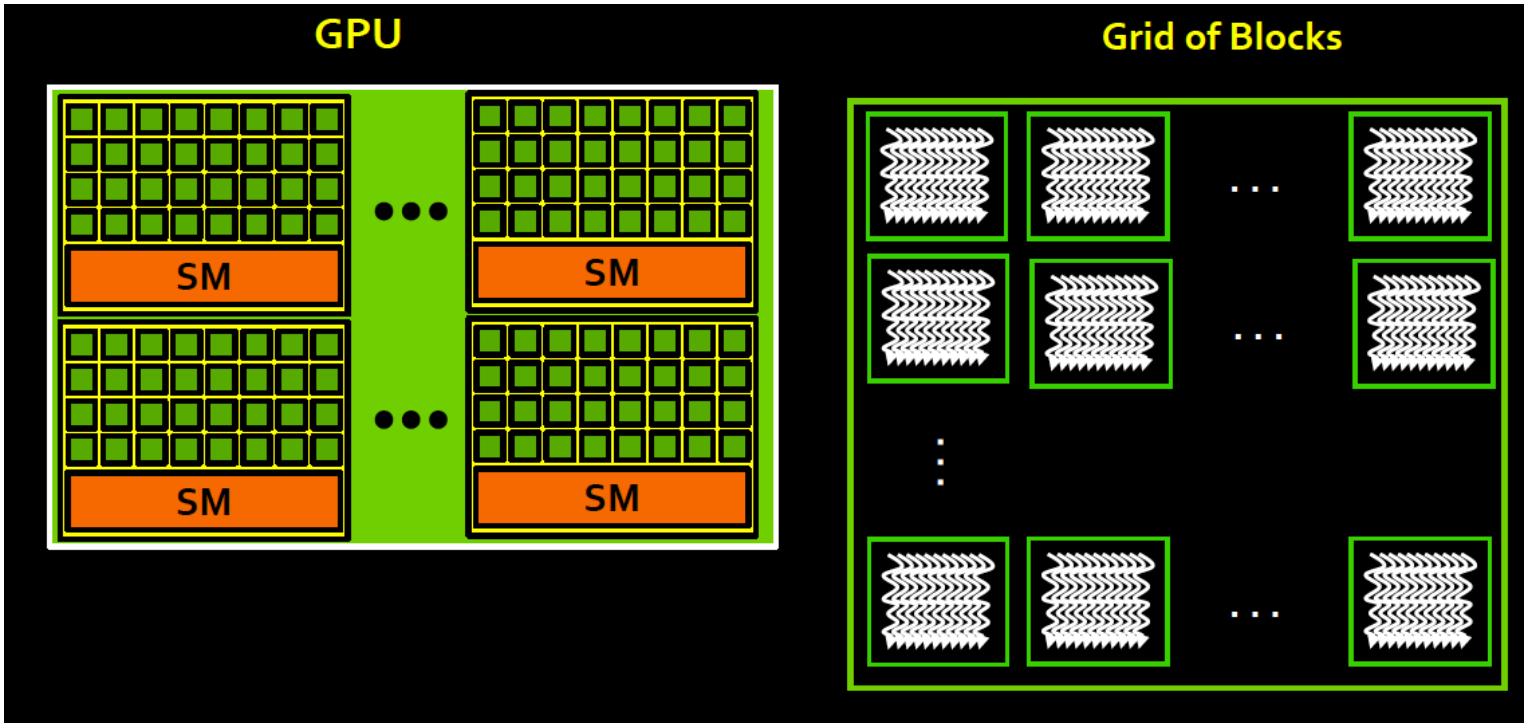
```
1 kernel<<<blocks,threads>>>(res_dev, a_dev, b_dev);
```

- What is block?
- Why blocks/threads instead of just threads?
- How many blocks/threads can we use?

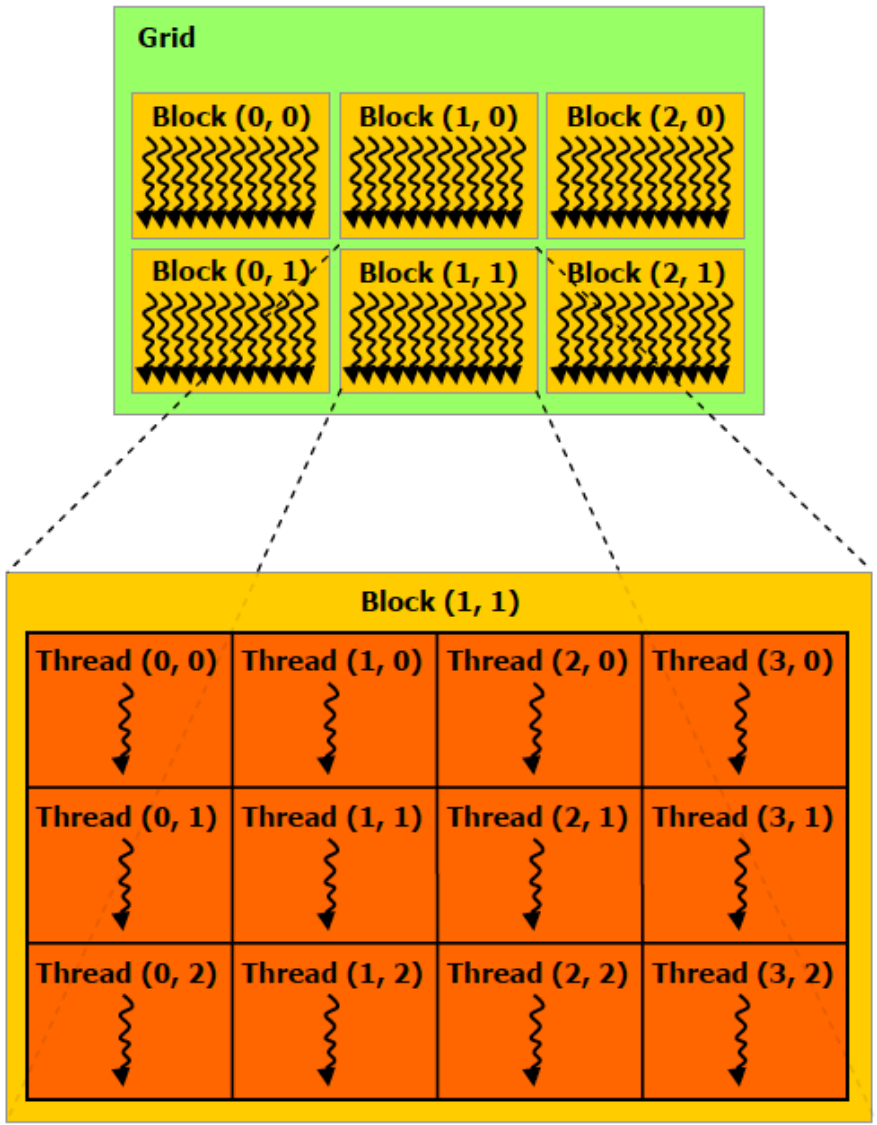
What is a Block?

- Threads are organized in Blocks.
- Blocks are executed concurrently on SMs
- Blocks have fast communication through shared memory
- Hardware dispatches thread blocks to available processor

What is a Block?

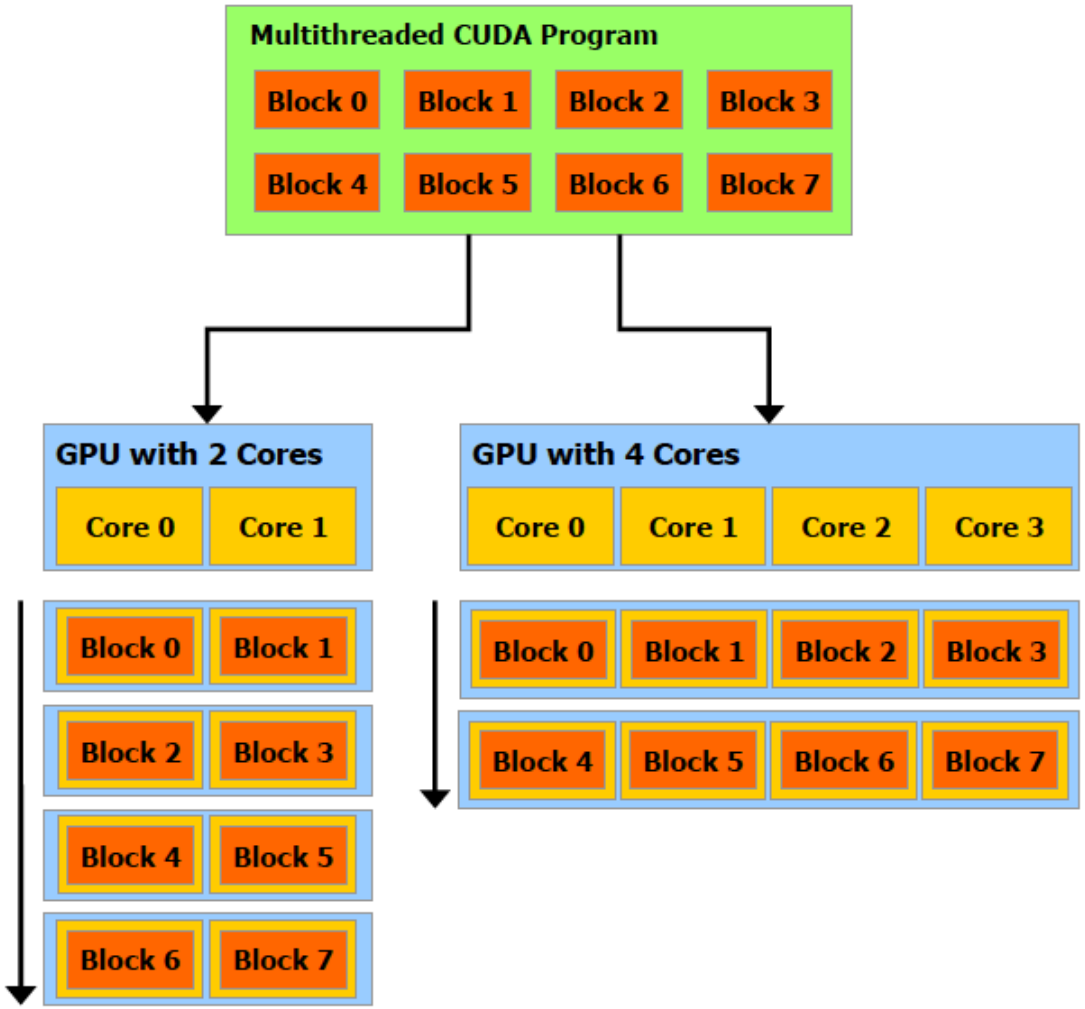


What is a Blocks?



Why Blocks? – Automatic Scalability

Scalable programming model



A multithreaded program is partitioned into blocks of threads that execute independently from each

How Many Threads/Blocks You May Use?

On Tesla K20X

- Max Blocks executed concurrently per SMX: 16
- Max Threads per SMX: 2048
- Max Threads per Block: 1024
- Could be changed in the future

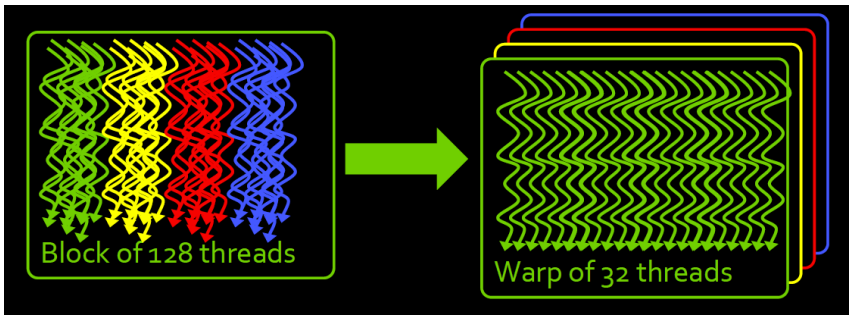
Find the Index

```
1 int i = blockIdx.x * blockDim.x + threadIdx.x;
2 int j = blockIdx.y * blockDim.y + threadIdx.y;
3 int k = blockIdx.z * blockDim.z + threadIdx.z;
```

Special CUDA Variables

- blockIdx: block Id
- threadIdx: thread Id

Warps and SIMT



- Thread blocks are executed as warps
- A Warp is a group of threads within a block that are launched together
- The hardware schedules each warp independently

Warps and SIMT

- Issue a single instruction to the threads in a warp: Single Instruction Multiple Threads
- The warp size is 32 threads
- When a warp is selected for execution, all threads execute the same instruction

Warps and SIMT

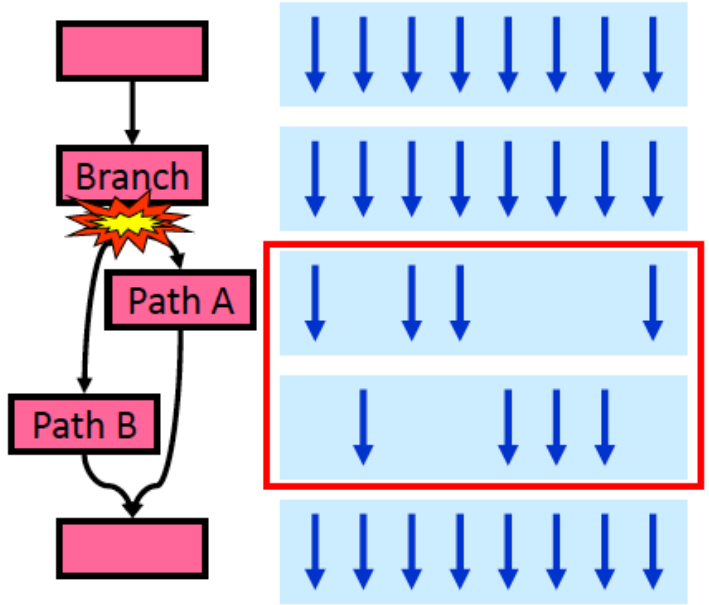
```
1 kernel1<<<N , 1>>>(...);
2 kernel2<<<N/32 , 32>>>(...);
3 kernel3<<<N/128 , 128>>>(...);
```

- Block sizes for full warps
- Thumb of Rule: pick up threads number as multiples of 32

Branch Divergence and Warps

Branch Divergence in Warps

- occurs when threads inside warps branches to different execution paths.

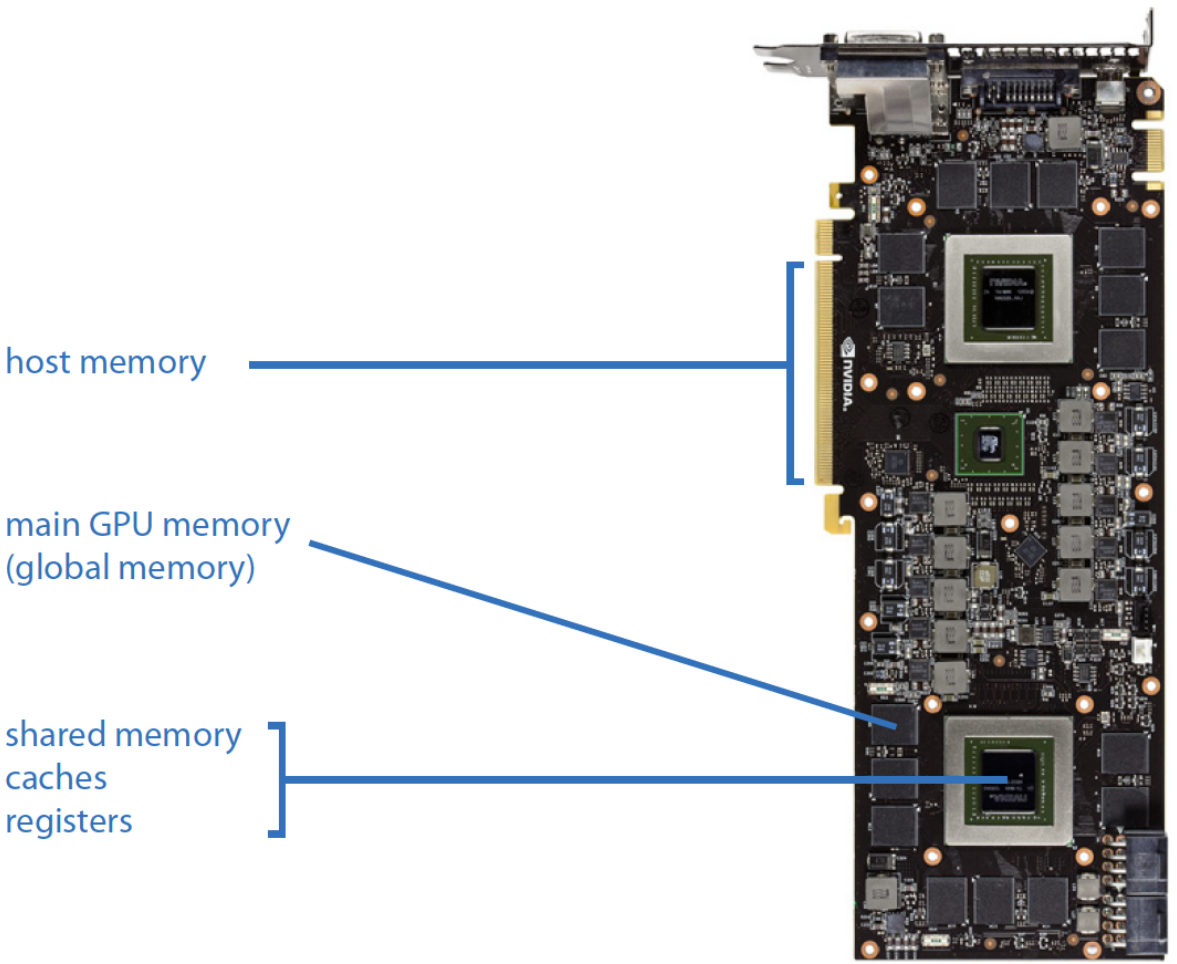


50% performance loss

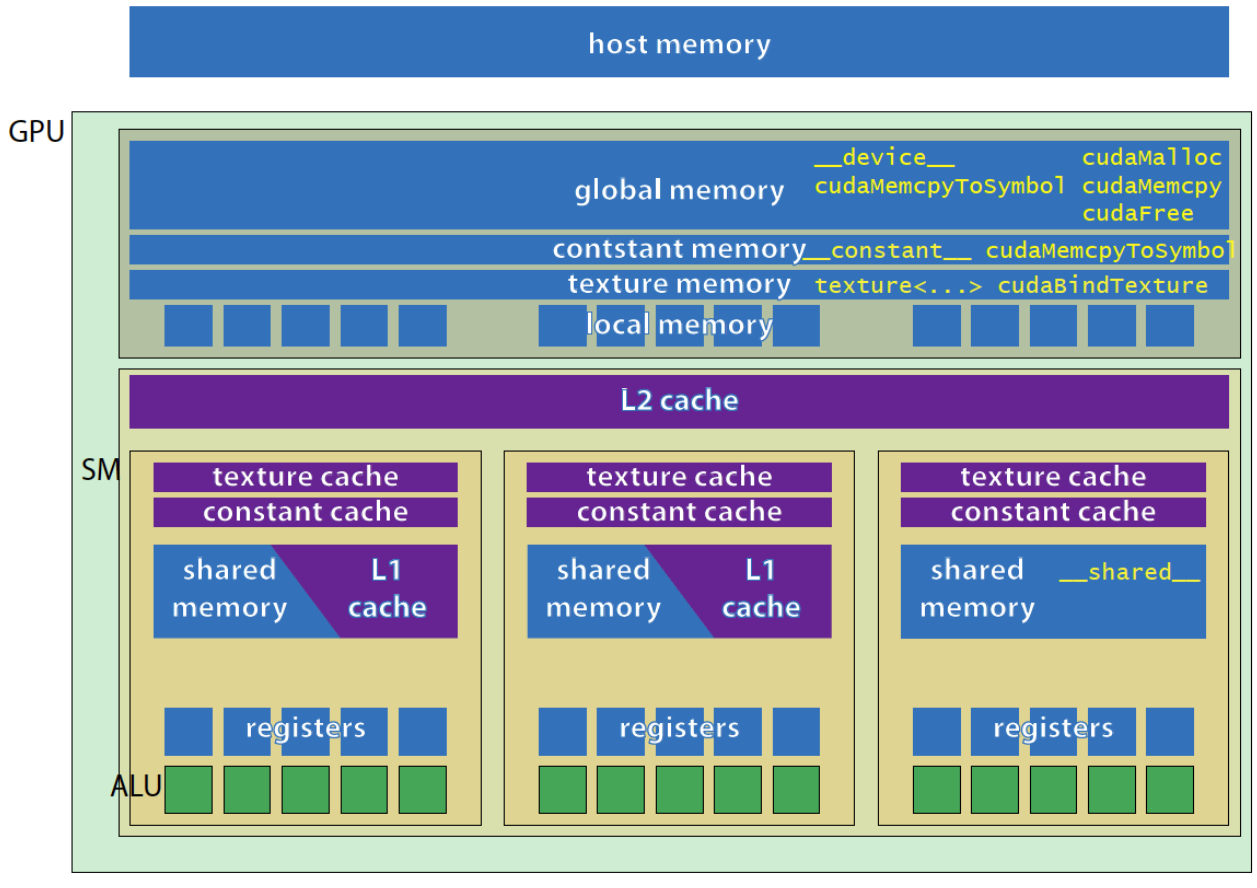
Reviews on Terminology

- Thread: Concurrent code and associated state executed on the device. The fine grain unit of parallelism in CUDA.
- Block: a group of threads that are executed together and form the unit of resources assignment
- Grid: a group of thread blocks that must all complete before the next phase of the program can begin on the GPU
- Warp: a group of threads (32) executed physically in parallel in GPU.

Memory Hierarchy



Memory Hierarchy



Memory Hierarchy

Registers	Registers are the fastest memory, accessible without any latency on each clock cycle, just as on a regular CPU. A thread's registers cannot be shared with other threads.
Shared Memory	Shared memory is comparable to L1 cache memory on a regular CPU. It resides close to the multiprocessor, and has very short access times. Shared memory is shared among all the threads of a given block. Section 3.2.2 of the Cuda C Best Practices Guide [1] has more on shared memory optimization considerations.
Global Memory	Global memory resides on the device, but off-chip from the multiprocessors, so that access times to global memory can be 100 times greater than to shared memory. All threads in the kernel have access to all data in global memory.
Local Memory	Thread-specific memory stored where global memory is stored. Variables are stored in a thread's local memory if the compiler decides that there are not enough registers to hold the thread's data. This memory is slow, even though it's called "local."
Constant Memory	64k of Constant memory resides off-chip from the multiprocessors, and is read-only. The host code writes to the device's constant memory before launching the kernel, and the kernel may then read this memory. Constant memory access is cached — each multiprocessor can cache up to 8k of constant memory, so that subsequent reads from constant memory can be very fast. All threads have access to constant memory.
Texture Memory	Specialized memory for surface texture mapping, not discussed in this module.

Race Condition

- When two or more threads want to access and operate on a memory location without synchronization
- solutions: synchronization between threads or atomic operations

Synchronization between Threads

```
1 __syncthreads();
```

- Synchronization between threads in a block
  - Every thread in the block has arrived at this point in the program
  - All loads have completed
  - All stores have completed
- Can hang your program if used within an if, switch or loop statement
- Threads within a warp are implicitly synchronized

Atomic Operations

- Atomic operations deal with race conditions
  - It guarantees that while the operation is being executed, others cannot access that location in memory
  - Still we cannot rely on any ordering of thread execution
- Examples:

```
1 atomicAdd
2 atomicSub
3 atomicMin
```

Lab 1: Data Transfer Time

In this lab, you'll get some hands-on experience of CUDA coding and compiling and running CUDA codes on the gpu nodes of BlueHive. You will be examining some of the factors that affect the performance of programs that use the graphics processing unit. In particular, you'll see the cost of transfering data back and forth to the graphics card and how the different threads are joined together.

Lab 1: Data Transfer Time

First we look at the time required to transfer data to the GPU and back. Begin by creating a folder for yourself and copying the file:

```
$ mkdir Your_CUDA_Folder
$ cd Your_CUDA_Folder
$ cp /home/bliu17/SummerSchool/CUDA/Labs/
    Lab1/* .
```

To compile the code, we need to load the cuda module

```
$ module load cuda/5.5/b1
$ nvcc -o addVectors addVectors.cu
```

Lab 1: Data Transfer Time

And to run the job on BlueHive, we need to connect to a gpu node. Since we can only use the head node to compile the code, it would be convenient to use a different window or tab to run the job:

```
1  $ salloc -p graphics --gres=gpu:1 -t 00:30:00
2  $ srun --pty $SHELL -l
3  $ ./addVectors
```

Lab 1: Data Transfer Time

You should get a printout with a time, which is how long the program took to add two vectors (of length 1,000,000). Record the number.

```
$ time: 5.132576 ms
```

Lab 1: Data Transfer Time

Now let's examine the code itself. Right near the top is the definition of the function kernel:

```
1 __global__ void kernel(int* res, int* a, int* b) {
2 //function that runs on GPU to do the addition
3 //sets res[i] = a[i] + b[i]; each thread is
     responsible for one value of i
4 int thread_id = threadIdx.x + blockIdx.x*blockDim.x;
5 if(thread_id < N) {
6 res[thread_id] = a[thread_id] + b[thread_id];
7 }
8 }
```

This is a pretty standard function to add the vectors *a* and *b*, storing the sum into vector *res*.

Lab 1: Data Transfer Time

Let's see how this time breaks down between the data transfer between the main system (called the host) and the graphics card. Open the file and comment out the line that calls the kernel:

```
1 kernel<<<blocks,threads>>>(res_dev, a_dev, b_dev);
```

(This is the 3rd time "kernel" appears in the file and occurs near the middle of main.) Then recompile and run the program again. Record the time. If you get errors recompiling the code on gpu node, you may want to open another terminal and login to BlueHive and compile the code there. Otherwise you will have to cancel the job and recompile it on the login node. Then you will need to connect to a gpu node gain and run it.

Lab 1: Data Transfer Time

The program is now transferring the data back and forth, but not actually performing the addition. You'll see that the running time hasn't changed much. This program spends most of its time transferring data because the computation does very little to each piece of data and can do that part in parallel.

Lab 1: Data Transfer Time

To see this another way, open the file again and uncomment the kernel call and the verify paragraph. The comment out the lines that transfer the data to the GPU; these are in the paragraph commented as "transfer a and b to the GPU" (use the search function to find it). The modify the kernel to use thread_id instead of a[thread_id] and b[thread_id]. (The program initializes a[i] and b[i] to both be i; see the "set up contents of a and b" paragraph.)

```
1    //res[thread_id] = a[thread_id] + b[thread_id];
2    res[thread_id] = thread_id + thread_id;
```

Lab 1: Data Transfer Time

The resulting program should be equivalent to the original one except that instead of having the CPU initialize the vectors and then copy them to the graphics card, the graphics card is using its knowledge of their value to compute the sum, thus avoiding the first data transfer. Recompile and rerun this program and record the time.

Lab 1: Data Transfer Time

Now the time is considerably less than the 3 milliseconds we started with. (We're no longer copying the two vectors, which are each a million entries long...)

Lab 1: Data Transfer Time

Minimize CPU<->GPU Data Transfers

- ~6GB/sec between CPU and GPU
- ~160GB/sec to GPU memory
- Transfers back and forth between CPU and GPU quickly become prohibitively expensive!
- Do as much as possible on the GPU

Lab 2: Thread Divergence

In this Lab we will study the second factor affecting the performance of GPU programs. Copy this file to your CUDA folder:

```
$ cd Your_CUDA_Folder
$ cp /home/bliu17/SummerSchool/CUDA/Labs/
    Lab2/* .
```

This file contains two kernels, creatively named kernel_1 and kernel_2. Examine them and verify that they should produce the same result. Then compile and run on BlueHive:

```
$ nvcc -o divergence divergence.cu
$ salloc -p graphics --gres=gpu:1 -t 00:30:00
$ srun --pty $SHELL -I
$ ./divergence
```

Just like what we saw in Lab 1, it will print a time:

```
$ time: 2.060096 ms
```

Lab 2: Branch Divergence

Then modify the code to use the other kernel, recompile and rerun. You'll see that the running times are quite different.

```
1   kernel_2<<<blocks,threads>>>(a_dev);
2 //kernel_1<<<blocks,threads>>>(a_dev);
```

```
1          $ time: 0.303872 ms
```

That there is a difference is not terribly surprising since the kernels do use different code. To further explore this difference, change the number of cases enumerated in kernel_2 (either delete some or use cut-and-paste to add more). You'll see that its running time changes, which should not happen;switch statements typically have running time independent of the number of cases since they're just a table lookup.

Lab 2: Branch Divergence

The reason for the slowdown has to do with how the GPU cores are organized. This organization is reflected in the grouping of threads into warps. Every thread in a warp considers the same instruction in each cycle. To allow for branches, the threads aren't required to all execute this instruction, but they spend their cycle either executing it or performing a noop. When the threads in a warp want to execute different instructions, the instruction being considered cycles between those that each thread wants to execute. Thus as the control flow causes the threads to split up, each of them spends more cycles executing noops and the program slows down.

Lab 2: Branch Divergence

Turn on the commented "print bucket contents" section of the code and check the value of the array.

Lab 3: Branch Divergence Revisit

In this Lab you are provided a program which is slightly different from the one in Lab 2.

```
$ cp /home/bliu17/SummerSchool/CUDA/Labs/Lab3/* .
```

- Download the code
- Compile and run on BlueHive2
- Check the code difference from Lab 2.

```
a[cell]=cell+1;
//a[cell]++
```

Lab 3: Branch Divergence Revisit

- Turn on the code of "print bucket contents"
- Run the code with kernel1 and kernel2
- Compare with results in Lab 2
- Explain the results

Lab 4: Memory Coalescing

Memory coalescing is a technique which allows optimal usage of the global memory bandwidth. If the threads in a block are accessing consecutive global memory locations, then all the accesses could be combined into a single request (or coalesced) by the hardware. This is an important tenique for devices with compute capability 1.X or CUDA 1.0. Most recent GPU devices or CUDA newer than 2.0 have more sophisticated global memory access and the effect won't be obvious.

Lab 4: Memory Coalescing

In this lab you are provided a program coalescing.cu with different memory access patterns. Download the code and scripts from

```
1 cp /home/bliu17/SummerSchool/CUDA/Labs/Lab4/* .
```

You will need to

- Compile and execute the program on BlueHive
- Change the code to make the memory access with threadID sequentially along columns and see how things would change

Here's the results ran on one of the old GPUs for you to compare with

```
Enter number of blocks in grid, each dimension, currently 1
1
Enter number of iterations, currently 100
100000
Array size (and total grid-block size) 16 x 16

Computation with memory coalescing possible
Results of computation, every N/8 numbers, eight numbers
     0       2       4       6       8      10      12      14
    32      34      36      38      40      42      44      46
    64      66      68      70      72      74      76      78
    96      98     100     102     104     106     108     110
   128     130     132     134     136     138     140     142
   160     162     164     166     168     170     172     174
   192     194     196     198     200     202     204     206
   224     226     228     230     232     234     236     238

Time to calculate results on GPU: 6.626624 ms.

Computation with memory coalescing not possible
Results of computation, every N/8 numbers, eight numbers
     0      32      64      96     128     160     192     224
     2      34      66      98     130     162     194     226
     4      36      68     100     132     164     196     228
     6      38      70     102     134     166     198     230
     8      40      72     104     136     168     200     232
    10      42      74     106     138     170     202     234
    12      44      76     108     140     172     204     236
    14      46      78     110     142     174     206     238

Time to calculate results on GPU: 25.101568 ms.
Speedup with coalescing = 3.787987

Enter c to repeat, return to terminate
```

Lab 5: Shared Memory

Shared memory is fast but with limited size (48KB) comparing with global memory. If your code has to access a piece of memory many times put it in shared memory will dramatically improve the performance. In this lab you are provided a program sharedmemory.cu to test the effect of using shared memory. Download the code and scripts from

```
1 cp /home/bliu17/SummerSchool/CUDA/Labs/Lab5/* .
```

And

- Compile and execute the program on BlueHive
- Understand the results and code

Online Resources

- cuda-zone: https://developer.nvidia.com/cuda-zone
- cuda-toolkit: http://developer.nvidia.com/cuda/cuda-toolkit
- online classes on youtube