GPU Programming with CUDA

Cheng(chx33@pitt.edu), Kim, Leonardo, Fangping, Daniel, Nick

Center for Research Computing

University of Pittsburgh

Table of Contents

Workshop Content

- ctest.cpp: An example C++ program
- cudaHello.cu: A first CUDA program
- ctest.cu: GPU equivalent of ctest.cpp
- ccompute.cu: CUDA program for 2D computation
- All the above files with "_full" extension: Complete source files ready to be compiled
- animate2D.py: Python script that generates animation from output files of ccompute.cu
- CUDA_workshop.pptx: Presentation slides!

Download workshop material

on cluster:

```
cp -r /ihome/workshops/GPU_CUDA_shared $HOME/
```

Part 1: Introduction to C/C++ Programming

Programming in C/C++: Intro

- Elementary data types and storage sizes: **bool** (1 Byte), **char** (1 Byte), **int** (4 Bytes), **float** (4 Bytes), **double** (8 Bytes); 1 Byte = 8 Bits [0...255]
  - Modifiers: **signed, unsigned, long, short**
  - Generic: **void**
  - Example: **long long unsigned int** (0 to 18,446,744,073,709,551,615)
  - Determine storage size in bytes: **sizeof**(...)
  - Allocate storage: **malloc**(…)
- Pointers: Stores address of variable, not data (**int** *i, **double*** x)
- Derived data types: struct, class
- Conditionals: **if**(condition==True){carry out a; } else { carry out b;}
- Comments: //Comment, or /*multi line comment block*/

Programming in C/C++: Intro

- Operators:
  - Unary: Increment/Decrement(++, --), Negation(!), Address, dereference(&,*)
  - Bitwise: And(&), Or(|), Left/right shift (<<, >>)
  - Logical: And (&&), Or (||), Equals (==)
  - Arithmetic: +, -, *, /, %(modulo)
  - Ternary: x = (Condition)? a: b equivalent to if(Condition) x=a; else x=b;
  - May be re-defined for derived data structures
- Loops:
  - **for**(i=0; i<N;i++) {instructions;}
  - **while**(condition==true){instructions;}, **do**{instructions} **while**(condition)
  - Exit loop: **break**
  - Scope: Variable declared within a loop not visible outside a loop!

Paradigms in C/C++

- Procedural programming: Organize large program into smaller parts, i.e. subroutines/modules/functions
  - Declaration: Specify data types of all inputs and output for function/subroutine, Examples: **int** myFunction(**int**, **double**), **void** mySub()
  - Definition: Body of instructions for function/subroutine, return value must match declaration, Example: **int** myFunction(**int** a, **double** b) {double result; instructions; **return** result;}
  - Invoke function: type of input arguments must match declaration! Example: **int** a; **double** b; **double** c = myFunction(a,b);
  - Scope: Variables only visible within the function in which they are declared!
- Object-oriented programming: Abstraction and encapsulation
  - Group data and functions into a **class**
  - Control access of internals from outside: **public**, **private**

Typical C/C++ Program

- **#include** header files containing declarations of functions, constants, and datatypes, read by pre-processor
- Starts with **main** function, may or may not return value upon exit;
- Variables **must** be declared before they can be used!
- Program written in plain text file ending with *.c or *.cpp, IDEs(Eclipse, Code::Blocks, Visual Studio, ...) can be helpful tools
- Spacing symbols (space, new line, tab) don't matter
- Program must be built before execution
- Optional: Invoke routines contained in libraries and link them to own program, must be declared within program!

Example(ctest.cpp): Prime factorization (main)

```
#include <iostream>
#include <stdlib.h>
#include <chrono>
using namespace std;
using namespace std::chrono;
long long unsigned  int chkprime_cpu(long long unsigned int);

int main(int argc, char *argv[] ) {
  char *eptr;
  long long unsigned factor; //main variable declarations
  unsigned long long Nstart = strtoull(argv[1],&eptr, 10); //read number from terminal

  auto start = high_resolution_clock::now(); //start clock
  factor=chkprime_cpu(Nstart); // check for prime factors

  if(!factor) cout<<Nstart<<" is prime!"<<endl;
  else cout<<Nstart<<" has prime factor "<<factor<<endl; //output prime factor
  auto stop = high_resolution_clock::now();
  auto duration = duration_cast<microseconds>(stop - start);
  cout <<"Time in microseconds: "<< duration.count() << endl; //output elapsed time
  return 0;
}
```

Example: Prime factorization (Function)

```
long long unsigned chkprime_cpu(long long unsigned int Number)
{
  unsigned long range = (unsigned long) sqrt(Number);

  for (long long unsigned  i; ;i++) //Fill in the loop to search for prime factor, exit when found!
  {

  }

  return 0;
}
```

Generate binary <OUTPUT> with C++ compiler:

```
module load gcc/8.2.0
g++  -o <OUTPUT>  <SOURCE.cpp>
```

Execute binary: `./<OUTPUT>  <INPUT PARAMETERS>`

Full compilable source file: ctest_full.cpp

Part 2: GPU Hardware & Architecture

Why GPU Programming?

- GPUs pushed the "power wall" hit by multi-core CPUs
- Massively parallel architectures developed as a response to high demand from gaming industry
- GPU's have many small processors: High latency, high parallelism
- if
  - (1) program is computationally intensive (not spending much time transferring data) and
  - (2) massively parallel, so computations can be independent.

  consider using the GPU!

Overview

- GPGPU (or simply GPU) is a device specialized for compute intensive, highly-parallel computations
- Programmable using general purpose programming with extensions--special libraries and instruction set
    - e.g. Nvidia CUDA--an extension of C/C++ (Also available for Fortran)
    - CUDA stands for Compute Unified Device Architecture
- We will discuss GPU hardware architecture, CUDA programming model, CUDA runtime environment and some examples

Why understand hardware?

- Parallel programming is fundamentally linked to the underlying hardware architecture
- Understanding of the underlying hardware enables a programmer to align their code well to the hardware
- Significant **performance gains** could be obtained by correctly mapping the CUDA code to GPU architecture
- Significant **performance losses** by mismatching CUDA code to GPU architecture

CPU and GPU assembly

- GPUs cannot work independently in a computer!
- a CPU is needed to "host" the GPU
- CPU sends instructions and data to GPU and receives results
- GPU is usually mounted on the PCI-e slot
  - PCI-e is peripheral component interconnect express
- if you own a PC, you can buy a GPU and plug it in and you are all set to do GPU programming!

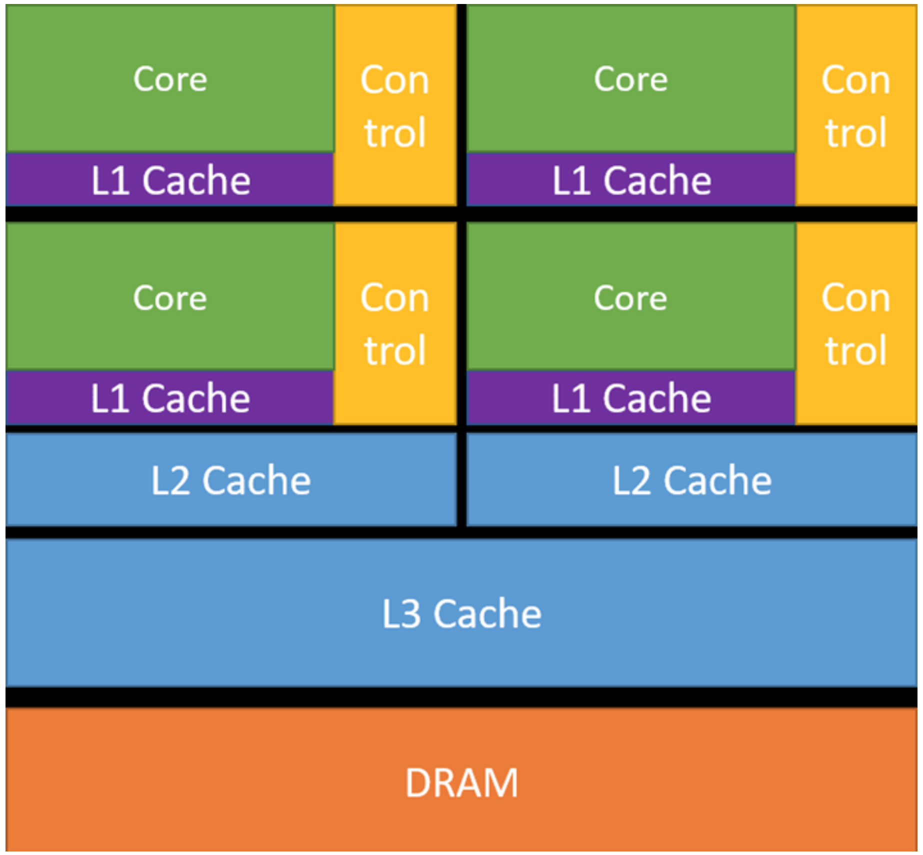
GPU Components

- Outermost assembly consists of the circuit board and cooling system
- Circuit board consists of:
  - GPU "chip"
  - memory
- GPU "chip" is organized as a collection of Streaming multiprocessors (SM)
- Each SM is a collection of Streaming Processors (SP)
- One SP is a GPU "core"
- Memory is organized as per thread local memory, per block **shared** memory, all blocks access **global** memory

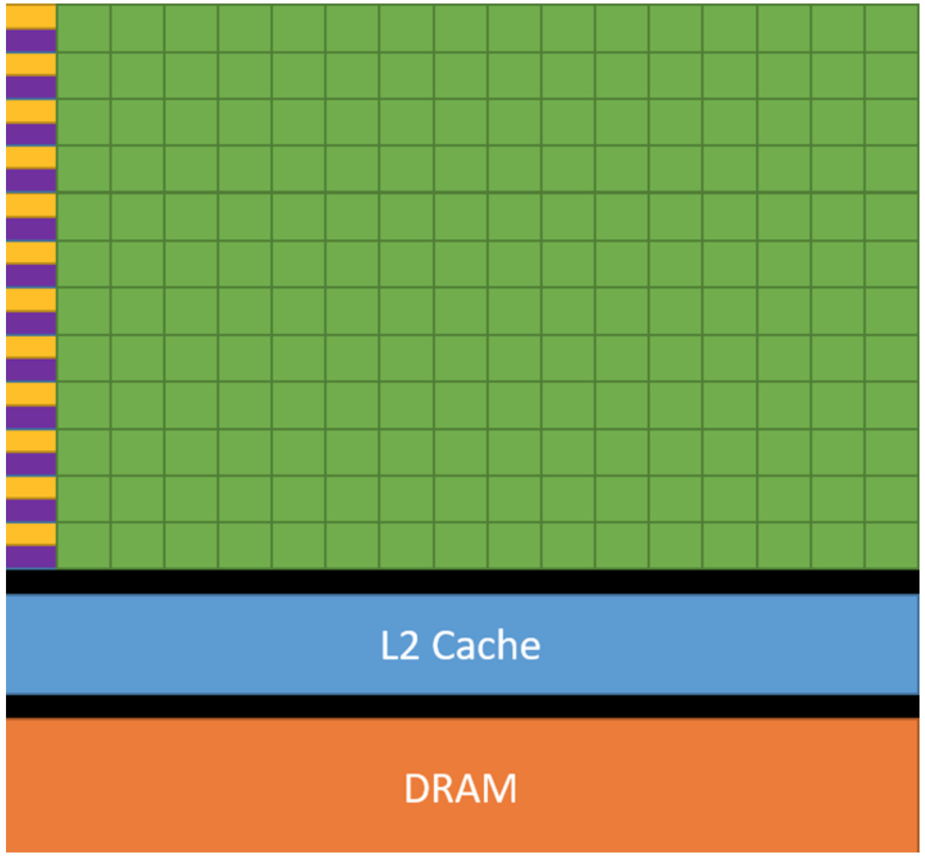
Additional Components

- Gigathread global scheduler: distributes thread blocks to Streaming Multiprocessors
- Warp scheduler: local thread scheduler at the SM level, 32 threads bundled together
- FMA: Fused Multiplication Addition unit that could do (A*B+C) in one step

CPU vs. GPU

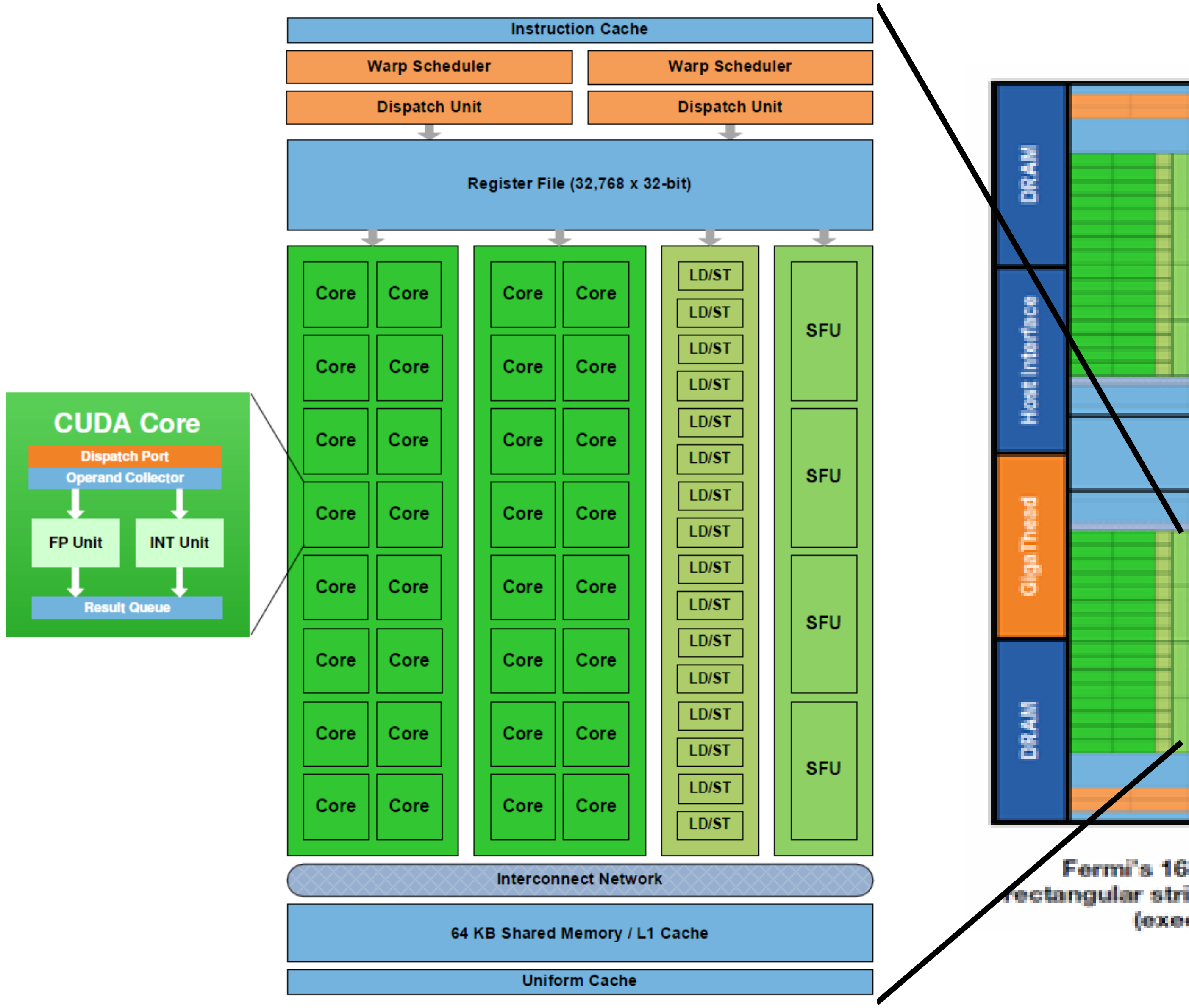


Central **P**rocessing **U**nits



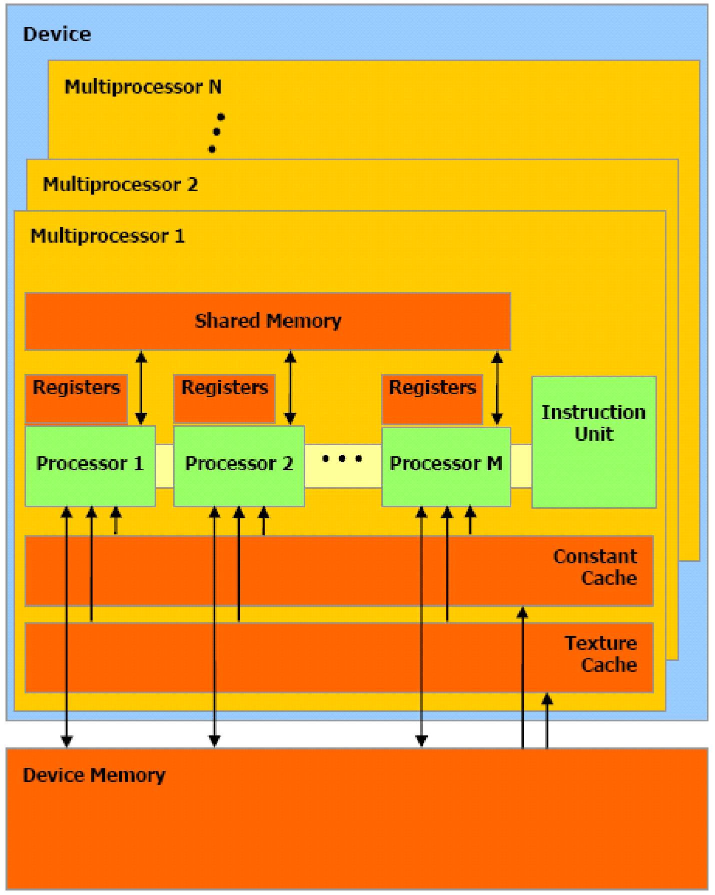
Graphics **P**rocessing **U**nits

Block Diagram of a GPU (Fermi Architect.)

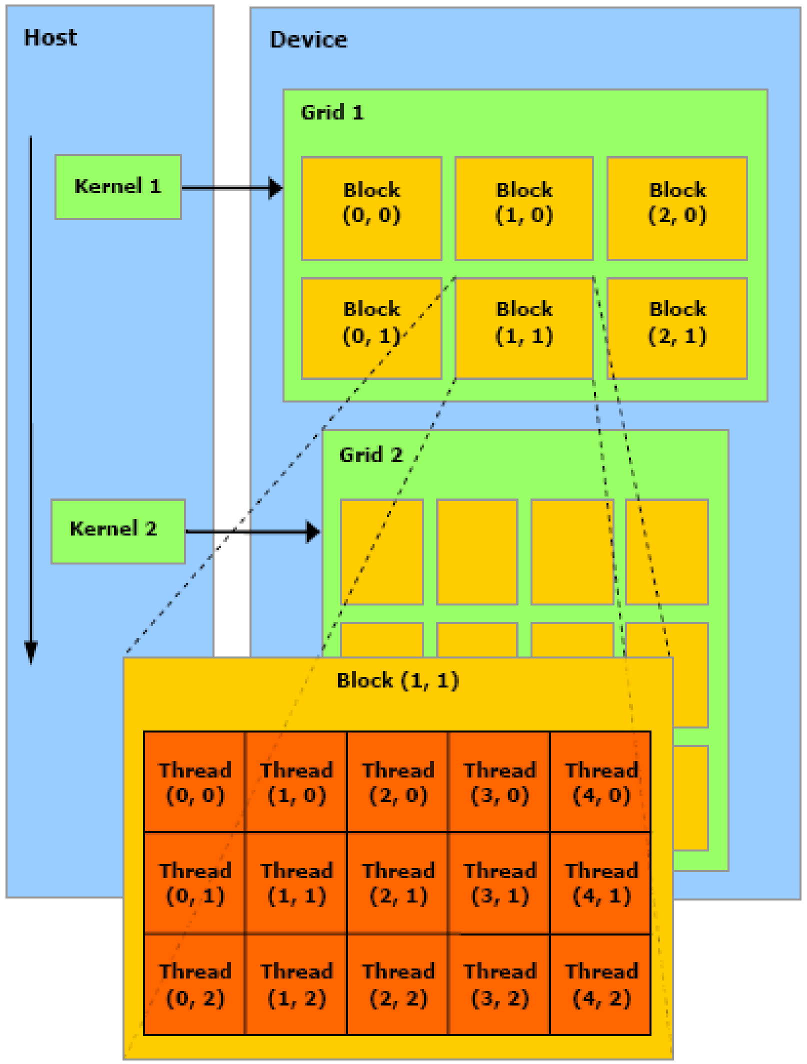


Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

How do we write program for the GPU?
CUDA from NVIDIA



Source: NVIDIA CUDA Programming Guide



Latest GPU resources at CRC

"Standard" A100 ("Ampere") partitions on GPU cluster

- 2 sockets per node, 64 CPU cores/socket
- 1 TB RAM/node
- 4 NVIDIA A100 GPUs/socket
- 40 GB memory per GPU
- Max of 16 CPUs per GPU

3 X86_64 nodes with 8 NVIDIA A100 40GB GPUs/node NVLink

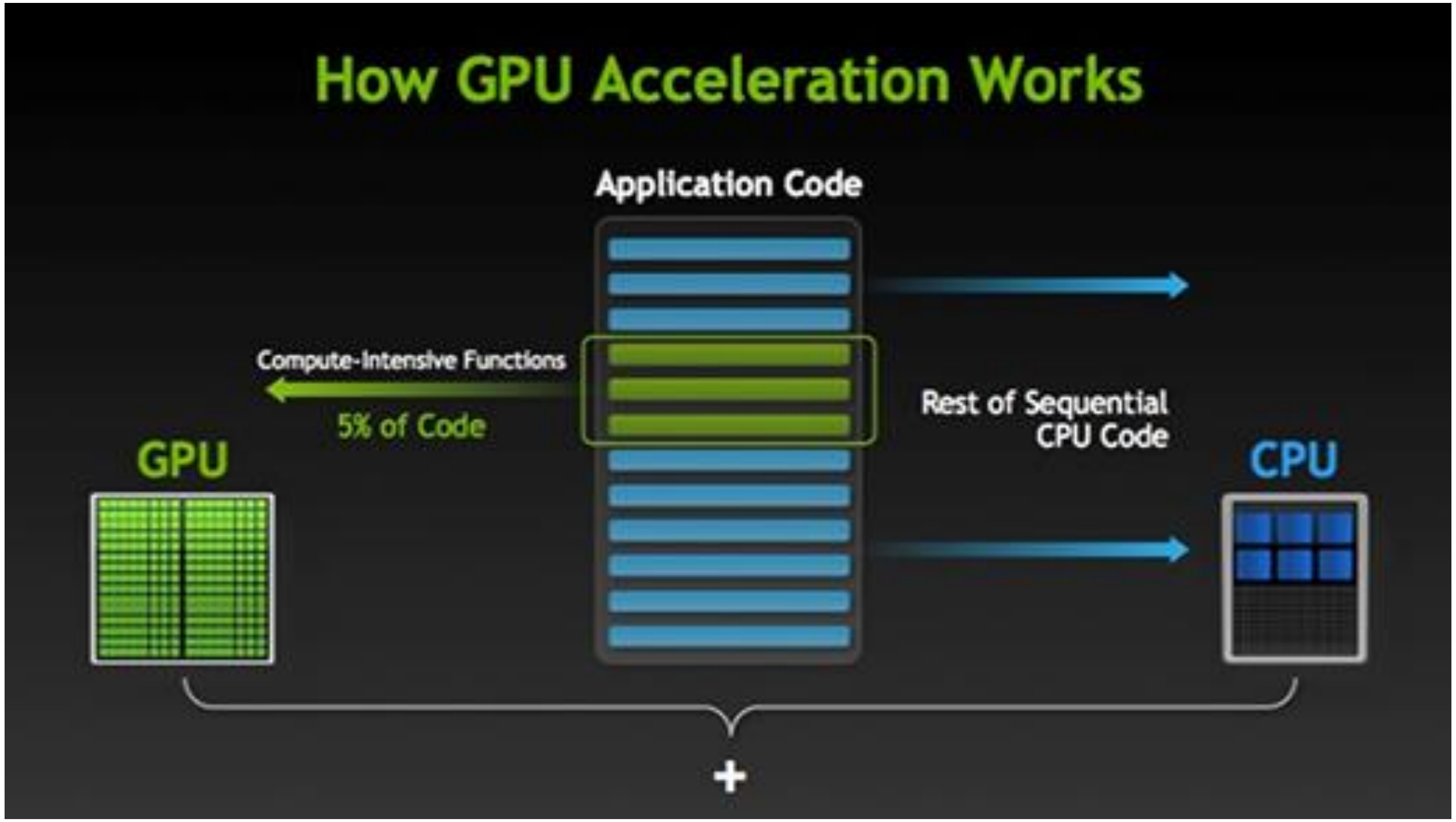
2 X86_64 nodes with 8 NVIDIA A100 80GB GPUs/node NVLink

Older nodes

- Nvidia GeForce gtx 1080
  - Each node has 4 GPUs
  - Each GPU has 2560 CUDA cores
- More at: https://crc.pitt.edu/resources/

Part 3: Introduction to CUDA-C/C++

50K feet overview of CUDA programming


How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
GPU
Rest of Sequential CPU Code
CPU
+

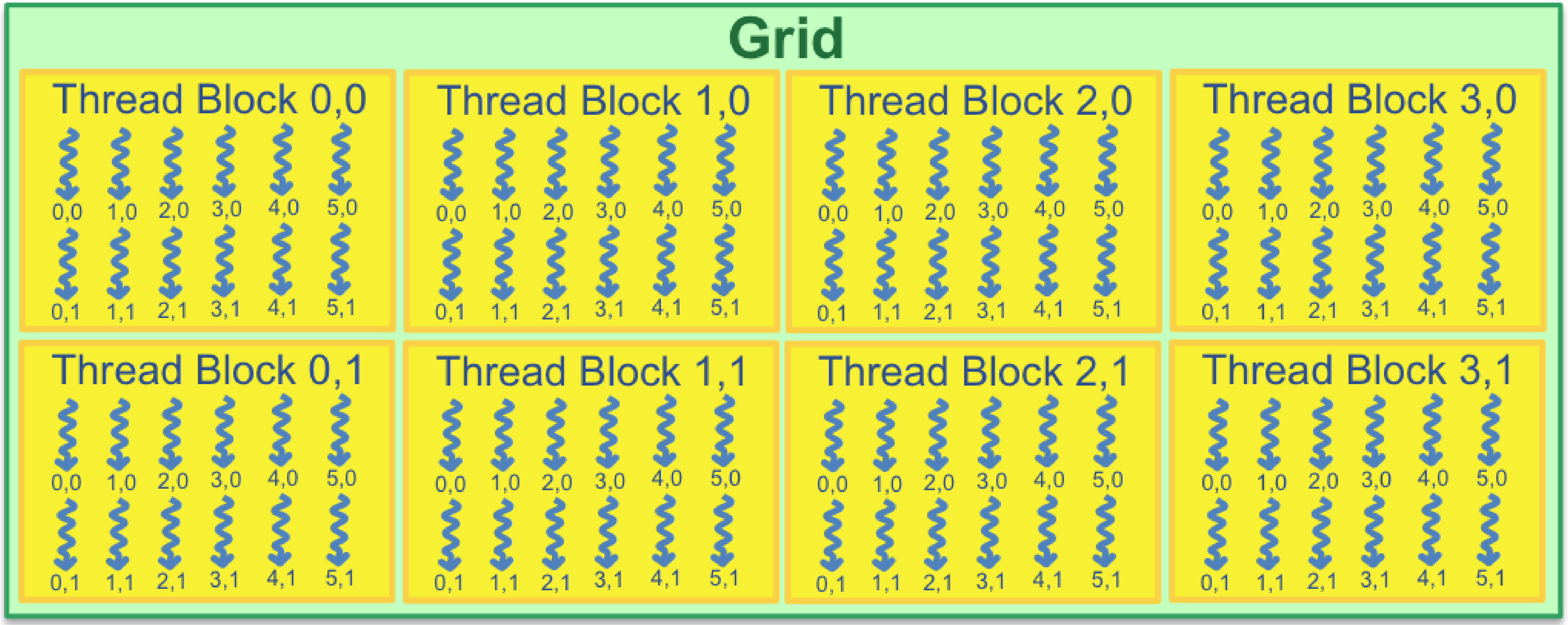
20K feet overview of CUDA programming

- CUDA program consists of code to be run on the **host**, i.e. the CPU, and code to run on the **device**, i.e. the GPU
- Function that is called by the host to execute on the **device** is called a **kernel**
- Running instance of a kernel is **thread**
- Threads in an application are grouped in **blocks**
- Entirety of the blocks is called the **grid** of that application

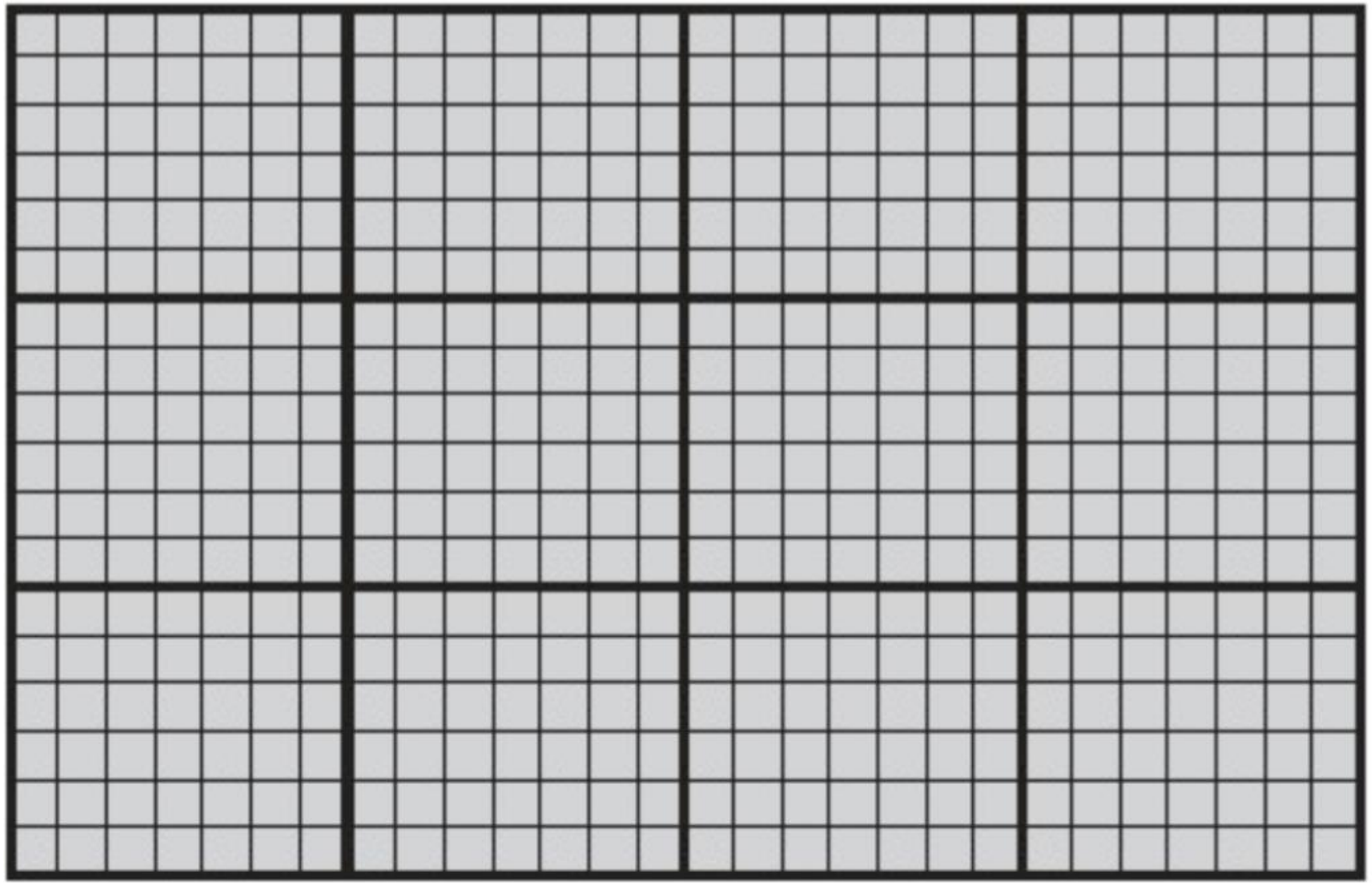
What is grid-block-thread business?

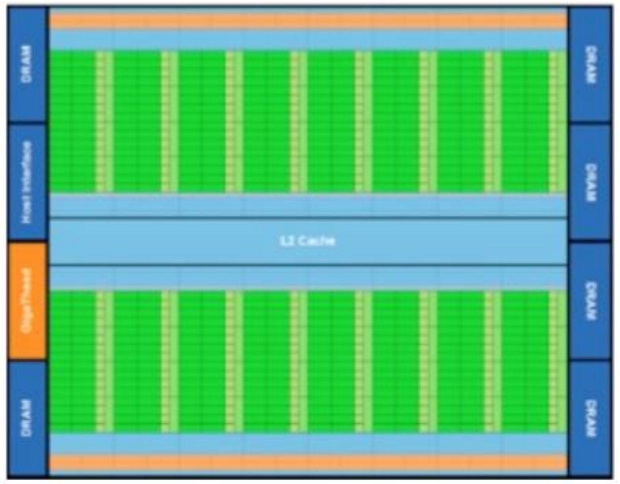
- Organization of **threads** for execution over GPU cores
- A thread is a minimal unit of execution
- **Kernel becomes thread at runtime**
- A group of threads is a block
- A collection of blocks is a grid
- Basic idea: Hide GPU latency with massive data parallelism!
- **One grid is scheduled and launched per kernel**
    - programmer must provide the **shape and size of the grid** when invoking a CUDA kernel!

A 2-D grid-block-thread diagram



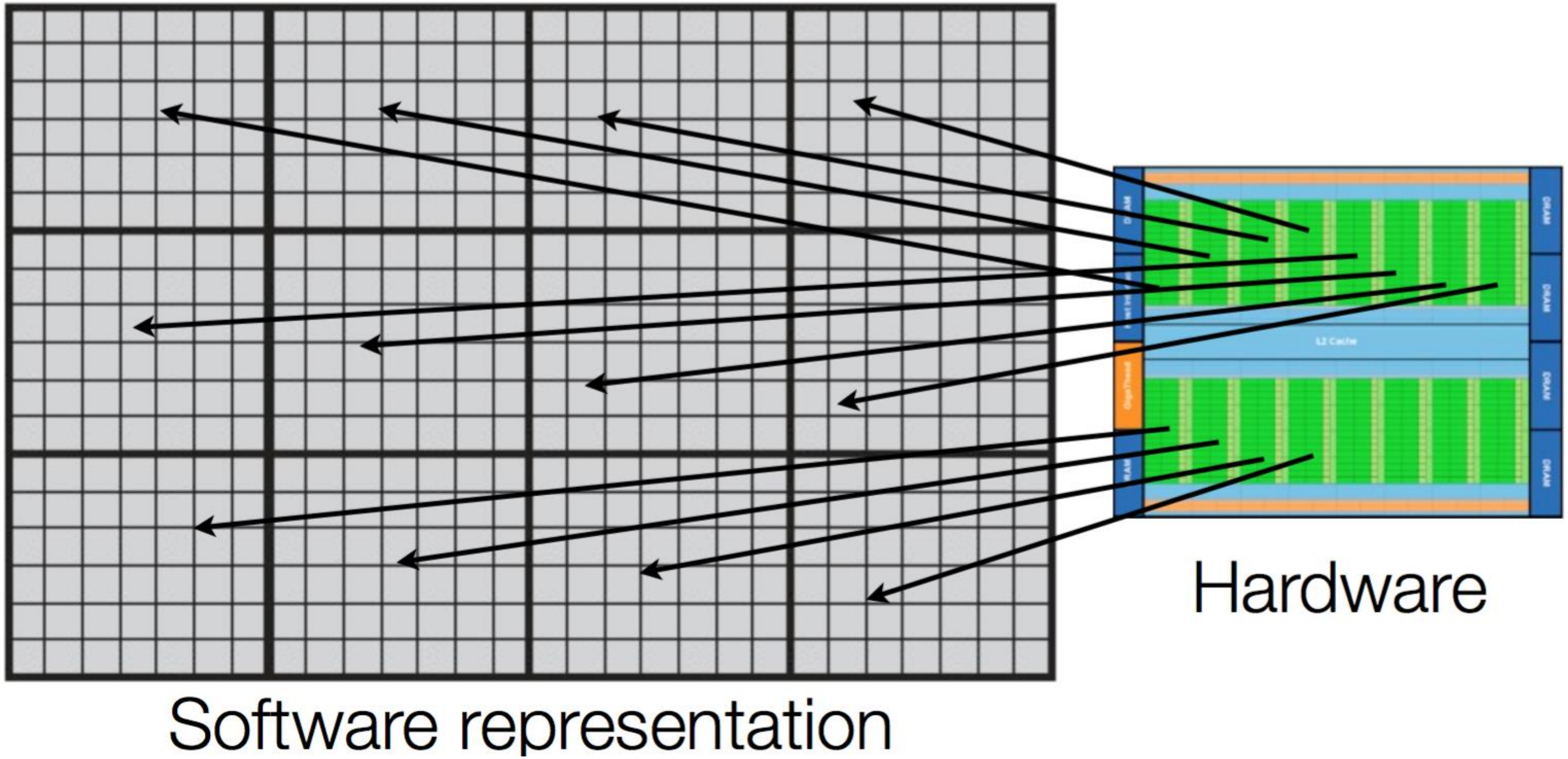
Two representations


Software representation



Hardware

Goal: Map program to GPU hardware



CUDA Jargon

- TypeFunctions
  - `__global__`: a CUDA kernel callable by host
  - `__device__`: a CUDA kernel callable by device
  - `__host__`: a regular function that runs on host
- Kernel invocation from host: kernel<<<blocks, threads>>>(args)
  - dim3 gridDim: how many blocks in grid in terms of **"X times Y times Z"**
  - dim3 blockDim: how many threads in a block in terms of **"X times Y times Z"**
  - dim3 blockIdx: location of a block in grid in terms of (x, y, z)
  - dim3 threadIdx: location of a thread in a block in terms of (x, y, z)
- Depending upon the problem, 1-, 2-, or 3- dimensions of the elements may be used
- Synchronization after kernel call = waiting on host for kernel to finish: **cudaDeviceSynchronize()**

Example(cudaHello.cu): Say Hello in CUDA

```
#include <stdlib.h>
#include <cuda.h>
#include <stdio.h>

__global__ void kernelHello() {
printf("Saying 'Hello' from block %i, thread %i \n"); //fill in the thread and block Id
}

int main(int argc, char *argv[] )
{
  //read number of blocks and threads from terminal
  int  nblocks = atoi(argv[1]);
  int nthreads = atoi(argv[2]);
  //fill in kernel call

  cudaDeviceSynchronize();

  return 0;
}
```

Full compilable source file: cudaHello_full.cu

Compile and run a CUDA program on CRC

GENCODE
A100: arch=compute_80,code=sm_80
V100: arch=compute_70,code=sm_70
GTX1080: arch=compute_61,code=sm_61
TitanX: arch=compute_52,code=sm_52

After loading gcc, load CUDA module

Nvidia CUDA compiler

Binary

Source code

```
module load cuda/11.0
nvcc -gencode $GENCODE -o <OUTPUT> <SOURCE.cu>
crc-interactive.py -g (Optional: -p <GPUPart>) -t <T> #interactive
./<OUTPUT> <INPUT PARAMETERS>
```

Execute compiled binary on GPU node

Grid-block-thread arithmetic

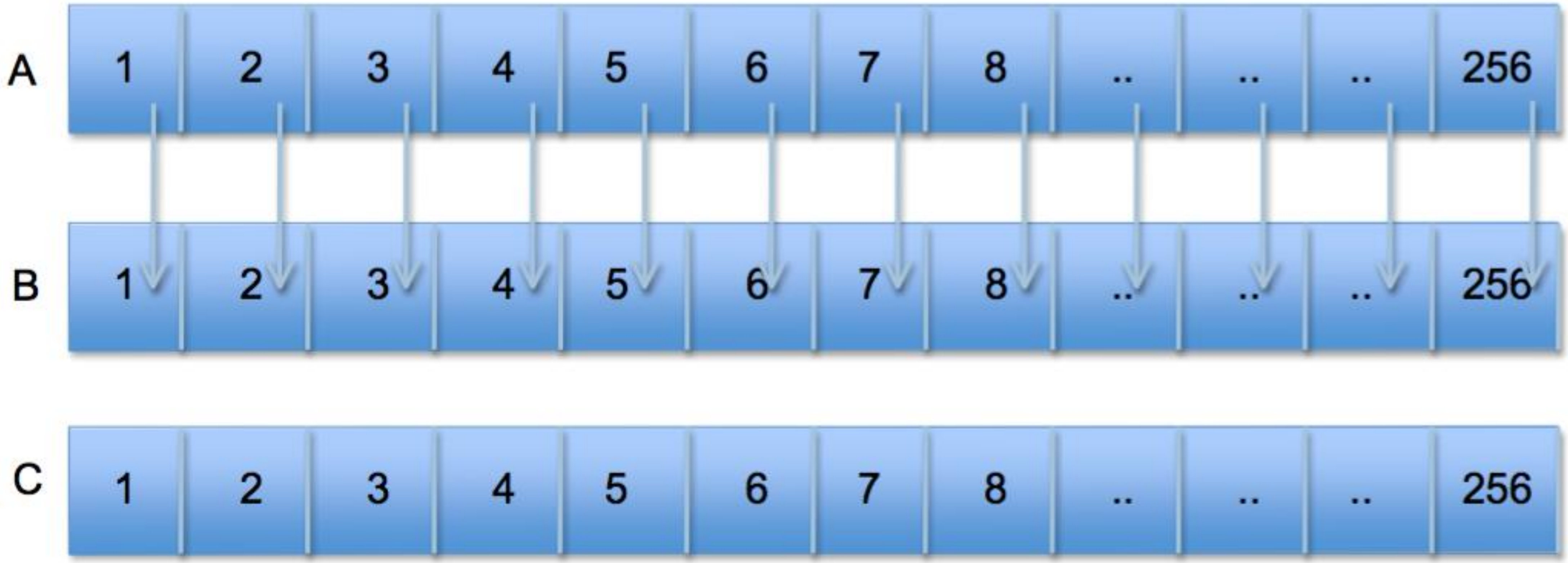
- there are 4 blocks in a grid and 16 threads in a block. How many threads in the grid?
    - 16 * 4 = 64
- Assuming there are 2048 thread and the block size is 512, how many blocks will be in the grid?
    - (ceil)2048/512=4
- given a **one dimensional** grid of size 2 blocks and block size 32 threads, how will you find location of nth thread in terms of blockidx, blockdim and threadidx?
    - **nth thread = threadIdx.x + blockDim.x*blockIdx**

8 parts of a CUDA program

1. Setup inputs on the host (CPU-accessible memory)
2. Allocate memory for inputs on the GPU
3. Copy inputs from host to GPU
4. Allocate memory for outputs on the host
5. Allocate memory for outputs on the GPU
6. Start GPU kernel
7. Copy output from GPU to host
8. Free up all memory upon completion

Newer CUDA versions: Unified memory, only need to be declared and allocated once for GPU and CPU (Steps 2-4)!

Let's see the 8 parts for a common example: vector add



1: Initialize inputs on host

```
double* h_a;
double* h_b;
double* h_c;

int N=256;

h_a = (double*)malloc(N*sizeof(double));
h_b = (double*)malloc(N*sizeof(double));
h_c = (double*)malloc(N*sizeof(double));

for( int i = 0; i < N; i++ ) {
   h_a[i] = <VALUE_a>;
   h_b[i] = <VALUE_b>;
}
```

2: Allocate memory for inputs on GPU

```
double* d_a;
double* d_b;


cudaMalloc(&d_a, N*sizeof (double));
cudaMalloc(&d_b, N*sizeof (double));
```

3: Copy inputs from host to GPU

```
cudaMemcpy( d_a, h_a, N*sizeof(double),
cudaMemcpyHostToDevice);

cudaMemcpy( d_b, h_b, N*sizeof(double),
cudaMemcpyHostToDevice);
```

4: Allocate memory for outputs on host

```
h_c = (double*)malloc(N*sizeof(double));
```

5: Allocate memory for outputs on GPU

```
double* d_c;
cudaMalloc(&d_c, N*sizeof (double));
```

6: Launch GPU kernel

```
//use 1-dimension for both grid and block sizes
int blockSize = 256;
int gridSize = (int) ceil((float) N/blockSize);

// Execute the kernel
add<<<gridSize, blockSize>>> (d_a, d_b, d_c, N);
```

7: Copy output from GPU to host
8: Free memory

```
cudaMemcpy(h_c, d_c, N*sizeof(double),
cudaMemcpyDeviceToHost);


cudaFree(d_c);
free(h_c);
```

The "add" kernel

```
//CUDA kernel. each thread takes care of one
element of c

__global__ void add(double *a, double *b,
                     double *c, int n)
{
  //get our global thread ID
  int id=threadIdx.x + blockIdx.x*blockDim.x;
  //make sure we do not go out of bounds
  if (id < n) c[id] = a[id] + b[id];
}
```

Example(ctest.cu): Prime factor with CUDA

```
int main(int argc, char *argv[] )
{
   char *eptr;
   unsigned long long  Number = strtoull(argv[1],&eptr, 10);
   int nthreads=atoi(argv[2]);
   int nblocks=atoi(argv[3]);//read number of blocks and threads from terminal


   long long unsigned *d_ip;
   long long unsigned*   ip = (long long unsigned*)malloc(sizeof(long long unsigned));
   *ip=0;
   cudaMalloc((void**)&d_ip, sizeof(long long unsigned) );
   cudaMemcpy(d_ip, ip, sizeof(long long unsigned), cudaMemcpyHostToDevice);
   auto start = high_resolution_clock::now();

   //fill in kernel call to check primes

   cudaMemcpy(ip, d_ip, sizeof(int), cudaMemcpyDeviceToHost);
   if(!*ip) cout<<Number<<" is prime! "<<endl;
   else  cout<<Number<<" has prime factor "<<*ip<<endl;


   auto stop = high_resolution_clock::now();
   auto duration = duration_cast<microseconds>(stop - start);

   cout <<"Time in microseconds: "<< duration.count() << endl;
   cudaFree(d_ip);
   free(ip);
   return 0;
}
```

Kernel: Prime factor with CUDA

```
__global__ void isprime_device(long long unsigned *d_ip, long long unsigned Number, long long unsigned sq_n) {

 //Fill in the values for start and end indices, depending on block and thread Id
  long unsigned avgSize = ;
  long unsigned start_index = ;


  long long unsigned istart =   ;
  long long unsigned iend = ;


  long long unsigned  i=istart;
  do //Fill in loop to check for prime factor
  {
    if();
    i++;


  }while( && (i<iend) );


}
```

Full compilable source file: ctest_full.cu

Test GPU and CPU prime factorization

Number to be factorized　　Number of blocks　　Number of threads

- ./chkprime_gpu 1286284145982677551 256 256
- ./chkprime_cpu 1286284145982677551
- Experiment with different block and thread numbers
- Try a smaller number or a composite number with small factor
- Generate large primes online: https://bigprimes.org/
- Similar idea: Search for Hash keys, Crypto-mining

Example: Animation of 2D acoustic pulse

Acoustics are described by linearized Euler equations!

PDE for potential ϕ in 2D cylindrical coordinates (r, θ):

$$\frac{\partial^2 \phi}{\partial t^2} - \left(\frac{\partial^2 \phi}{\partial r^2} + \frac{1}{r}\frac{\partial \phi}{\partial r} + \frac{1}{r^2}\frac{\partial \phi}{\partial \theta^2} \right) = 0$$

Initial conditions for pressure pulse with width 1/a:

$$\phi(t=0) = 0,\ p(t=0) = \frac{\partial \phi}{\partial t}(t=0) = r^2 e^{-ar^2}\cos(2\theta)$$

Exact solution for ϕ:

$$\phi(t,r,\theta) = \int_0^\infty \frac{\omega^2}{(2a)^3} e^{-\omega^2/(4a)} \cdot J_2(\omega r)\sin(\omega t)\cos(2\theta)\, d\omega$$

Exact solution for p:

$$p(t,r,\theta) = \frac{\partial \phi}{\partial t} = \int_0^\infty \frac{\omega^3}{(2a)^3} e^{-\omega^2/(4a)} \cdot J_2(\omega r)\cos(\omega t)\cos(2\theta)\, d\omega$$

Bessel function of order 2

- Create animation with CUDA: Divide 2D domain into 2D grid of blocks, each block containing one single point in domain, **dim3 grid(nxblocks, nyblocks)**
- Divide each block into threads, each thread assigned a specific time
- Calculate values for all (r, θ, t) in parallel: **computeField<<<grid, nthreads>>>**

Example(ccompute.cu): Simulating acoustics

```
int main(int argc, char *argv[] )
{
  char *eptr;
  unsigned long long  Number = strtoull(argv[1],&eptr, 10);

  int nxblocks=atoi(argv[1]);
  int nyblocks=atoi(argv[2]);
  int nthreads=64;
  double t = atof(argv[3]);
  double dt=t/(nthreads-1);
  int ntotal=nxblocks*nyblocks;

  dim3 grid(nxblocks,nyblocks); //generate 2D grid of blocks

  double *d_f;
  double*   f = (double*)malloc(sizeof(double)*ntotal*nthreads);

  cudaMalloc((void**)&d_f, sizeof(double)*ntotal*nthreads );
  cudaMemcpy(d_f, f, sizeof(double)*ntotal*nthreads, cudaMemcpyHostToDevice);

  //Fill in kernel call to calculate field values

  cudaMemcpy(f, d_f, sizeof(double) * ntotal*nthreads, cudaMemcpyDeviceToHost);
  writeOutput(f,nxblocks,nyblocks,nthreads);//write output files
  cudaFree(d_f);
  free(f);
  return 0;
}
```

Digression: Multidimensional representation of grid and block

in the previous slides, we used:

```
int blockSize = 256;
int gridSize = (int) ceil((float) N/blockSize);
```

this is OK since we are want to arrange threads in a 1-dimensional setup.

we could have written the above two lines as follows without any problems:

```
dim3 blockSize(256, 1, 1);
dim3 gridSize((int) ceil((float) N/blocksize.x,1,1);

vecAdd <<<gridSize, blockSize>>> (d_a, d_b, d_c, N);
```

Example(ccompute.cu): Simulating acoustics

```
__device__ void eval(double *d_f, int Id, double r, double costheta,double t)
{
  double omega_i;
  double d_omega=OMEGA_MAX/NMAX;
  for(int i=1;i<NMAX;i++)
  {
    omega_i=d_omega*i;
    d_f[Id]+=powf(omega_i,2)/(powf(2*A,3))*exp(-omega_i*omega_i/(4*A))*jnf(2,omega_i*r)*omega_i*cos(omega_i*t);
  }
  d_f[Id]*=d_omega*costheta;
}


__global__ void computeField(double *d_f, double dt) {
  int totalNx=gridDim.x >> 1;
  int totalNy=gridDim.y >> 1;

  int Id = //fill in computation of linear ID based on block indices

  double t= //fill in time computation based on thread Id;


  int xi=(int)(blockIdx.x - (gridDim.x >> 1));
  int yi=(int)(blockIdx.y - (gridDim.y >> 1));
  double x=(double)xi/totalNx*XMAX;
  double y=(double)yi/totalNy*YMAX;


  double r=sqrt(x*x+y*y);
  double costheta= (r>1.0e-10)? 2*powf(x/r,2)-1: 0;
  d_f[Id]=0;
  //fill in kernel call to evaluate field value
}
```

Full compilable source file: ccompute_full.cu

Compile and run a CUDA program on CRC

GENCODE

A100: arch=compute_80,code=sm_80
V100: arch=compute_70,code=sm_70
GTX1080: arch=compute_61,code=sm_61
TitanX: arch=compute_52,code=sm_52

Nvidia CUDA compiler

After loading gcc, load CUDA module

Binary

Source code

```
module load cuda/11.0
nvcc -gencode $GENCODE -o <OUTPUT> <SOURCE.cu>
crc-interactive.py -g (Optional: -p <GPUPart>) -t <T> #interactive
./<OUTPUT> <INPUT PARAMETERS>
```

Execute compiled binary on GPU node

Run and view animation

x, y Resolution   Total time

- ./simulatePulse 256 256 3.0
- On VIZ node:
  - module load python/3.7.0
  - python animate2D.py

Summary/Recap

- GPUs offer a massive amount of processing power capacity
- CUDA is the programming language used to program Nvidia GPUs
- Efficient mapping of problem to GPUs key to performance
- Understanding memory allocation helps improve performance

Outlook/Advanced Topics

- Communication and synchronization between threads/blocks (data exchange, reduction)
- Multi-streaming
- Use of external CUDA libraries
- Multiple GPUs/CPUs: CUDA/MPI/OpenMP
- GPU peer to peer communication: GPUDirect RDMA

Credits

- Oak Ridge National Laboratory tutorials
- Nvidia Documentation
- GPU courses at universities
    - Caltech
    - UWisconsin
    - UChicago

Thank You for Your Attention !
Questions?