GPU Programming Workshop

Session 1: Introduction to OpenACC and CUDA

Special Technical Projects Team

Feb 2, 2021

UCAR

Overview

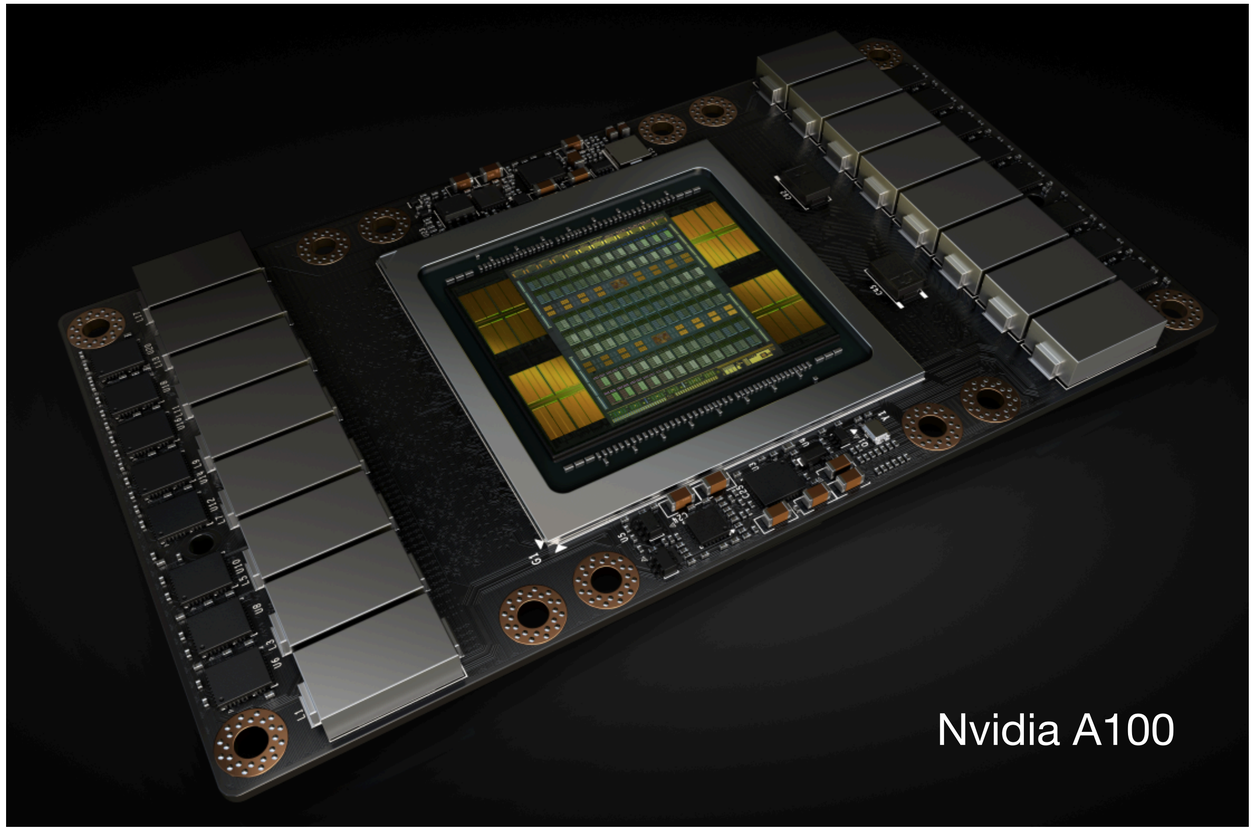
- Introduction to GPU architecture, key concepts, and terminologies
- OpenACC programming model
  - Coding practice
- CUDA programming model
  - Coding practice
- OpenACC and CUDA coding practice (naive matrix multiplication)

Before going further, clone the workshop code, slides, and reference materials using:
git clone https://github.com/NCAR/GPU_workshop.git

Introduction and GPU Architecture

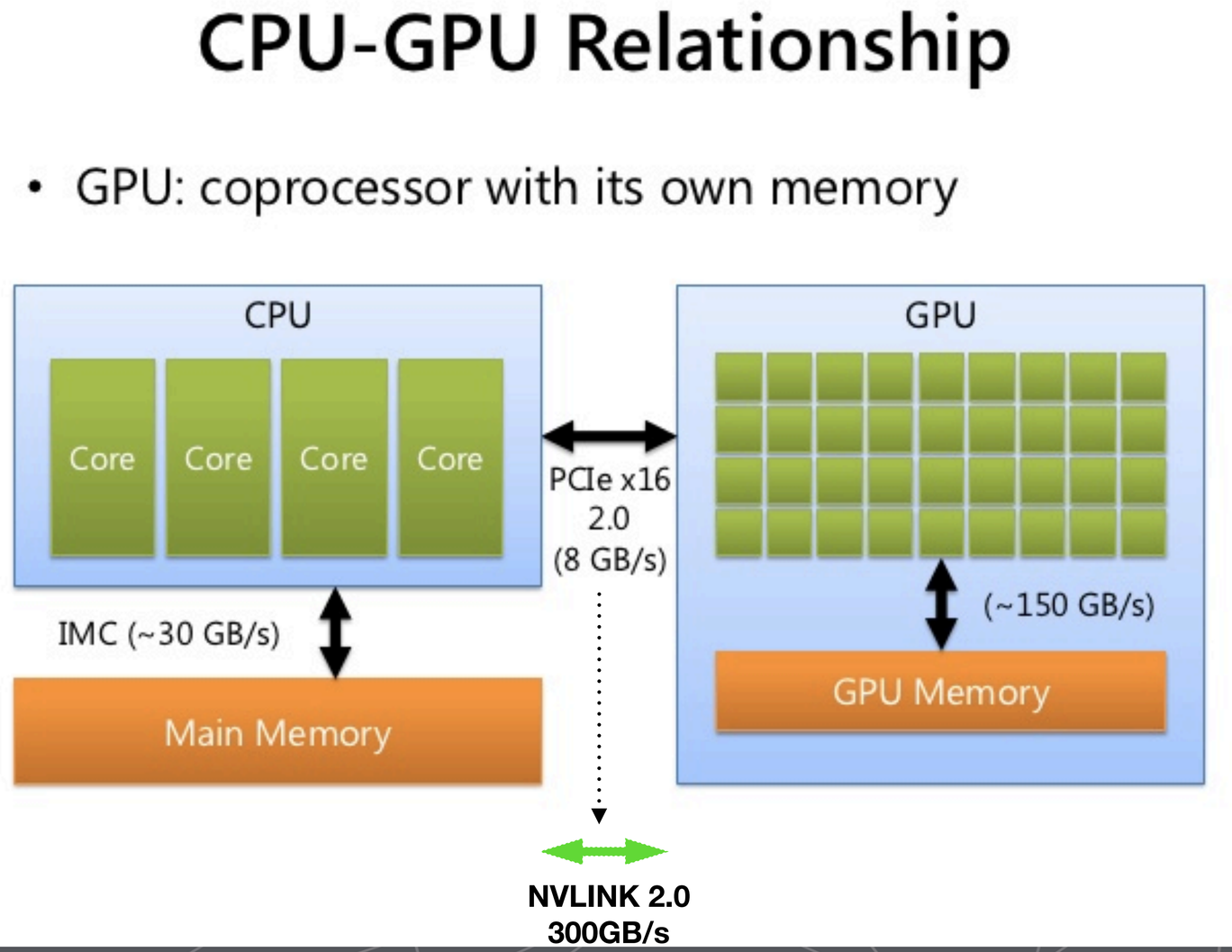
Why GPUs?

- Classic CPU single-threaded performance reached its limit
- GPUs naturally can handle massively parallel compute loads at high bandwidth
- GPUs is a **throughput-oriented**, many-core architecture, with thousands of cores bundled into dozens of SIMD multiprocessors
- Operational latencies for executions are higher on the GPU than on the CPU because of clock difference
- Operational latencies must be covered by thread-level and instruction-level parallelism
- GPUs have their own RAM, which can be accessed by the host



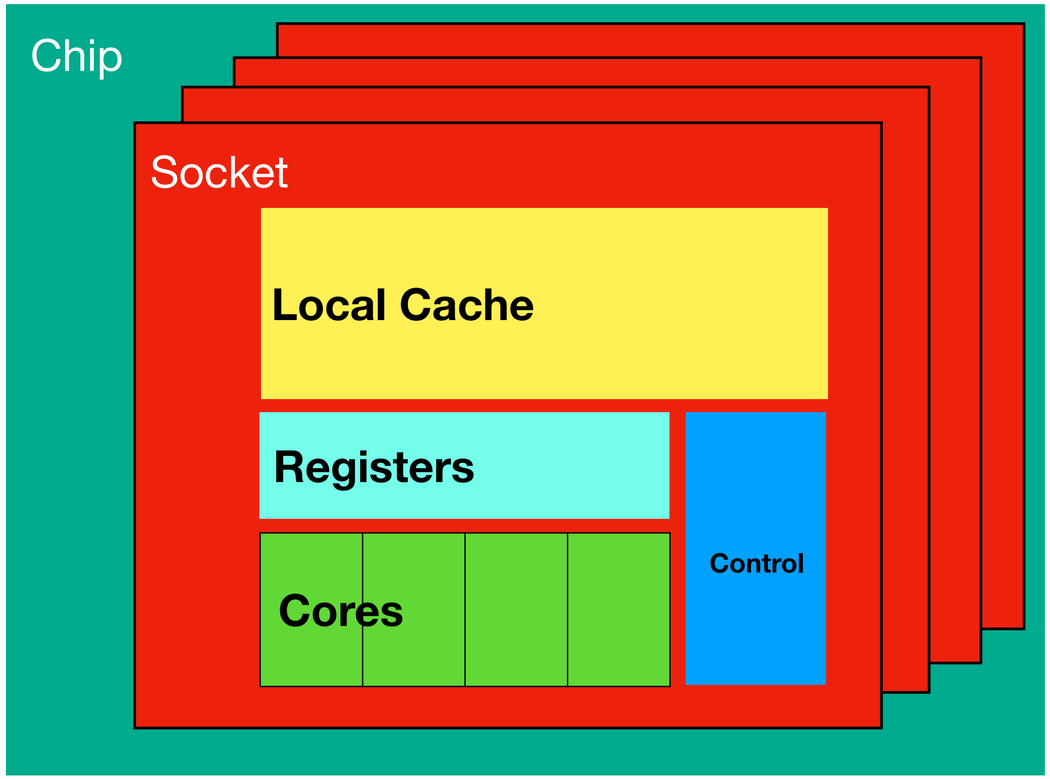
GPU Node Architecture

- Transfer between the chip and the GPU is conducted over PCI express bus or NvLink (Power systems)
- The third version of PCI-3
  - Delivers 32GB/s per lane for 'x' number of lanes
- The more lanes, the faster the transfer

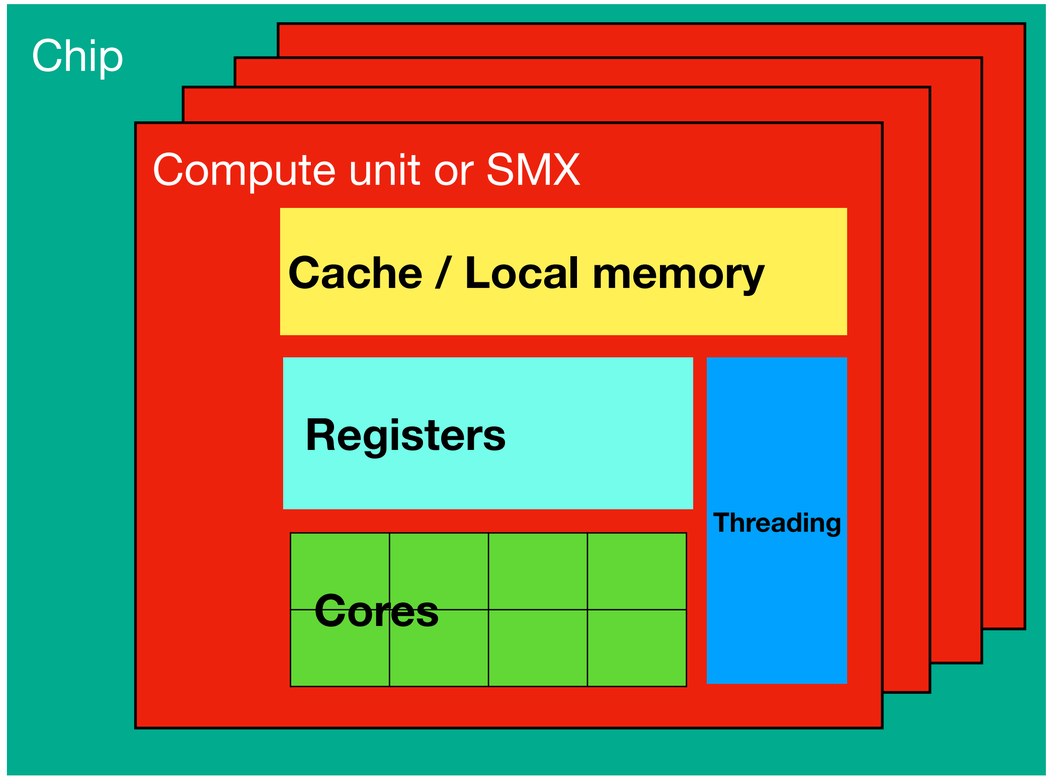


CPU and GPU are Designed Very Differently

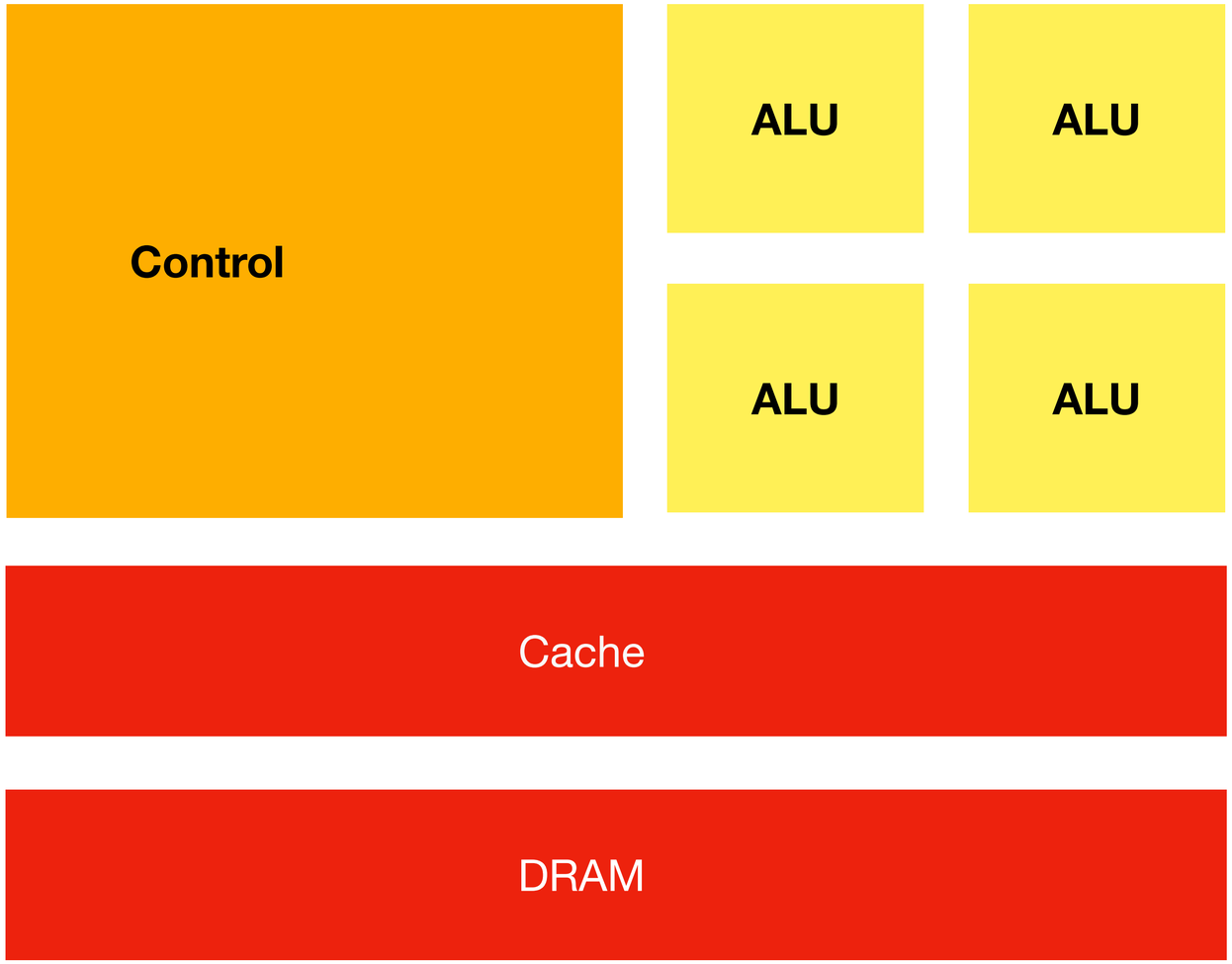
CPU
Latency Oriented Cores



GPU
Throughput Oriented Cores

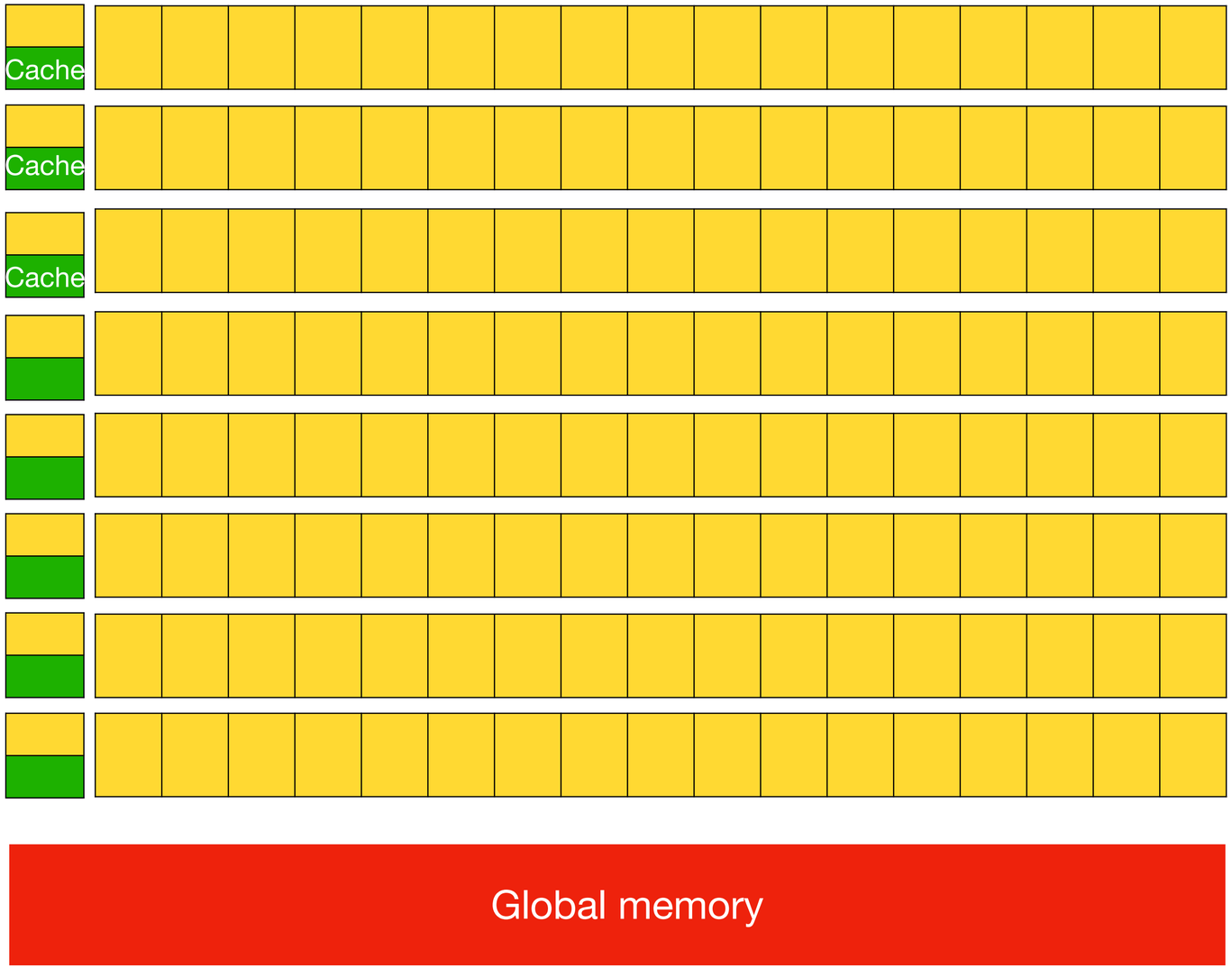


CPUs: Latency Oriented Design



- Powerful ALU
  - Great for serial operations
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency (Data prefetch)

GPUs: Throughput Oriented Design

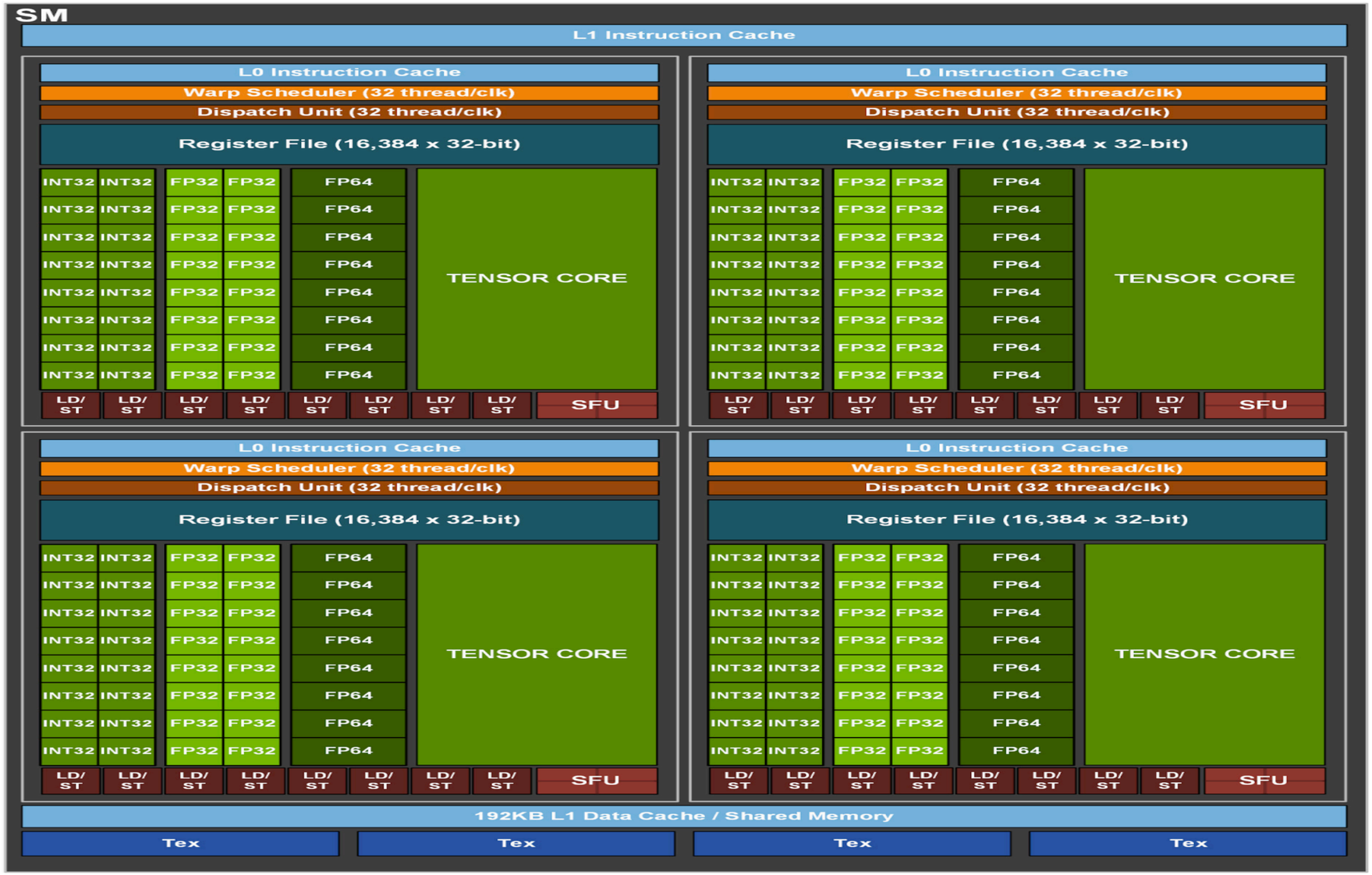


- Simple but a very large number of cores devoted to Floating Point operations.
- Small caches to boost memory throughput
- Simple control
  - No branch prediction
  - No data prefetching
- Require massive number of threads to hide operational latencies
  - Threading logic

NVIDIA Ampere A100

https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

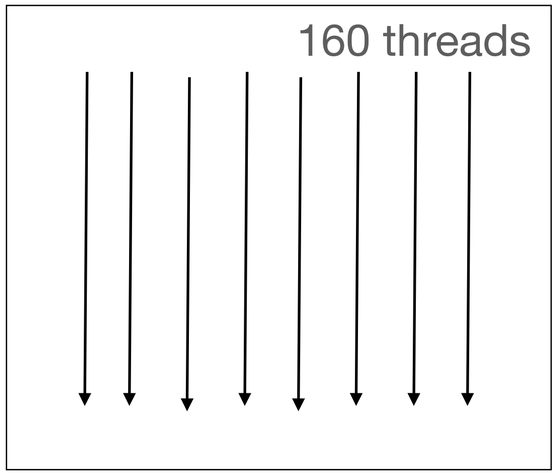
Ampere SMX architecture



Warps

- A Warp is a collection of 32 threads
- GPU always executes all 32 threads in a Single Instruction Multiple Threads (SIMT) approach
- Due to SIMT all threads in a warp normally execute the same instruction in the same single clock cycle

Logical view



Thread block

Hardware view

32 threads
32 threads
32 threads
32 threads
32 threads

Execution view

Warp scheduler			
SP	SP	SP	SP
SP	SP	SP	SP
SP	SP	SP	SP
SP	SP	SP	SP
SP	SP	SP	SP

CPU vs GPU Threads & Register Usage per Thread

CPU threads	GPU threads
• Heavy weight threads • Plenty of resources per thread • Limited number of threads can be launched because number of cores are limited • Data cached through L1, L2 and L3 cache	• Light weight threads • Very limited resources per thread • A very high number of threads can be launched • Data cached through Shared memory and L2 cache • Requires special attention to number of registers allocated per thread

Software Abstraction

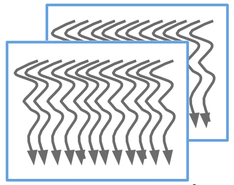
Software term

Threads

Kernel is executed by threads processed by CUDA Core

Vector (OpenACC)

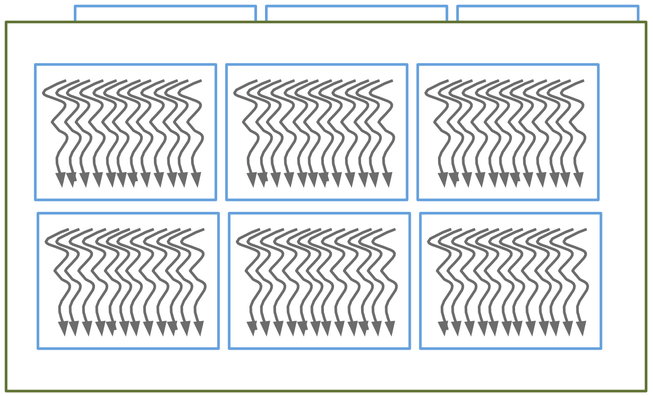
Blocks



512–1024 threads / block

Maximum 8 blocks per SM
32 parallel threads are executed at the same time in a *WARP*

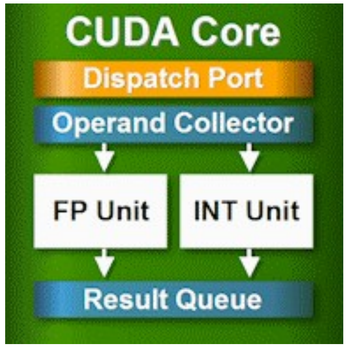
Grids

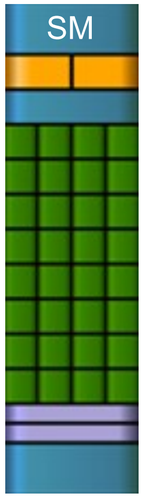


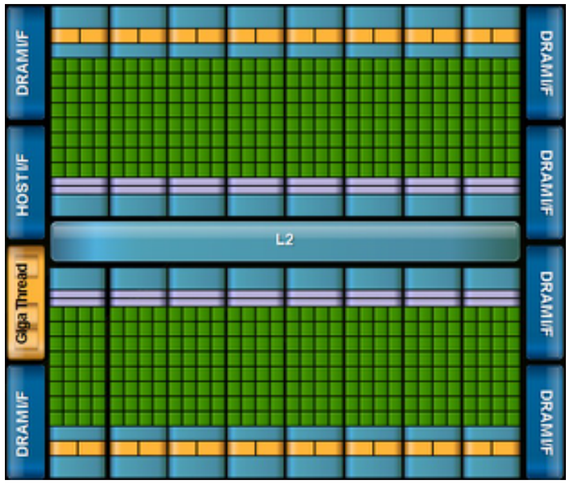
gangs (OpenACC)

One grid per kernel with multiple concurrent kernels

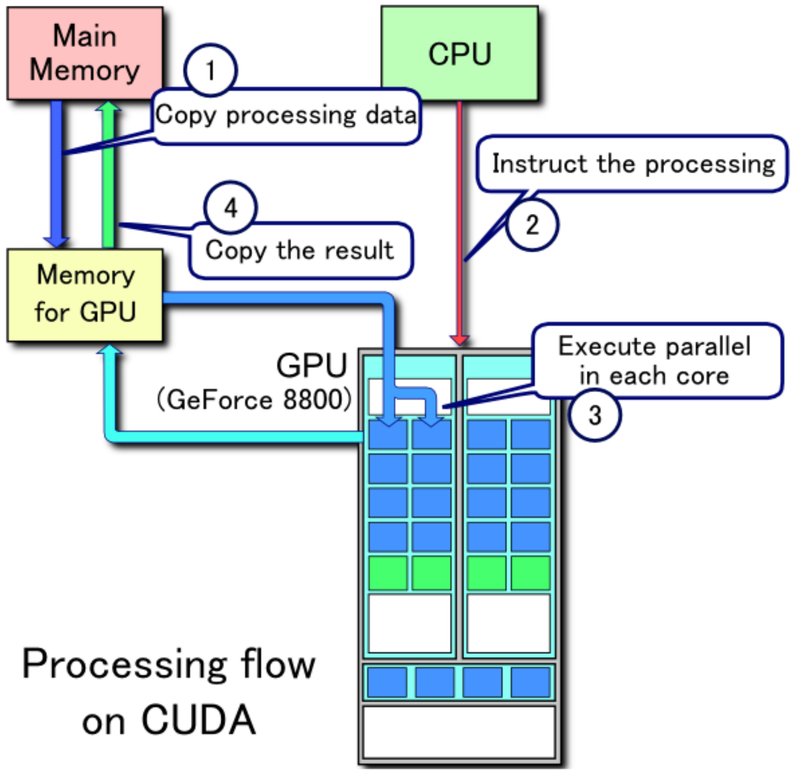
Hardware term







Developing a GPU Program



https://en.wikipedia.org/wiki/CUDA

Example of CUDA process flow*

1. Copy data from main memory to GPU memory
2. CPU initiates threads on the GPU
3. GPU executes parallel code
4. Copy the results from GPU memory to main memory

* This flow occurs in OpenACC as well, we just have to give more specifics in CUDA

File Structure

```
Lesson_#_Name/
├── CUDA
│   ├── exercise
│   │   ├── build.sh
│   │   ├── common.cpp
│   │   ├── cuda_file.cu
│   │   ├── functions.cpp
│   │   ├── main.cpp
│   │   ├── Makefile
│   │   ├── pch.h
│   │   └── submit.sh
│   └── solution
└── OpenACC
    ├── exercise
    │   ├── build.sh
    │   ├── common.cpp
    │   ├── functions.cpp
    │   ├── main.cpp
    │   ├── Makefile
    │   ├── openacc_file.cc
    │   ├── pch.h
    │   └── submit.sh
    └── solution
```

To run the exercise:

1. cd into desired directory
2. ./build.sh
3. sbatch submit.sh

- Each lesson has two sub-directories:
  - **CUDA** - Contains the CUDA version of the code
  - **OpenACC** - Contains the OpenACC version of the same code
- Both CUDA and OpenACC directories have at least these two sub-directories:
  - **Exercise** - This is the code that you will be working on.
  - **Solution** - This is the solution code.
- Within each sub-directory is a similar set of files:
  - **build.sh** - "./build.sh" will load required modules and then call the **make clean** and **make** commands with the Makefile to compile all files and create an executable.
  - **common.cpp** - Includes all the common functions for initializing matrices, printing matrices, and verifying the result against the CPU execution.
  - **language specific file** : It is either .cu extension for CUDA or .cc extension for OpenACC. This file contains the code to be executed on the GPU.
  - **functions.cpp** - Contains the code to be run on the host.
  - **Makefile** - Compiles all files using appropriate flags and creates the desired executable. Differs slightly between CUDA and OpenACC in terms of compiler flags. It can also be used to clean the object and executable files.
  - **main.cpp** - Calls both the host and device routines and calculates the time taken for each.
  - **pch.h** - Header file that contains all function declarations from all other files.
  - **submit.sh** - submit shell script that submits the job. To be used as "sbatch submit.sh".

OpenACC

OpenACC Introduction

- OpenACC is a directives-based parallel programming model that is designed for performance and portability
- Enables easy conversion of sequential CPU based code to GPUs by adding pragmas
- General syntax:
  - C - #pragma acc [directive] [clauses]
  - FORTRAN - !$acc [directive] [clauses]
- Multiple directives can be chained together in the same line and applied to the same blocks of code
- There is no need for explicit allocation/copying of data to and from the GPU in OpenACC

OpenACC Directives/Constructs

Construct	C code	Description
Kernels	#pragma acc kernels [clauses]	Surrounds loops to be executed on the GPU
Parallel	#pragma acc parallel [clauses]	Launches a number of gangs in parallel each with a number of workers, each with vector or SIMD operations
Loop	#pragma acc loop [clauses]	Applies to the immediately following loop or nested loops and describes the type of parallelism to execute these loops on the GPU
Data	#pragma acc data [clauses]	Defines a region of the program within which data is accessible by the GPU
Host data	#pragma acc host_data [clauses]	Makes the address of the device data available on the host
Cache	#pragma acc cache (list)	Added to the top of the loop. The elements in the list are cached in the software managed data cache
Update	#pragma acc update [clauses]	Copies data between the host memory and the data allocated on the GPU memory, and vice-versa
Clause - Collapse	#pragma acc loop collapse(n)	The collapse clause is used to specify how many tightly nested loops are associated with the loop clause.
Clause - Reduction	#pragma acc loop reduction(+:temp)	The reduction clause specifies a reduction operator and one or more vars.

OpenACC Introduction

- A vector is a group of threads that execute an operation on a chunk of data at once
- A worker executes one vector or SIMD operation. Workers in the same gang are able to share resources such as memory
- Gangs are a group of workers which in turn execute vectorized / SIMD operations

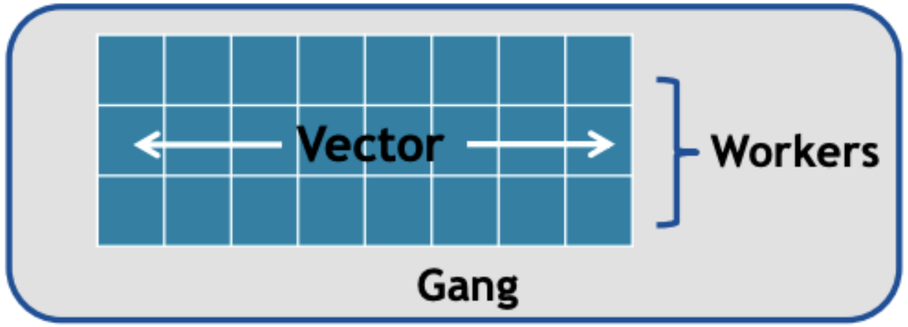


Image from https://www.olcf.ornl.gov tutorial slides

Kernels vs Parallel construct

- Kernels construct - The compiler is able to detect parts of the code that can be parallelized and parallelizes it. Sometimes it can parallelize the code better than the programmer.
- The downside is the compiler is conservative, and may not parallelize code that it deems unsafe

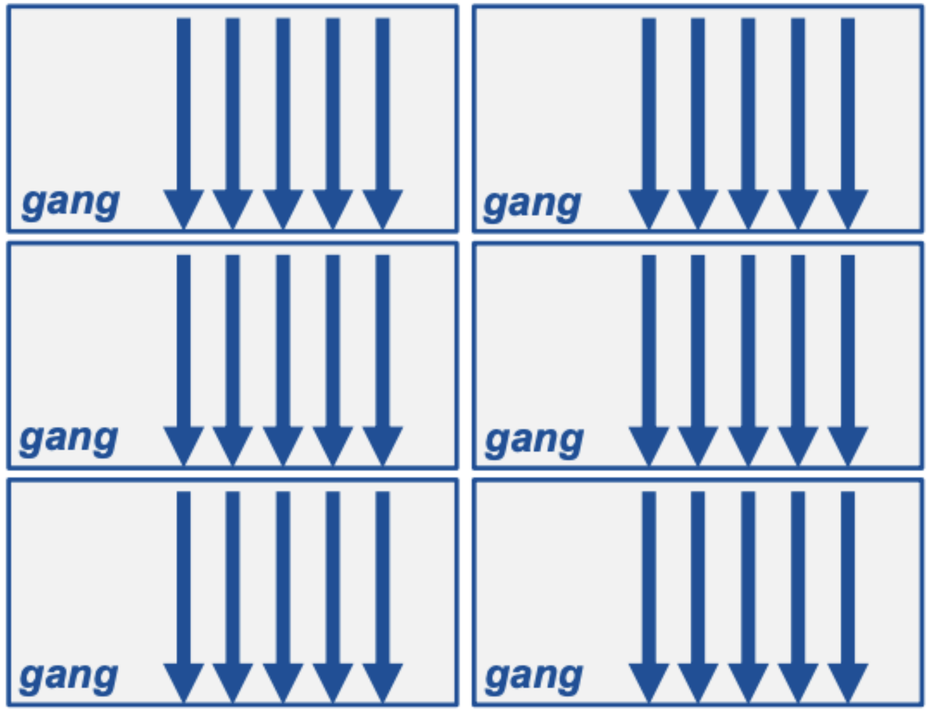
OpenACC Introduction

When you use "#pragma acc parallel", multiple gangs of workers are launched. The "#pragma acc loop" directive tells the compiler how to evenly split the work across the gangs.

C/C++

```
#pragma acc parallel
{
  #pragma acc loop
  for(int i = 0; j < N; i++)
    a[i] = 0;
}
```

- parallel directive tells the compiler to launch one or more parallel gangs.
- without the loop directive each of the gangs will run the same loop redundantly which is not efficient.
- when the compiler encounters the loop directive it parallelizes the block of code immediately after it, and splits the work across each of the gangs equally.



Data construct

- *#pragma acc data [clauses]* defines a region below it within which the data is accessible by the GPU.

Clause	Description
copy(list)	1. Allocates the data in list on the GPU 2. Copies the data from the host to the GPU when entering the region 3. Copies the data from the GPU to the host when exiting the region
copyin(list)	Allocates the data in list on the GPU and copies the data from the host to the GPU when entering the region.
copyout(list)	Allocates the data in list on the GPU and copies the data from the GPU to the host when exiting the region.
create(list)	Allocates the data in list on the GPU, but does not copy data between the host and device.
present(list)	The data in the list must be already present on the GPU from some containing data region, which then can be found and used.

The information in this table and more about different directives and clauses at:
https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf

Matrix Addition with OpenACC

Matrix Addition CPU Base Code

```
// Calculate A+B=C on the host
void cpu_matrix_add(const float *A, const float *B, float *C, const int rows, \
const int cols) {
  for (int i = 0; i < rows; i++){
    for (int j = 0; j < cols; j++) {
      int idx = i*cols+j;
      C[idx] = A[idx] + B[idx];
    }
  }
}
```

- Element wise addition of matrices A and B, which is then stored in C
- A and B are both linear arrays. We calculate the index of the element in the linear array based on the row and column in the matrix.
- rows = number of rows in the matrix
- cols = number of columns in the matrix
- size of A, B, C = rows*cols

OpenACC Matrix Addition

- **C**
  - **#pragma acc data copyin(data[0:numElements]) copyout(data[0:numElements])**
  - **#pragma acc parallel loop collapse(N)**

- ***FORTRAN***
  - ***!$acc data copyin(data list) copyout(data list)***
  - ***!$acc parallel loop collapse(N)***

OpenACC Makefile

```
# nvc++ is the compiler program used to compile all files
CC=nvc++

# Compilation flags
# -acc tells the compiler to look for OpenACC hints and compile those regions differently
# -gpu tells the compiler that the target architecture is a GPU
#     "cc60,cc70" generate code for Pascal (compute complexity 6.0) and Volta GPU architectures
# -Minfo=all has the compiler print out all info related to compilation
# -I${NVHPC_ROOT_PATH}/include sets the path to the header files
ACCFLAGS= -acc -gpu=cc60,cc70,managed -Minfo=all -I{NVHPC_ROOT_PATH}/include

# Resulting compilation : nvc++ - c $(ACCFLAGS) openacc_filename.cc
%.o: %.cc Makefile
		$(CC) -c $(CFLAGS) $(ACCFLAGS) $<
```

For the purposes of the tutorials we just use **make clean** to remove files from a previous build and **make** to compile the object files and executable

Exercise 1 - Matrix Addition on GPU Using OpenACC

- Instructions to follow while implementing the Matrix Addition. Use directives and clauses discussed to modify the code in matrix_add.cc to tell the compiler to:
    1. Allocate memory, copy matrices from host to GPU, and copy result back from GPU to host.
    2. Parallelize loops used for adding the two matrices.

The information about different directives and clauses can be found at:
https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf

To build use ./build.sh

To submit use sbatch submit.sh

Adjust the size of the matrices within by editing the last line of submit.sh

Exercise 1 : OpenACC C/C++ Matrix Addition Check

1. What kind of GPU did you use to offload the computations, and how much memory does it have available?

2. What compiler did we use to build the code?

3. Did the GPU execution validate for non-square matrices?

4. What was the CPU execution time for adding two 1024x1024 matrices? For size 16384x16384?

5. What was the GPU execution time for adding two 1024x1024 matrices? For size 16384x16384?

6. Rebuild and run without the "-g" debug flags, and check the times again.

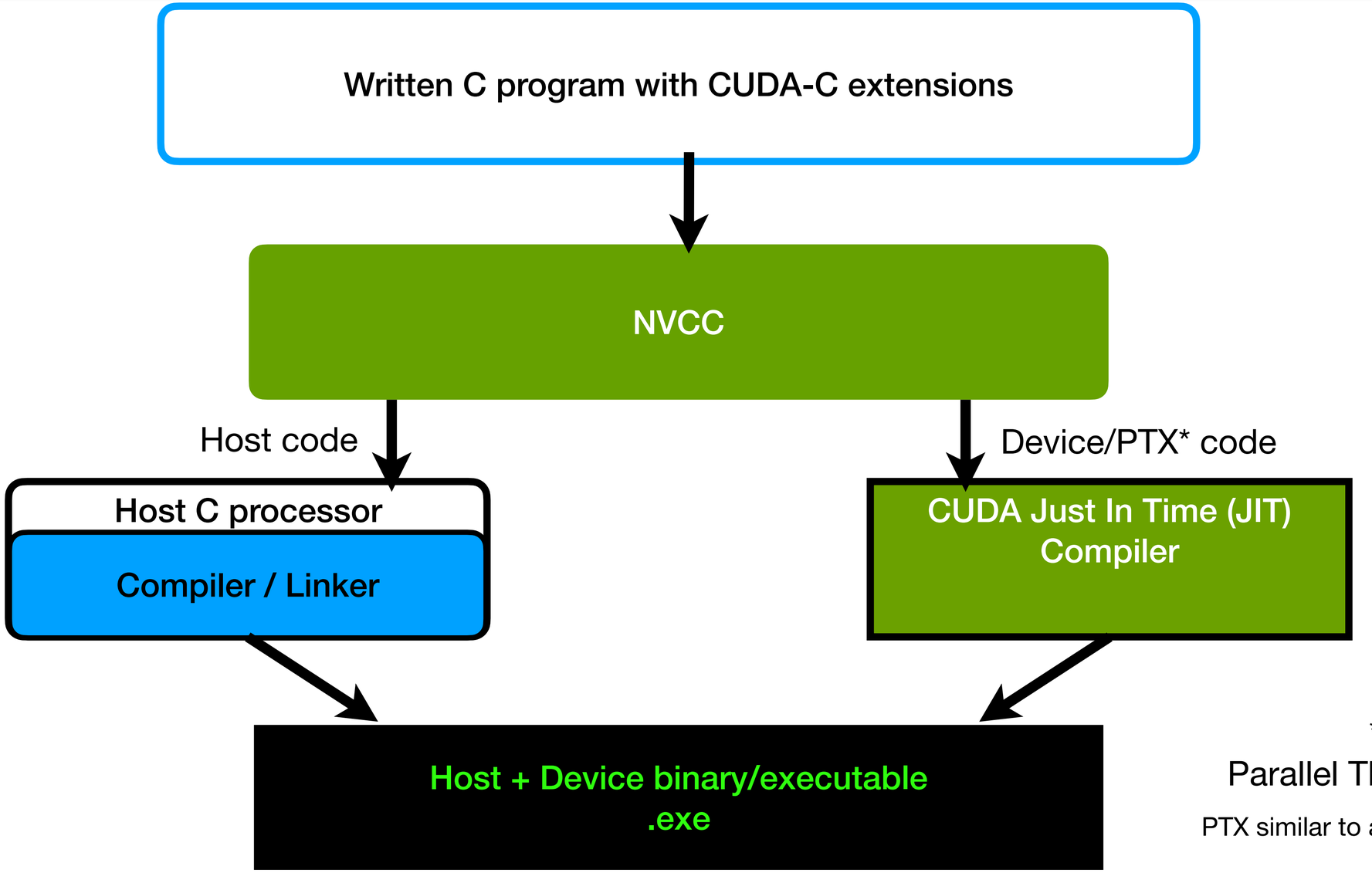
Matrix Addition with CUDA

CUDA Makefile

```
# nvcc is the compiler program used to compile CUDA-C
CUDACC=nvcc

# Compilation flags
# std=c++11 defines the c++ standard version being used
# -O3 sets the compiler's optimization level to 3 which prioritizes reducing execution time
# -I${CUDA_ROOT_PATH}/include sets the path to the header files
CUDACFLAGS= -g -std=c++11 -O3 -I${CUDA_ROOT_PATH}/include

# Linking Flags (used only when making the binary)
# -L sets the path to the library files
CUDALDFLAGS=-L${CUDA_ROOT_PATH}/lib64 -lcudart

# Resulting compilation : nvcc - c $(CUDACFLAGS) cuda_filename.cu
%.o: %.cu Makefile
        $(CUDACC) -c $(CUDACFLAGS) $CUDALDFLAGS $<
```

For the purposes of the tutorials we use “**make clean**” to remove files from a previous build and “**make**” to compile the object files and executable

CUDA Compilation



*PTX =
Parallel Thread Execution

PTX similar to assembly code on CPU

Exercise 2 - CUDA Matrix Addition

Add commands to:

- Allocate memory on GPU using cudaMalloc function
- Copy the matrices from host to device using cudaMemcpy function
- Decide on the thread layout (grid and block dimensions)
- Launch the kernel
- Logic to add two vectors on GPU
- Block the CPU until GPU execution is finished

Syntax help for above exercise

https://kapeli.com/cheat_sheets/CUDA_C.docset/Contents/Resources/Documents/index

Exercise 2 - CUDA Matrix Addition

Add commands to:

- Allocate memory on GPU using cudaMalloc function
- Copy the matrices from host to device using cudaMemcpy function
- Decide on the thread layout (grid and block dimensions)
- Launch the kernel
- Logic to add two vectors on GPU
- Block the CPU until GPU execution is finished

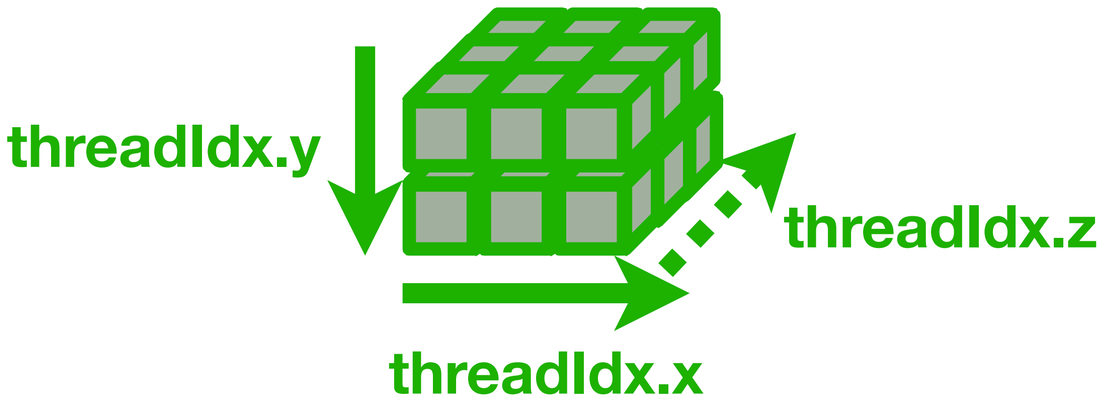
Syntax help for above exercise

https://kapeli.com/cheat_sheets/CUDA_C.docset/Contents/Resources/Documents/index

Understanding CUDA Blocks and Grids

Matching Threads with Data

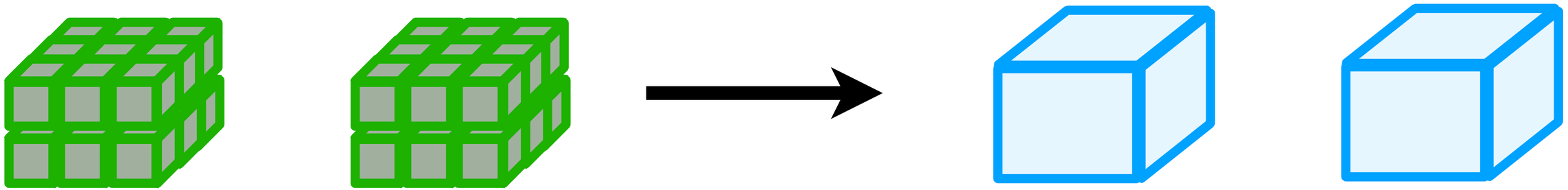
Threads on the GPU are grouped into thread **Blocks** and then a **Grid** of blocks. A thread is uniquely identified within a block based on its index in each dimension: **threadIdx.x**, **threadIdx.y**, and **threadIdx.z**.



Matching Threads with Data

Threads on the GPU are grouped into thread **Blocks** and then a **Grid** of blocks. A thread is uniquely identified within a block based on its index in each dimension: **threadIdx.x**, **threadIdx.y**, and **threadIdx.z**.

A **Block** is executed by one streaming multiprocessor and has 3 dimensions: **blockDim.x**, **blockDim.y**, and **blockDim.z**. A block is uniquely identified within a grid by its index in each dimension: **blockIdx.x**, **blockIdx.y**, and **blockIdx.z**.



2 sets of threads grouped together

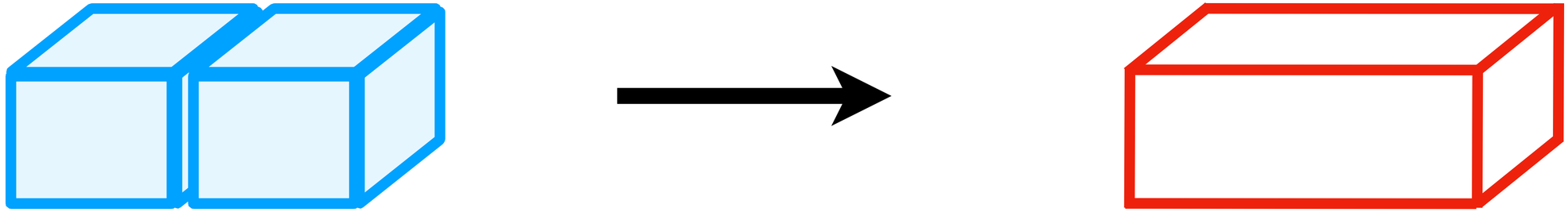
Becomes 2 blocks each with **blockDim** = **{x,y,z}** = {3,2,3}

Matching Threads with Data

Threads on the GPU are grouped into thread **Blocks** and then a **Grid** of blocks. A thread is uniquely identified within a block based on its index in each dimension: **threadIdx.x**, **threadIdx.y**, and **threadIdx.z**.

A **Block** is executed by one streaming multiprocessor and has 3 dimensions: **blockDim.x**, **blockDim.y**, and **blockDim.z**. A block is uniquely identified within a grid by its index in each dimension: **blockIdx.x**, **blockIdx.y**, and **blockIdx.z**.

A **Grid** is the largest set that groups blocks together. One grid is launched per kernel (CUDA call) and has 3 dimensions: **gridDim.x**, **gridDim.y**, and **gridDim.z**



The 2 blocks with each
blockDim = **{x,y,z}** = {3,2,3}

Becomes 1 grid with
gridDim = **{x,y,z}** = {2,1,1}

Matching Threads with Data

A thread is uniquely identified within a block based on its index in each dimension: **threadIdx.x**, **threadIdx.y**, and **threadIdx.z**.

A block is uniquely identified within a grid by its index in each dimension: **blockIdx.x**, **blockIdx.y**, and **blockIdx.z**.

To uniquely identify a thread within a grid a **blockIdx** needs to be converted to an offset in terms of threads using **blockDim** and then combined with the **threadIdx**.

column_t = **blockIdx.x** * **blockDim.x** + **threadIdx.x**
row_t = **blockIdx.y** * **blockDim.y** + **threadIdx.y**
aisle_t = **blockIdx.z** * **blockDim.z** + **threadIdx.z**

These column, row, and aisle values are used with the dimension of the data being operated on to calculate the global linear address of a thread.

Matching Threads with Data

Data on the GPU

Grid, **Blocks**, and **Threads** on GPU

```
add_matrices<<<gridDim, blockDim>>>
```

Matching Threads with Data

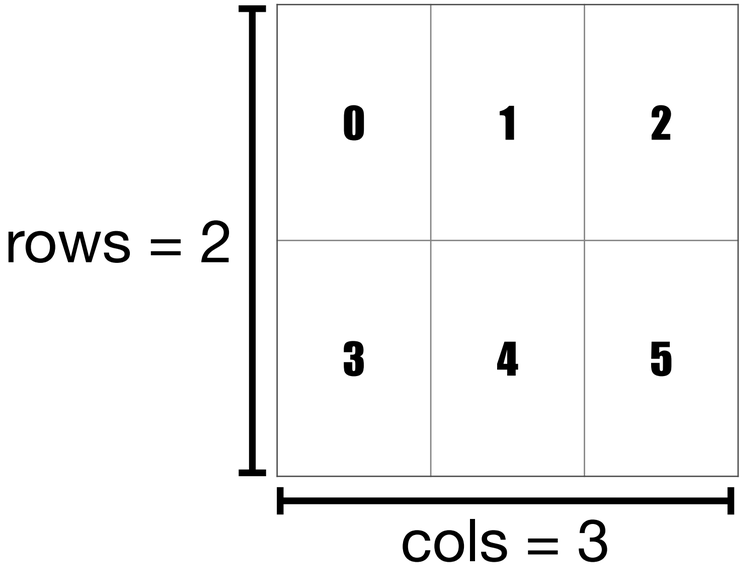
Data on the GPU

3.0	3.0	3.0
3.0	3.0	3.0

Grid, **Blocks**, and **Threads** on GPU

```
add_matrices<<<gridDim, blockDim>>>
```

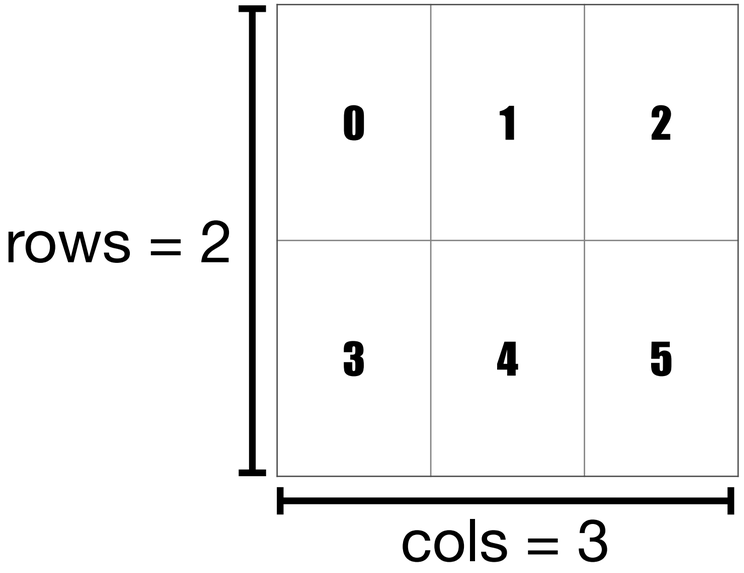
Matching Threads with Data



Grid, **Blocks**, and **Threads** on GPU

```
add_matrices<<<gridDim, blockDim>>>
```

Matching Threads with Data

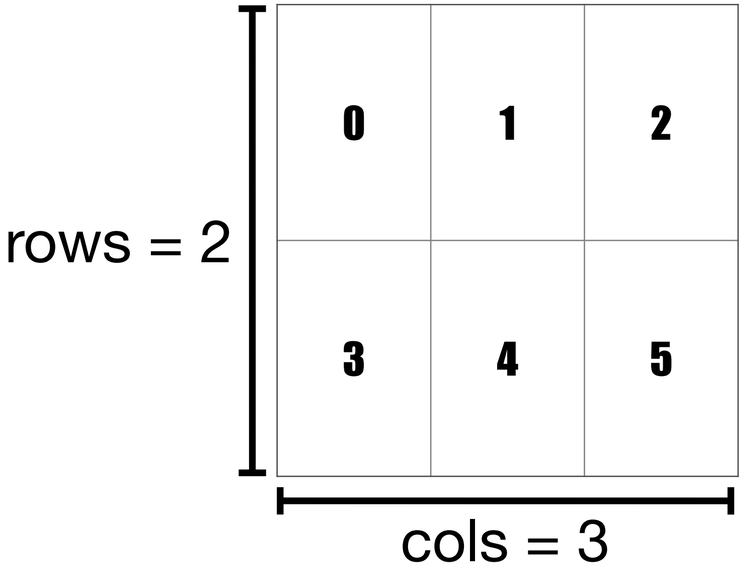


When the GPU kernel is launched, it will be assigned to one **Grid** with a 3D set of **Blocks** inside and a 3D set of **Threads** inside the blocks. For our examples with a matrix, we will always have **gridDim.z** = **blockDim.z** = 1.

Grid, **Blocks**, and **Threads** on GPU

```
add_matrices<<<gridDim, blockDim>>>
```

Matching Threads with Data



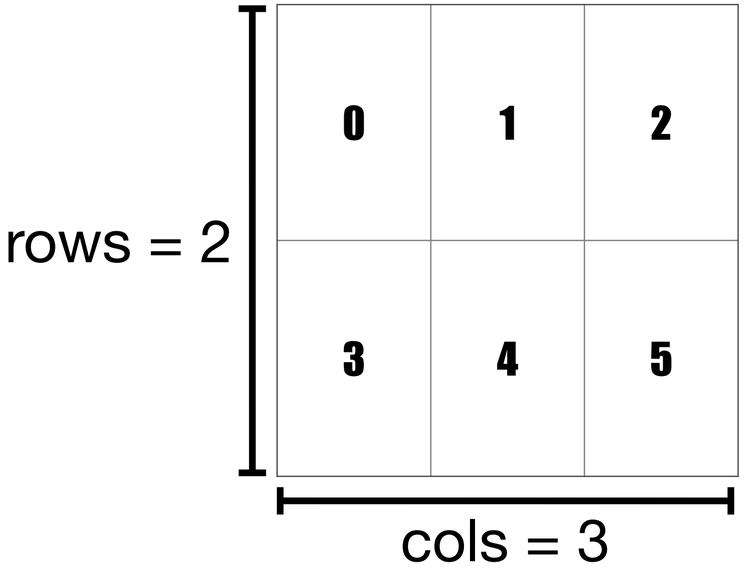
When the GPU kernel is launched, it will be assigned to one **Grid** with a 3D set of **Blocks** inside and a 3D set of **Threads** inside the blocks. For our examples with a matrix, we will always have **gridDim.z** = **blockDim.z** = 1.

To find the layout of blocks, first we chose the layout of the threads. The number of threads in the x-direction will be **blockDim.x** and the number of threads in the y-direction is the **blockDim.y**.

Grid, **Blocks**, and **Threads** on GPU

```
add_matrices<<<gridDim, blockDim>>>
```

Matching Threads with Data

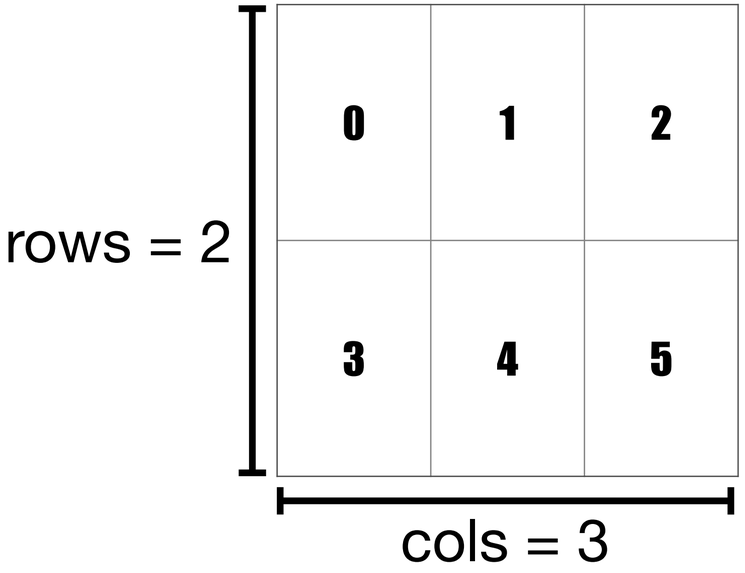


We want to choose **blockDim.x** and **blockDim.y** so the blocks map easily onto the matrix.

Grid, **Blocks**, and **Threads** on GPU

```
add_matrices<<<gridDim, blockDim>>>
```

Matching Threads with Data



We want to choose **blockDim.x** and **blockDim.y** so the blocks map easily onto the matrix.

We have options* here, but we will use:

blockDim.x = 1
blockDim.y = 2

blockDim = {x,y} = {1,2}

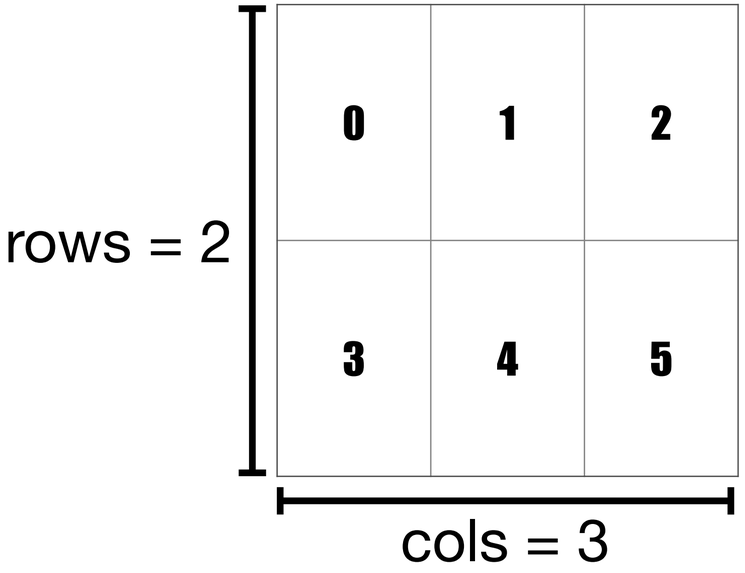
So, a block will map to a column of the matrix.

*Typically choose powers of 2

Grid, **Blocks**, and **Threads** on GPU

```
add_matrices<<<gridDim, blockDim>>>
```

Matching Threads with Data



We want to choose **blockDim.x** and **blockDim.y** so the blocks map easily onto the matrix.

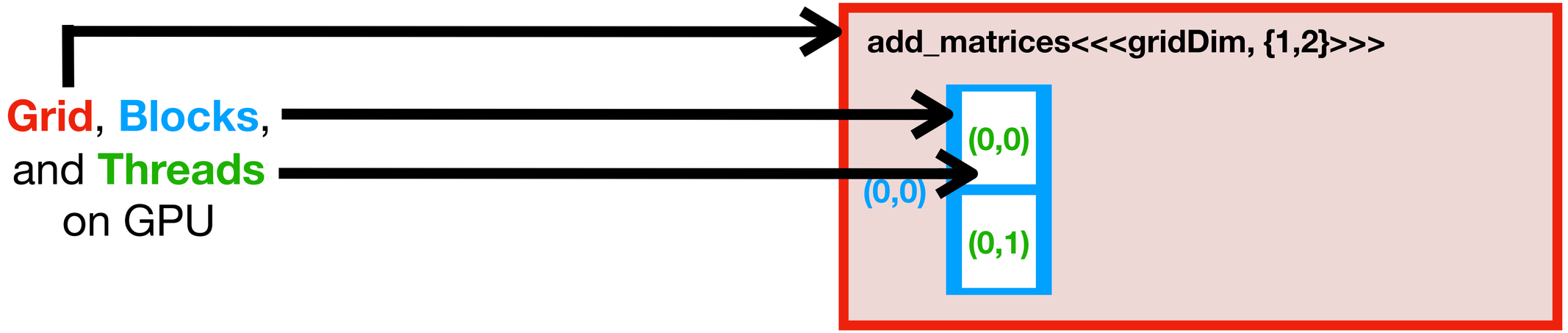
We have options* here, but we will use:

blockDim.x = 1
blockDim.y = 2

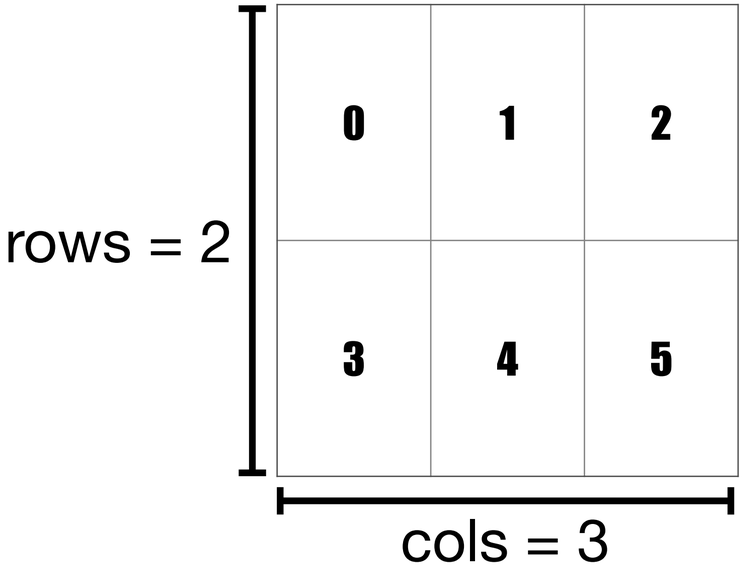
blockDim = {x,y} = {1,2}

So, a block will map to a column of the matrix.

*Typically choose powers of 2



Matching Threads with Data

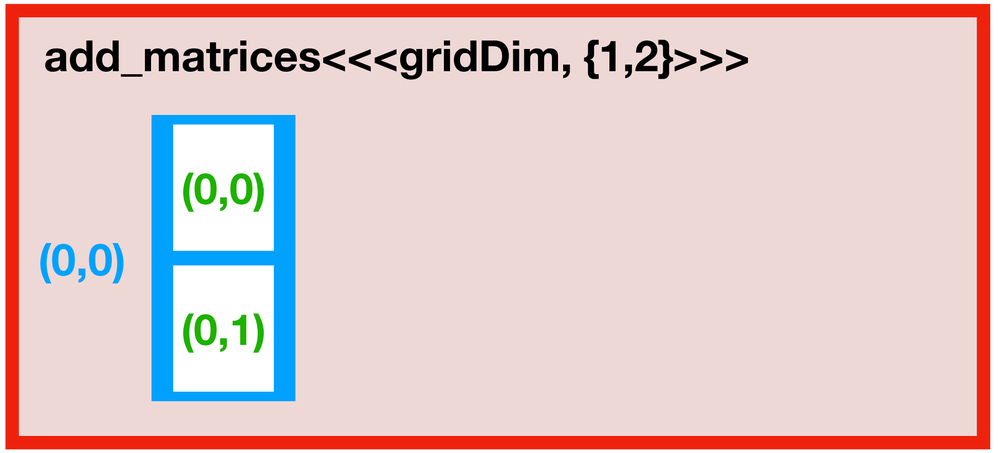


Once the block dimensions are decided, the **gridDim** can be calculated. These are based on the dimensions of the matrix being worked on.

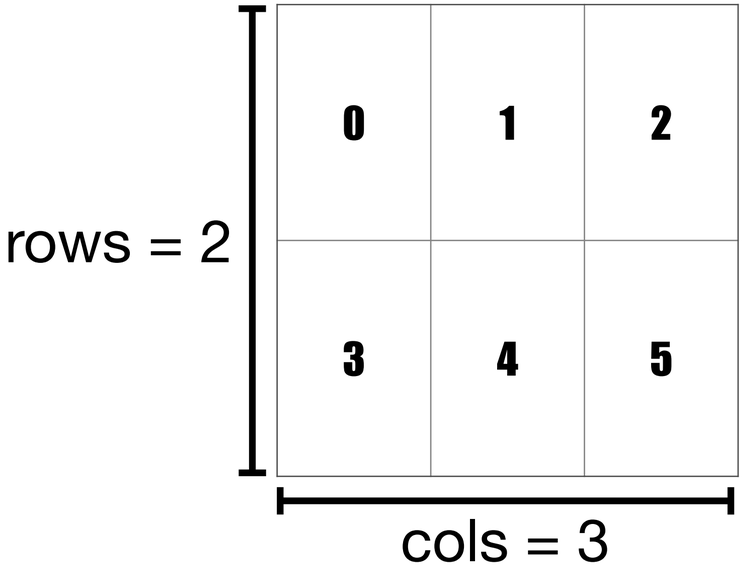
Grid, **Blocks**, and **Threads** on GPU

threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data



Once the block dimensions are decided, the **gridDim** can be calculated. These are based on the dimensions of the matrix being worked on. **Note the use of integer division here!**

gridDim.x = (cols + **blockDim.x** - 1) / **blockDim.x**
gridDim.y = (rows + **blockDim.y** - 1) / **blockDim.y**

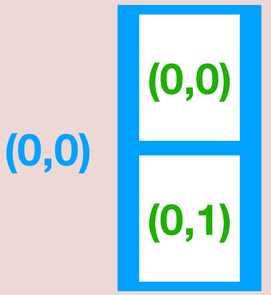
gridDim.x = (3 + **1** - 1) / **1** = **3**
gridDim.y = (2 + **2** - 1) / **2** = **1**

Grid, **Blocks**, and **Threads** on GPU

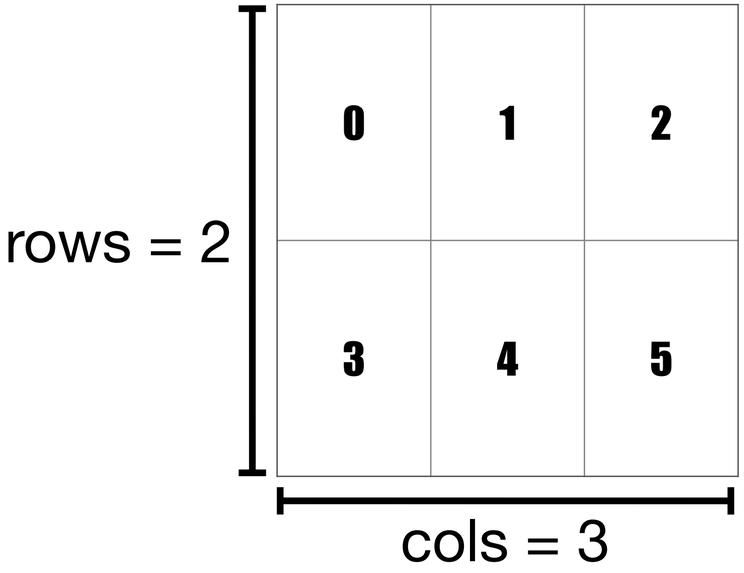
threadIdx = **(x,y)**

blockIdx = **(x,y)**

add_matrices<<<gridDim, {1,2}>>>



Matching Threads with Data



Once the block dimensions are decided, the **gridDim** can be calculated. These are based on the dimensions of the matrix being worked on. **Note the use of integer division here!**

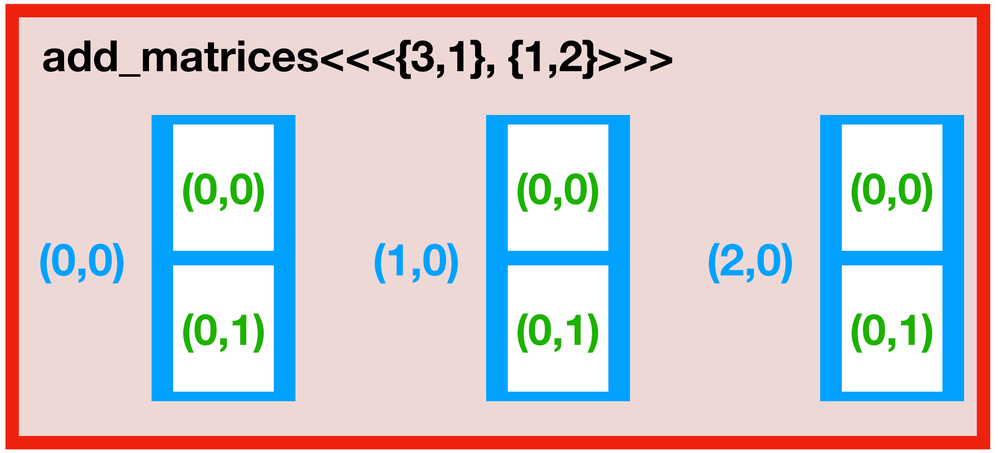
gridDim.x = (cols + **blockDim.x** - 1) / **blockDim.x**
gridDim.y = (rows + **blockDim.y** - 1) / **blockDim.y**

gridDim.x = (3 + **1** - 1) / **1** = **3**
gridDim.y = (2 + **2** - 1) / **2** = **1**

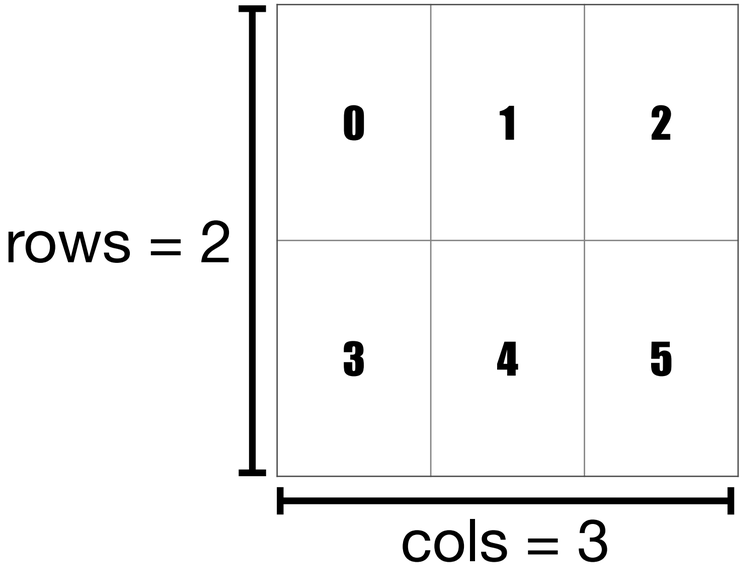
Grid, **Blocks**, and **Threads** on GPU

threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data

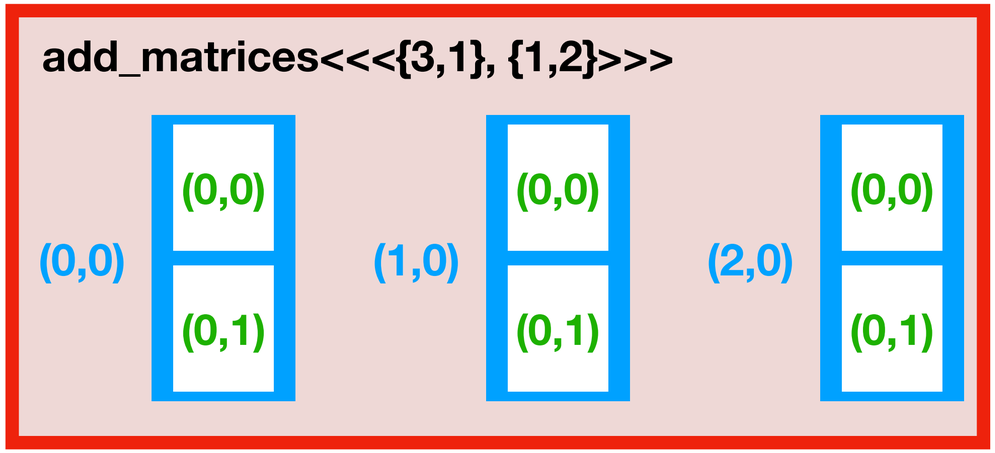


How does a particular thread map onto this matrix?

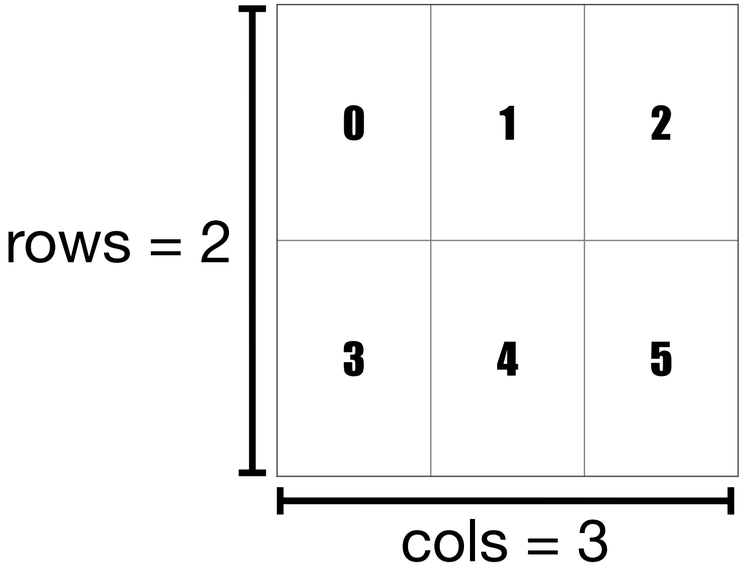
Grid, **Blocks**, and **Threads** on GPU

threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data



How does a particular thread map onto this matrix?

A thread is uniquely identified by its overall position in the grid, which is based on the size of the matrix and the thread's **blockIdx**

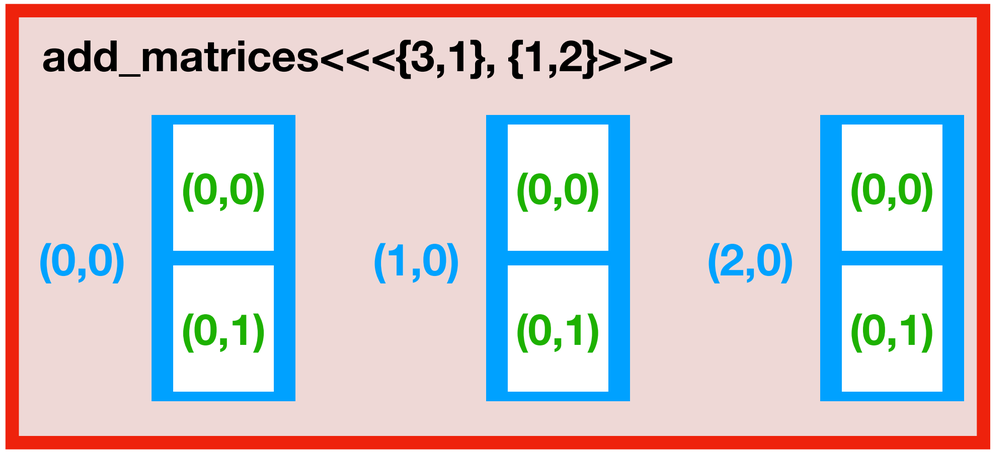
and

threadIdx

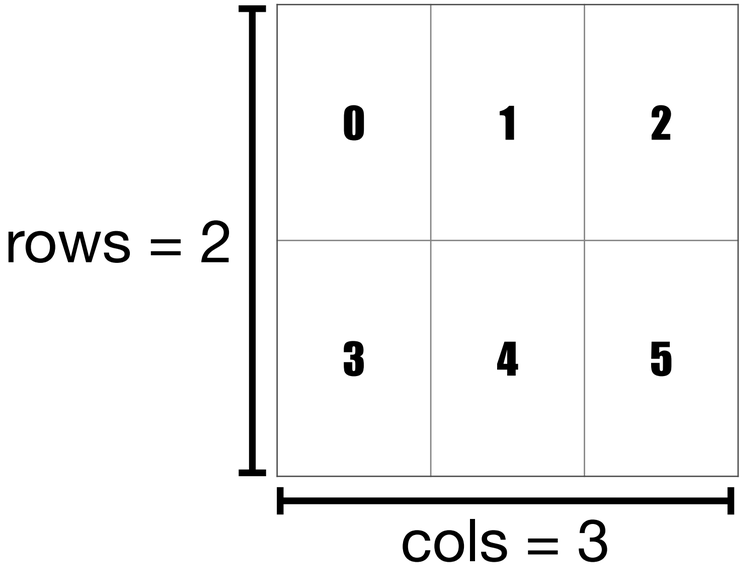
Grid, **Blocks**, and **Threads** on GPU

threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data



How does a particular thread map onto this matrix?

A thread is uniquely identified by its overall position in the grid, which is based on the size of the matrix and the thread's

blockIdx - the coordinate of a block in the grid.

We will use **blockIdx.x** and **blockIdx.y**

and

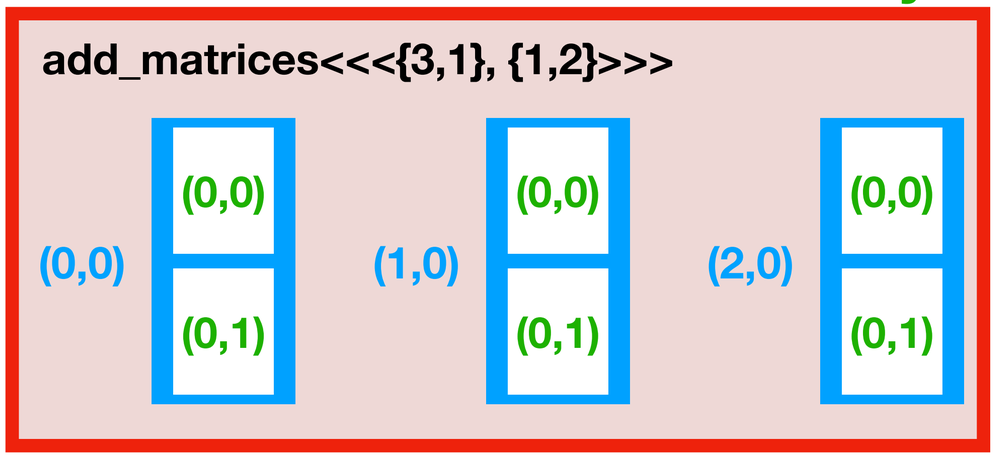
threadIdx - the coordinate of a thread in a block.

We will use **threadIdx.x** and **threadIdx.y**

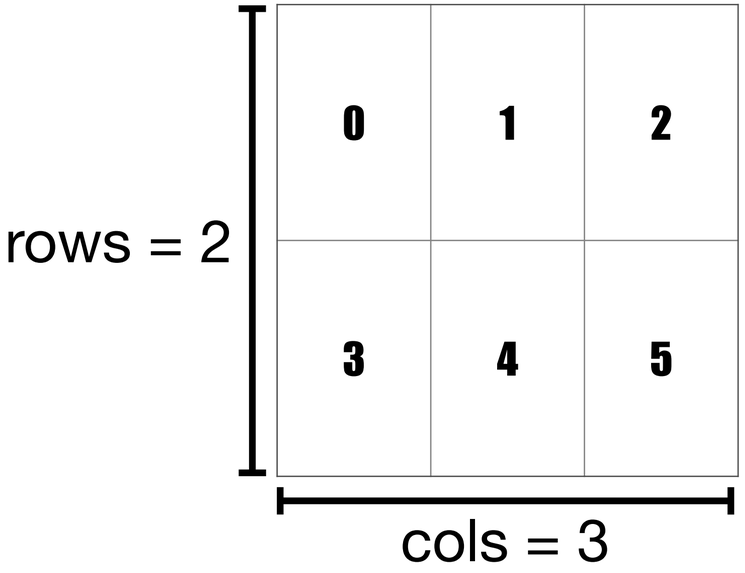
Grid, **Blocks**, and **Threads** on GPU

threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data



What entry does thread **(0,1)** in block **(1,0)** map onto?

First we find it's relative position in the x and y dimensions.

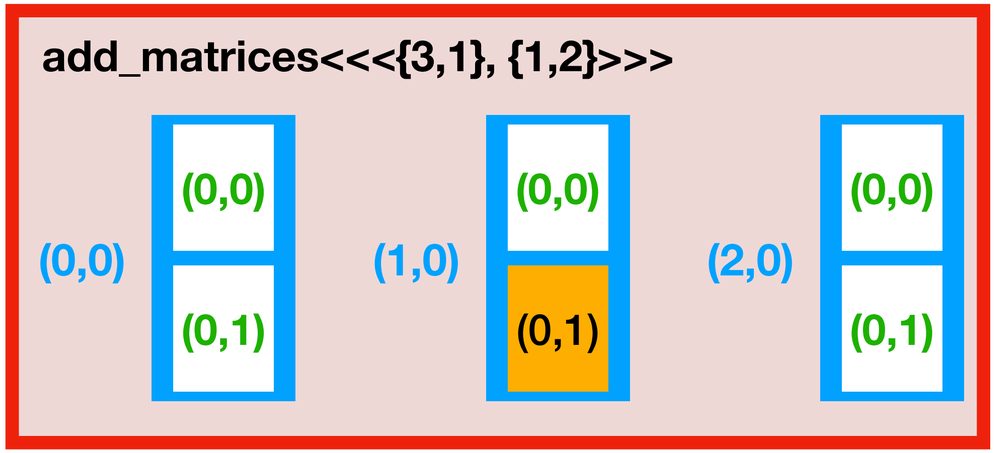
column_t = **blockIdx.x** * **blockDim.x** + **threadIdx.x**

row_t = **blockIdx.y** * **blockDim.y** + **threadIdx.y**

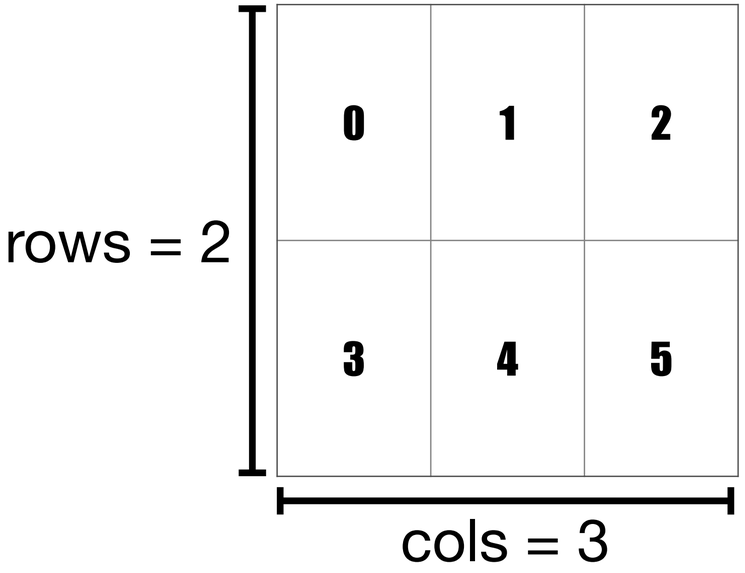
Grid, **Blocks**, and **Threads** on GPU

threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data



What entry does thread **(0,1)** in block **(1,0)** map onto?

First we find it's relative position in the x and y dimensions.

column_t = **1** * **1** + **0** = 1

row_t = **0** * **2** + **1** = 1

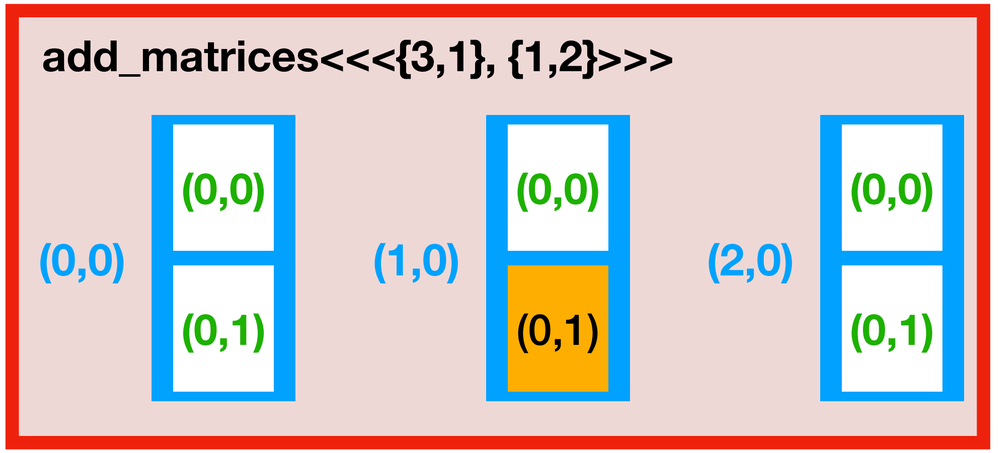
Then, we use the cols of the matrix to find the global linear address (GLA)

GLA = column_t + row_t * cols

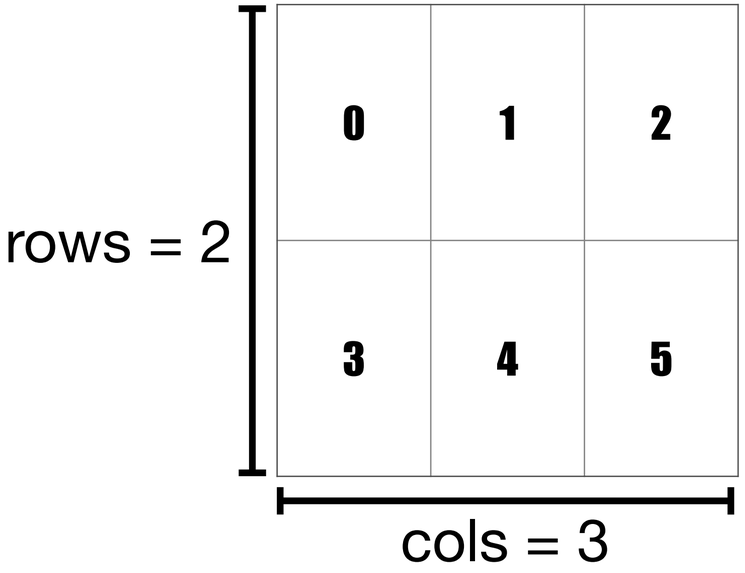
Grid, **Blocks**, and **Threads** on GPU

threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data



What entry does thread **(0,1)** in block **(1,0)** map onto?

First we find it's relative position in the x and y dimensions.

column_t = **1** * **1** + **0** = 1

row_t = **0** * **2** + **1** = 1

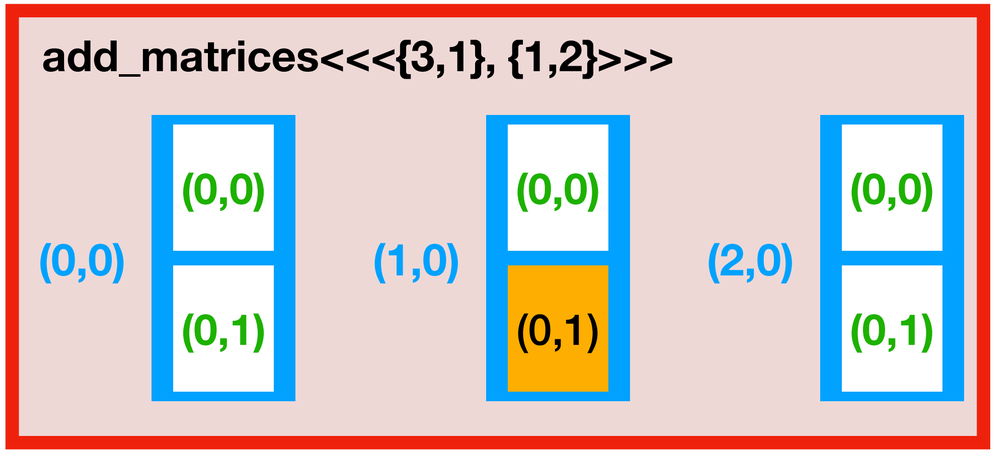
Then, we use the cols of the matrix to find the global linear address (GLA)

GLA = column_t + row_t * cols = 1 + 1 * 3 = 4

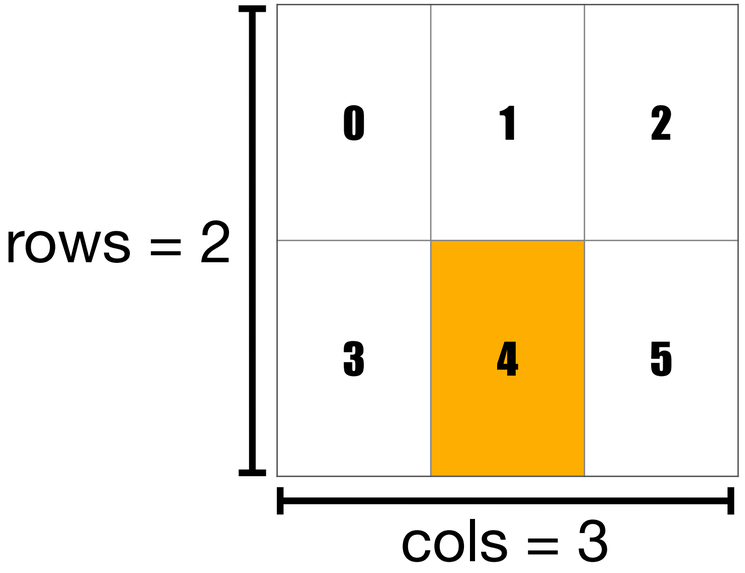
Grid, **Blocks**, and **Threads** on GPU

threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data



What entry does thread **(0,1)** in block **(1,0)** map onto?

First we find it's relative position in the x and y dimensions.

column_t = **1** * **1** + **0** = 1

row_t = **0** * **2** + **1** = 1

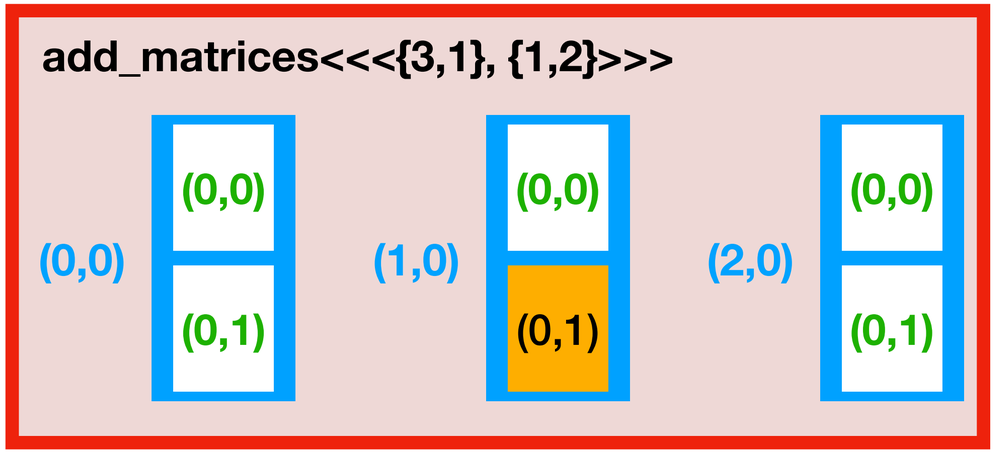
Then, we use the cols of the matrix to find the global linear address (GLA)

GLA = column_t + row_t * cols = 1 + 1 * 3 = **4**

Grid, **Blocks**, and **Threads** on GPU

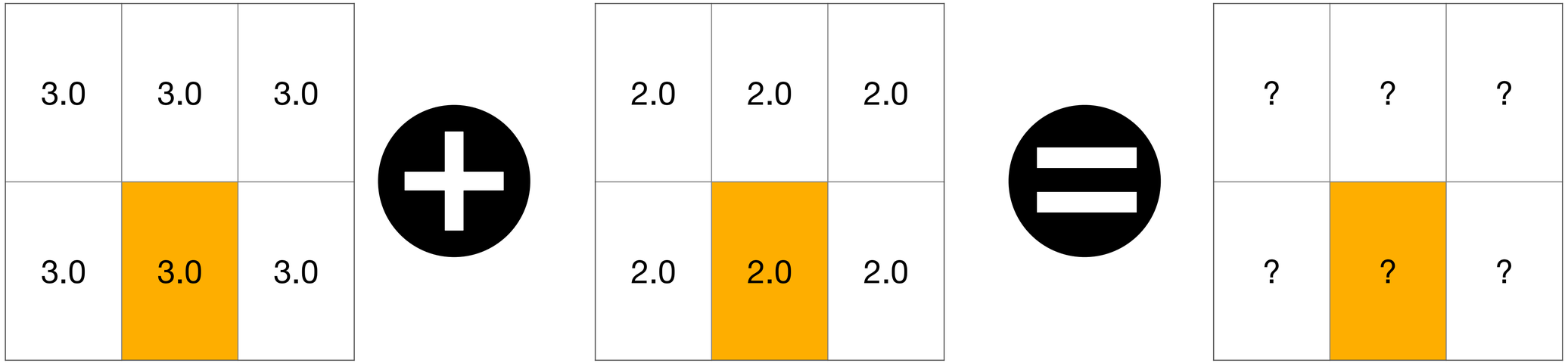
threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data

Data on the GPU

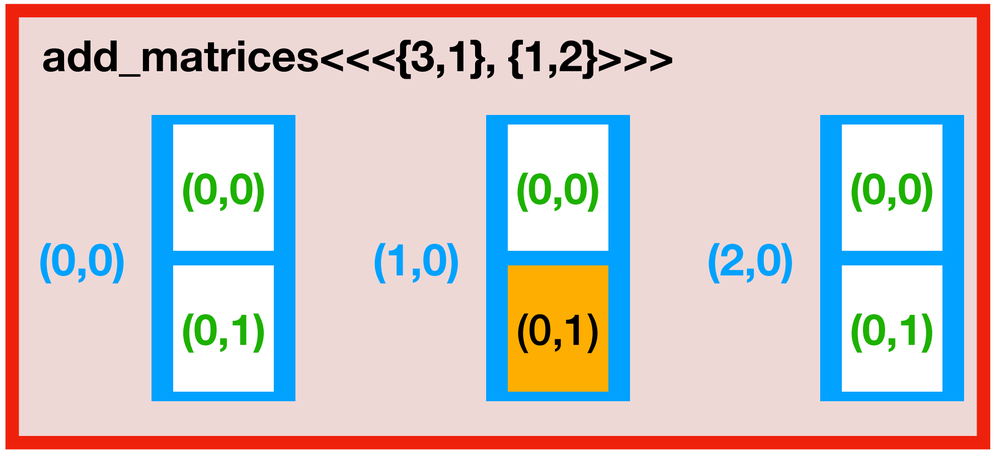


* So **thread (1,0)** in **block (0,1)** will access the 4th element in all three matrices *

Grid, **Blocks**, and **Threads** on GPU

threadIdx = **(x,y)**

blockIdx = **(x,y)**



Matching Threads with Data: Questions

What if the matrix had been 150x100 instead, and we had used **blockDim = {32,32}**?
What would the **gridDim** be?

If we used **blockDim = {32,32}** and

gridDim.x = (cols + **blockDim.x**) / **blockDim.x**

gridDim.y = (rows + **blockDim.y**) / **blockDim.y**

to calculate the grid dimension instead, what happens if both dimensions of the matrix were multiples of the **blockDim**?

Matching Threads with Data: Answers

What if the matrix had been 150x100 instead and we had used **blockDim = {32,32}**? What would the **gridDim** be?

Answer: **gridDim.x** = (100 + **32**-1) / **32** = 4
gridDim.y = (150 + **32**-1) / **32** = 5

If we used **blockDim = {32,32}** and
gridDim.x = (cols + **blockDim.x**) / **blockDim.x**
gridDim.y = (rows + **blockDim.y**) / **blockDim.y**
to calculate the grid dimension instead, what happens if both dimensions of the matrix were multiples of the **blockDim**?

Answer: The calculation simplifies to
gridDim.x = (cols/**blockDim.x)** + 1
gridDim.y = (rows/**blockDim.y)** +1
which is 1 block too many since cols and rows are evenly divided. Remember, this is integer division!

Execution Flow of CUDA Matrix Addition

1. Allocate host memory for matrices A, B, and C (h_A, h_B, cpu_C)

```
float *h_A,

// Allocate host matrices
h_A = (float*)malloc(dx*dy*sizeof(float));
```

2. Initialize the matrices h_A, h_B, and cpu_C (first touch).

```
// Init matrices
InitializeMatrixSame(h_A, dx, dy, MAT_A_VAL,"h_A");
```

3. Allocate memory on the device for the matrices (d_A, d_B, and d_C).

```
float *d_A, *d_B, *d_C;

// Allocate device matrices
cudaMalloc(&d_A, dx*dy*sizeof(float));
```

4. Copy the matrices on the host to the matrices on the device.

```
cudaMemcpy(d_A, h_A, dx*dy*sizeof(float), cudaMemcpyHostToDevice);
```

Execution Flow of CUDA Matrix Addition

5. Launch the kernel to perform the matrix addition. **Kernel launch is an asynchronous call**. Need to call a CPU blocking function.

```
// Calcuate A+B=C on the device
add_matrices<<<grid, block>>>(d_A, d_B, d_C, dx, dy);

cudaDeviceSynchronize();
```

6. Copy the result from device to host

```
cudaMemcpy(h_C, d_C, dx*dy*sizeof(float), cudaMemcpyDeviceToHost);
```

7. Free the device and host memory

```
// Cleanup
cudaFree(d_A);
```

```
// Cleanup
free(h_A);
```

8. Reset the GPU (optional - should be called only at the very end of the program)

```
cudaDeviceReset();
```

Exercise 2: Matrix Addition Using CUDA

Add commands to:

- Allocate memory on GPU using cudaMalloc function
- Copy the matrices from host to device using cudaMemcpy function
- Decide on the grid and block dimensions
- Launch the kernel
- Logic to add two vectors on GPU
- Block the CPU until GPU execution is finished

Syntax help for above exercise

https://kapeli.com/cheat_sheets/CUDA_C.docset/Contents/Resources/Documents/index

To build use ./build.sh

To submit use sbatch submit.sh

Adjust the size of the matrices within by editing the last line of submit.sh

Exercise 2: CUDA Matrix Addition Check

1. What compiler did we use to build the CUDA-C code?

2. Did the GPU execution validate for non-square matrices?

3. What was the CPU execution time for adding two 1024x1024 matrices? For size 16384x16384?

4. What was the GPU execution time for adding two 1024x1024 matrices? For size 16384x16384?

5. Based on the previous results, how long will it take to add two 65536 x 65536 matrices? Using the nvprof output, what percentage of the time was spent on computation vs. data transfer?

Debug Problem 1: CUDA Build Error

- Scenario: Help!!!! Our team recently switched to using nvhpc/20.11, but now our CUDA code won't build. It was working fine with pgi/20.4! We've included our code, build script, and Makefile in the '**Lesson_1_matrixAdd/CUDA/debug1_buildErr**' directory. Here's a screenshot of the g*sh-d4rn error:

```
Currently Loaded Modules:
  1) ncarenv/1.2   2) nvhpc/20.11   3) cuda/11.0.3



rm *.o matrix_add.exe
rm: cannot remove '*.o': No such file or directory
rm: cannot remove 'matrix_add.exe': No such file or directory
make: *** [clean] Error 1
nvc++ -c -g -O3 -std=c++11 -Wall main.cpp
nvc++ -c -g -O3 -std=c++11 -Wall common.cpp
nvc++ -c -g -O3 -std=c++11 -Wall functions.cpp
nvc++ -c -g -std=c++11 -O3 -I/glade/u/apps/dav/opt/cuda/11.0.3//include matrix_add.cu
nvc++ -o matrix_add.exe main.o common.o functions.o matrix_add.o   -L/glade/u/apps/dav/opt/cuda/11.0.3//lib64 -lcudart
/usr/bin/ld: cannot find matrix_add.o: No such file or directory
make: *** [matrix_add.exe] Error 2
```

Debug Problem 2: Kernel Launch

- Scenario: Oops, we've got another problem with our CUDA Matrix Addition code. We used the command:

/glade/u/apps/dav/opt/cuda/11.0.3/extras/demo_suite/deviceQuery

to find the maximum dimensions we could use for our matrix addition kernel on a V100. A snippet of the output from the command is below:

```
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

Why is the code in the **'Lesson_1_matrixAdd/CUDA/debug2_KernLaunch'** folder giving the error below when we're using a **thread block size of only (256,256,1)?**

Fatal error: Kernel execution failure or cudaMemcpy H2D failure (invalid configuration argument at matrix_add.cu:59)
***** FAILED - ABORTING**

Debug Problem 3: OpenACC Illegal Address

- Scenario: Our OpenACC Matrix Addition code that worked for square matrices is behaving strangely for non-square matrices. For example, it fails during runtime for 8192x4096 size matrices with this error about an "Illegal address":

```
Failing in Thread:1
call to cuStreamSynchronize returned error 700: Illegal address during kernel execution
```

Our teammate says it's because we need more memory on the GPU, since the default 1024x1024 case ran fine. (Can you PLEASE tell them why they're wrong?!?) For some matrix dimensions it runs, but fails the verification test. We've included our output and error files for 5 test cases in the **'Lesson_1_matrixAdd/OpenACC/debug3_IllegalAdd'** folder. We thought OpenACC handled the GPU memory allocation, so we're not sure why we're getting this error?

Extra Coding Practice 1: OpenACC-F Matrix Addition

- A FORTRAN version of the Matrix Addition code is located in the '**Lesson_1_matrixAdd/OpenACC_FORTRAN/exercise'** folder
- Using the same set of OpenACC directives we used for the C version, please offload the calculation of 'c_gpu' to the GPU.
- Hint: refer to OpenACC API Quick Reference Guide (https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf) to see the corresponding FORTRAN syntax.

COFFEE/SUNSHINE BREAK....

See you at 3:15!

Naive Matrix Multiplication with OpenACC

Naive Matrix Multiplication with OpenACC

As we saw earlier when working on the Matrix Addition code, using OpenACC to parallelize code is much easier than CUDA.

Parallelizing code with openACC involves adding directive to your sequential code. First of all, lets take a look at our cpu matrix multiplication code that runs sequentially.

```
// Calculate AxB=C on the host
float temp = 0.0;
for (int i = 0; i < m; i++)
    for (int j = 0; j < q; j++)
    {
          temp = 0.0;
          for (int k = 0; k < p; k++)
          {
              temp += A[i*p+k] * B[k*q+j];
          }
          C[i*q+j] = temp;
    }
}
```

We will be adding OpenACC directives to this base version of our matrix multiplication code.

Naive Matrix Multiplication with OpenACC

Navigate to the Lesson 2 OpenACC exercise folder

Exercise 3: Naive Matrix Multiplication Using OpenACC

In the "matrix_mult.cc" file, we will be adding OpenACC directives to the "openacc_matrix_mult" function.

- Add a directive that copies in/out the required Matrix variables.

- Add a directive that collapses the two tightly-nested for loops. Note: Add the reduction clause.

- Add a directive that vectorizes the inner most loop. Note: We will also be adding the reduction clause to this directive.

Exercise 3 : OpenACC C/C++ Matrix Multiplication Check

1. Did the GPU execution validate for non-square matrices?

2. For approximately what size matrices did GPU execution outperform single-threaded CPU execution?
   (Assuming you built without debug flags and ran without any profiling, ex: nvprof).

Questions?

Naive Matrix Multiplication with CUDA

Naive Matrix Multiplication with CUDA

1. Allocate memory on the host for matrices h_A, h_B, cpu_C and gpu_C.

```
// Allocate host matrices
h_A = (float*)malloc(rowsA*colsA*sizeof(float));
```

2. Initialize the Matrices using the "InitializeMatrixSame" function. Note: This function is defined in the common.cpp file

```
// Init matrices
InitializeMatrixSame(h_A, rowsA, colsA, MAT_A_VAL,"h_A");
```

3. Carry out Matrix Multiplication on the CPU.

4. Allocate memory on the device for the matrices that will be used in the devices' matrix multiplication.

```
//Allocate device memory
cudaMalloc(&d_A, m*n*sizeof(float));
```

Naive Matrix Multiplication with CUDA

5. Copy the values from the matrix variables on the host to the matrix variables on the device using cudaMemcpy().
   Note: We are only copying matrices h_A and h_B.

```
cudaMemcpy(d_A, h_A, m*n*sizeof(float), cudaMemcpyHostToDevice);
```

6. Launch the Cuda Kernel that will carry out the GPU matrix multiplication on the device.

```
mmul<<<grid,block>>>(d_A,d_B,d_C,m,n,q);
```

Inside the Kernel:

- Calculate the row and column indexes based on the block Id, block dimensions and the thread Id.

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

- Check to make sure the calculated row and column variables are not outside the assigned matrix dimensions.

```
if( col < q && row < m)
```

- Carry out Matrix multiplication and store the result in the result matrix.

```
for(int i = 0; i < n; i++)
{
    sum += a[row*n+i] * b[i*q+col];
}
c[row*q+col] = sum;
```

Naive Matrix Multiplication with CUDA

7. Pause the host until the device is done carrying out the matrix multiplication. This is done by calling the "cudaDeviceSynchronize()" function immediately after the Kernel launch.

8. Copy the result from the devices' result variable to the "gpu_C" matrix variable.

```
// Transefr results from device to host
cudaMemcpy(gpu_C, d_C, sizeof(float)*m*q, cudaMemcpyDeviceToHost);
```

9. Deallocate the memory for d_A, d_B and d_C on the device using cudaFree().

```
cudaFree(d_A);
```

11. Carry out result verification.

```
// Check for correctness
MatrixVerification(gpu_C, cpu_C, m, q, VERIF_TOL);
```

12. Deallocate the memory for h_A, h_B, cpu_C and gpu_C on the host using free().

```
free(h_A);
```

Naive Matrix Multiplication with CUDA

Navigate to the Lesson 2 CUDA exercise folder

Exercise 4: Naive Matrix Multiplication using CUDA

Inside the "MatrixMult.cu" file, we will be filling out chunks of code to complete the "gpuMult" and "mmul" functions:

1) For the "mmul" function:
   - Write code to calculate the row and column variables using thread ID, block ID and block dimensions.
   - Write code that Multiplies Matrices A and B and store result in matrix C.

2) For the "gpuMult" function:
   - Write code that calculates the grid and block dimensions that will be used to launch the Cuda Kernel.
   - Write code to Launch the Cuda Kernel.
   - Write code to copy the values stored in the d_C matrix variable to the gpu_C variable.
   - Write code to deallocate the memory used by the d_A, d_B and d_C variables.

Introduction to nvprof (NVIDIA Profiler)

nvprof is a command line profiler that provides information about your program execution along with timing information.

Below is an example of the nvprof output from our Matrix Multiplication code. This output is generated by adding the nvprof command while running our executable. Example: nvprof ./matMult.exe

```
pgc++ -c -g -O3 -std=c++11 -Wall   main.cpp
pgc++ -c -g -O3 -std=c++11 -Wall   functions.cpp
pgc++ -c -g -O3 -std=c++11 -Wall   common.cpp
nvcc -c -g -std=c++11 -O3 -I/glade/u/apps/dav/opt/cuda/11.0.3//include matrixMul.cu
pgc++ -o output.exe main.o functions.o common.o matrixMul.o  -L/glade/u/apps/dav/opt/cuda/11.0.3//lib64 -lcudart
==149693== NVPROF is profiling process 149693, command: ./output.exe

Entire CPU process took 3.360000 seconds.... Starting Cuda Process.
Done. Matrix multiplication on GPU took 0.360000 seconds.
==149693== Profiling application: ./output.exe
==149693== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   43.82%  1.4974ms         2  748.68us  745.47us  751.90us  [CUDA memcpy HtoD]
                   38.48%  1.3149ms         1  1.3149ms  1.3149ms  1.3149ms  mmul(float*, float*, float*, int, int, int)
                   17.71%  605.18us         1  605.18us  605.18us  605.18us  [CUDA memcpy DtoH]
      API calls:   97.41%  249.69ms         3  83.231ms  271.31us  249.02ms  cudaMalloc
                    1.15%  2.9541ms         3  984.69us  975.76us  1.0010ms  cudaMemcpy
                    0.53%  1.3542ms         1  1.3542ms  1.3542ms  1.3542ms  cudaDeviceSynchronize
                    0.29%  752.64us       101  7.4510us     130ns  333.63us  cuDeviceGetAttribute
                    0.29%  745.28us         3  248.43us  224.24us  289.07us  cudaFree
                    0.27%  696.19us         1  696.19us  696.19us  696.19us  cuDeviceTotalMem
                    0.04%  90.345us         1  90.345us  90.345us  90.345us  cuDeviceGetName
                    0.02%  44.011us         1  44.011us  44.011us  44.011us  cudaLaunchKernel
                    0.00%  7.4660us         1  7.4660us  7.4660us  7.4660us  cuDeviceGetPCIBusId
                    0.00%  3.3140us         4     828ns     190ns  2.3040us  cudaGetLastError
                    0.00%  2.0040us         3     668ns     223ns  1.1470us  cuDeviceGetCount
                    0.00%  1.5240us         2     762ns     511ns  1.0130us  cuDeviceGet
                    0.00%     234ns         1     234ns     234ns     234ns  cuDeviceGetUuid
```

Introduction to nvprof (NVIDIA Profiler)

A more detailed output can be generated by using: nvprof --print-gpu-trace ./output.exe

This command is most useful when running your code on two GPUs. It will display what GPU each part/Kernel of your code is running on. Below is an example of the output from running the command above.

```
rm *.o output.exe
pgc++ -c -g -O3 -std=c++11 -Wall   main.cpp
pgc++ -c -g -O3 -std=c++11 -Wall   functions.cpp
pgc++ -c -g -O3 -std=c++11 -Wall   common.cpp
nvcc -c -g -std=c++11 -O3 -I/glade/u/apps/dav/opt/cuda/11.0.3//include matrixMul.cu
pgc++ -o output.exe main.o functions.o common.o matrixMul.o  -L/glade/u/apps/dav/opt/cuda/11.0.3//lib64 -lcudart
==42032== NVPROF is profiling process 42032, command: ./output.exe

Entire CPU process took 3.260000 seconds.... Starting Cuda Process.
Done. Matrix multiplication on GPU took 0.270000 seconds.
==42032== Profiling application: ./output.exe
==42032== Profiling result:
   Start  Duration           Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput  SrcMemType  DstMemType          Device   Context    Stream
  Name
377.59ms  798.97us                    -               -         -         -         -  4.0000MB  4.8891GB/s    Pageable      Device  Tesla V100-SXM2         1         7
  [CUDA memcpy HtoD]
378.62ms  784.35us                    -               -         -         -         -  4.0000MB  4.9802GB/s    Pageable      Device  Tesla V100-SXM2         1         7
  [CUDA memcpy HtoD]
379.42ms  1.4475ms            (32 32 1)       (32 32 1)        32        0B        0B         -           -           -           -  Tesla V100-SXM2         1         7
  mmul(float*, float*, float*, int, int, int) [118]
380.88ms  625.09us                    -               -         -         -         -  4.0000MB  6.2491GB/s      Device    Pageable  Tesla V100-SXM2         1         7
  [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler sh
ows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
```

Exercise 4 : CUDA Matrix Multiplication Check

1. Did the GPU execution validate for non-square matrices?

2. Using the nvprof output, what percentage of the time for the GPU execution was spent on computation vs. data transfer? How does this ratio change as the matrix sizes increase?

Questions?

Debug Problem 4: OpenACC GPU Check

- Scenario: There was a potential runtime error in all of the OpenACC exercises we looked at today! Our function that validates the CPU results against the ostensible "GPU" results doesn't check whether the GPU was actually used. Can you help our team verify that we're actually running on a GPU?
  - Hint: When not using nvprof, here are two environment variables you can set in the submit script that will provide information about the OpenACC execution:
    - **export NV_ACC_TIME=1**
      - prints kernel timing data
    - **export NV_ACC_NOTIFY=** [1,2,4,8,16] (bitmask)
      - 1 - launch
      - 2 - data upload/download
      - 4 - wait (explicit or implicit) for device
      - 8 - data/compute region
      - 16 - data createallocatedeletefree

Extra Coding Practice 2: OpenACC-F Matrix Multiplication

- A FORTRAN version of the Matrix Multiplication code is located in the '**Lesson_2_matrixMult/OpenACC_FORTRAN/exercise'** folder
- Using the same set of OpenACC directives we used for the C version, please offload the calculation of 'c_gpu' to the GPU.
- Hint: refer to OpenACC API Quick Reference Guide (https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf) to see the corresponding FORTRAN syntax.

Extra Coding Practice 3: Matrix Mult Scaling Curves

- Scenario: We'd like you to gather performance data to compare how the GPU and CPU Matrix Multiplication code performs as the size of the matrices increases. We know it's not fair to compare the times for a single GPU to a single-threaded CPU, so **we'd like try using OpenACC's Multicore CPU capabilities** to run the code on a fully-subscribed CPU node. (Documented here: https://docs.nvidia.com/hpc-sdk/compilers/openacc-gs/index.html#using-openacc)
- Then, for increasing sizes of square matrix multiplication, you can fill out this example table with performance data. Feel free to optimize the GPU and CPU code - the fastest times (with correct results) win!

Square Mat Dimension	Single GPU Execution Time	Single CPU Node Execution Time
256		
512		
1024		
2048		
4096		
8192		

References

1. https://www.olcf.ornl.gov/wp-content/uploads/2020/04/OpenACC-Course-2020-Module-1.pdf
2. https://www.olcf.ornl.gov/wp-content/uploads/2020/02/OpenACC_Course_2020_Module_2.pdf
3. https://www.olcf.ornl.gov/wp-content/uploads/2020/06/OpenACC_Course_2020_Module_3_updated.pdf
4. https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf
5. https://docs.nvidia.com/hpc-sdk/compilers/openacc-gs/
6. https://docs.computecanada.ca/wiki/OpenACC_Tutorial_-_Optimizing_loops