GPU Programming Workshop

Session 2: Introduction to CUDA

Special Technical Projects
&
Client Support Group

UCAR

April 13, 2021

Overview

- Connect to Rescale!
- Review of key GPU Architecture concepts and terminologies
- Intro to CUDA
- Breakout Room 1
  - Coding Practice - Naive Matrix Multiplication with CUDA
- Shared Memory
- Breakout Room 2
  - Coding Practice - Shared Memory Matrix Multiplication with CUDA

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

Task 1: Accessing the Workshop Instance

- Download the ssh key given to you by email
- Change file permissions of the private key

```
chmod 600 <path_to_private_key>
```

- Use ssh from a terminal window to connect to the instance IP address given to you
  - -i : Specifies identity file/private key
  - -p : Specifies the port number to use to connect

```
ssh -i <path_to_private_key> -p 22 <instance IP>
```

Find your instance IP and username here or ask your mentor later:
https://docs.google.com/spreadsheets/d/1m0W_u4XNAT2j6XjQr4hwElbft2OX_5xwE9zEuR91-3Q/edit?usp=sharing

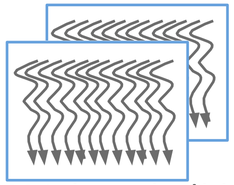
Software Abstraction

Software term

Threads

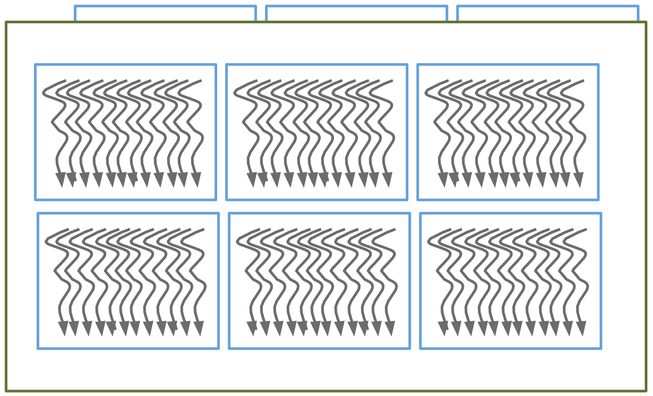
Vector (OpenACC)

Blocks



512–1024 threads / block

Grids



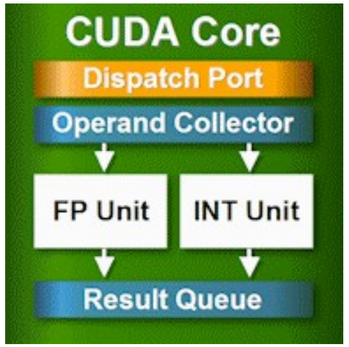
gangs (OpenACC)

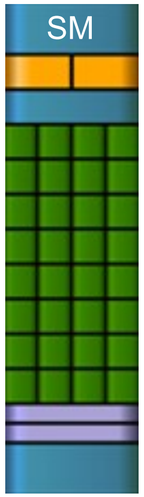
Kernel is executed by threads processed by CUDA Core

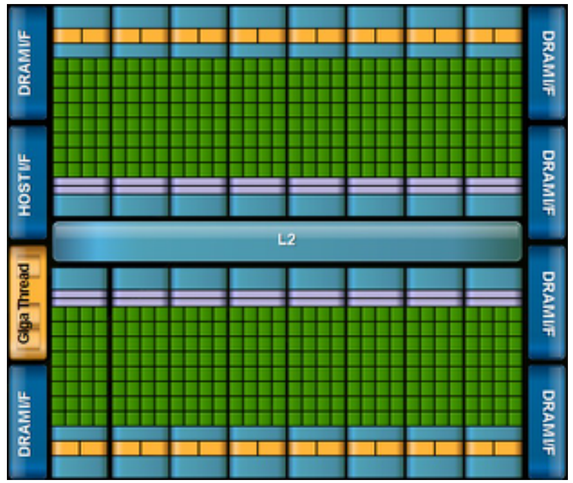
Maximum 8 blocks per SM
32 parallel threads are executed at the same time in a *WARP*

One grid per kernel with multiple concurrent kernels

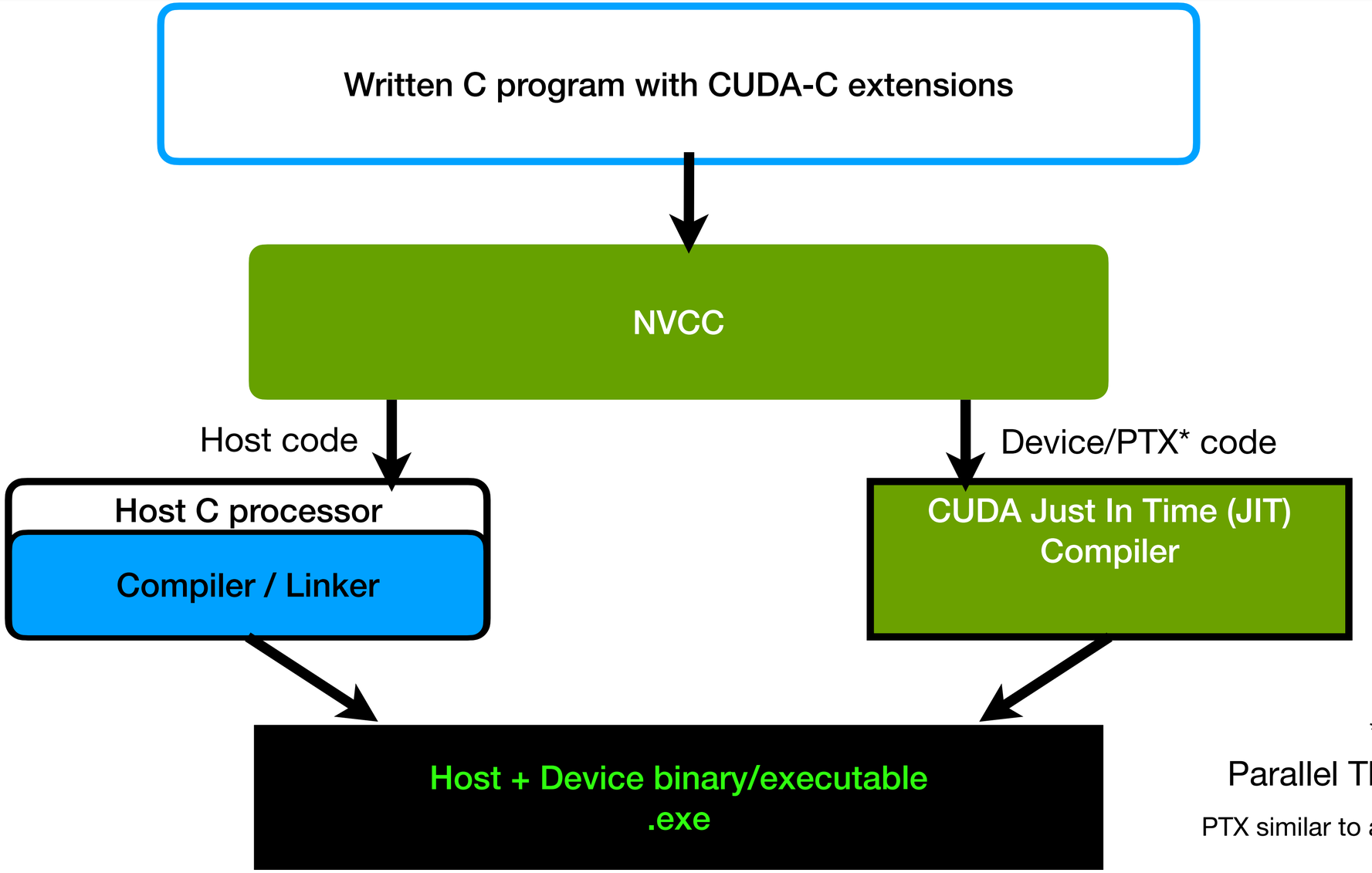
Hardware term







CUDA Compilation



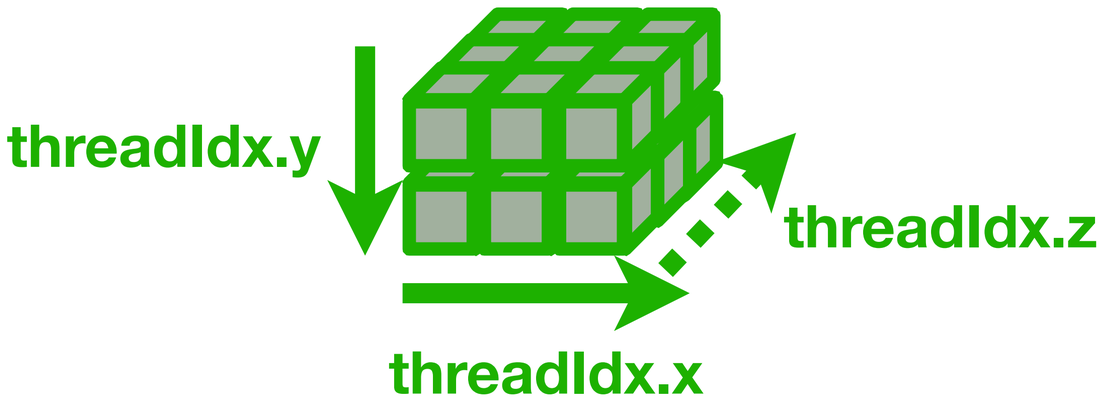
*PTX =
Parallel Thread Execution

PTX similar to assembly code on CPU

Understanding CUDA Blocks and Grids

Threads, Blocks, and Grids

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**.



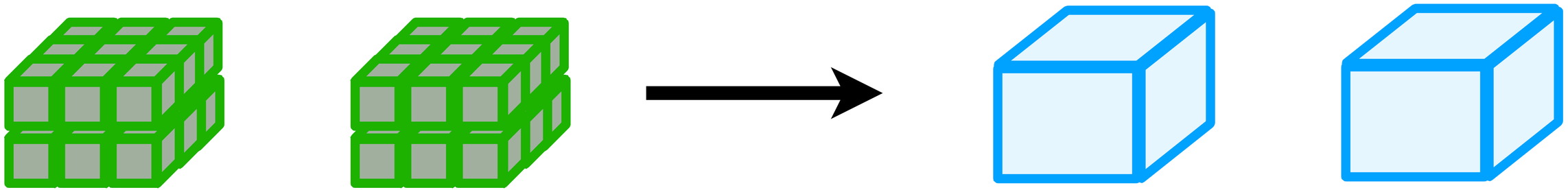
Breakout Session 1

- **Task 1: Coding Practice - CUDA Matrix Multiplication**

Threads, Blocks, and Grids

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

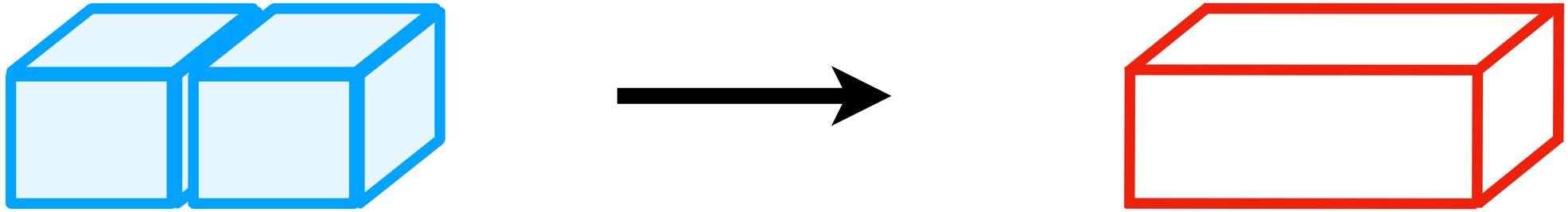
Becomes 2 blocks each with **blockDim** = **{x,y,z}** = {3,2,3}

Threads, Blocks, and Grids

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}

Threads, Blocks, and Grids

threadIdx.x threadIdx.y threadIdx.z	blockDim.x blockDim.y blockDim.z	blockIdx.x blockIdx.y blockIdx.z	gridDim.x gridDim.y gridDim.z
Defines what position a thread is in within a block	Defines how many threads there are within a block	Defines what position a block is within a grid	Defines how many blocks there are per grid (or CUDA call)

blockDims are typically chosen by the programmer - keeping in mind that a block can only have a total of 1024 threads per block, the warp size, and the dimensions of the matrix being operated on.

gridDims are calculated using the array dimensions and **blockDims**.

e.g. **gridDim.x** = (**arrayDim.x** + **blockDim.x** - 1) / **blockDim.x**

threadIdx and **blockIdx** are assigned to a thread/block and are used with **blockDim** and array dimensions to calculate the **global linear address** of the thread.

Threads, Blocks, and Grids - An Aside

threadIdx.x threadIdx.y threadIdx.z	blockDim.x blockDim.y blockDim.z	blockIdx.x blockIdx.y blockIdx.z	gridDim.x gridDim.y gridDim.z
Defines what position a thread is in within a block	Defines how many threads there are within a block	Defines what position a block is within a grid	Defines how many blocks there are per grid (or CUDA call)

The number of dimensions used is a programming decision, but should match the type of problem being worked on. If not assigned while programming, a dimension is assumed to be 1.

Examples:

- A vector: only x dimension

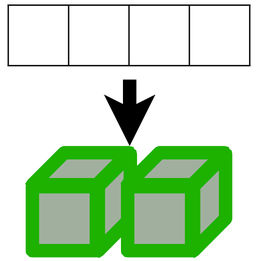
Threads, Blocks, and Grids - An Aside

threadIdx.x	blockDim.x	blockIdx.x	gridDim.x
threadIdx.y = 0	blockDim.y = 1	blockIdx.y = 0	gridDim.y = 1
threadIdx.z = 0	blockDim.z = 1	blockIdx.z = 0	gridDim.z = 1
Defines what position a thread is in within a block	Defines how many threads there are within a block	Defines what position a block is within a grid	Defines how many blocks there are per grid (or CUDA call)

The number of dimensions used is a programming decision, but should match the type of problem being worked on. If not assigned while programming, a dimension is assumed to be 1 and an index to be 0.

Examples:

- A vector: only x dimension



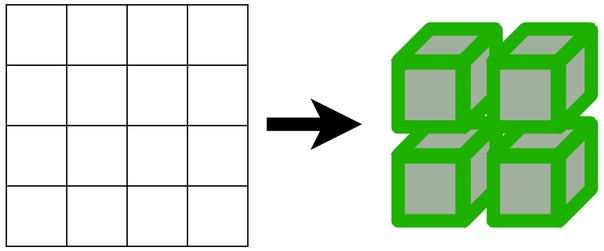
Threads, Blocks, and Grids - An Aside

threadIdx.x threadIdx.y threadIdx.z = 0	blockDim.x blockDim.y blockDim.z = 1	blockIdx.x blockIdx.y blockIdx.z = 0	gridDim.x gridDim.y gridDim.z = 1
Defines what position a thread is in within a block	Defines how many threads there are within a block	Defines what position a block is within a grid	Defines how many blocks there are per grid (or CUDA call)

The number of dimensions used is a programming decision, but should match the type of problem being worked on. If not assigned while programming, a dimension is assumed to be 1 and an index to be 0.

Examples:

- A vector: only x dimension
- A matrix: x and y dimensions



Uniquely Identifying a Thread

A **global linear address** (GLA) is used to uniquely identify a thread within a grid and used to match up array element(s) and a thread.

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 three values are sometimes enough on their own, but can be combined with the array dimensions to calculate the GLA of a thread:

GLA = column_t + row_t * **arrayDim.x** + aisle_t * **arrayDim.x** * **arrayDim.y**

Uniquely Identifying a Thread - Simplifying

A **global linear address** (GLA) is used to uniquely identify a thread within a grid and used to match up array element(s) and a thread.

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**.

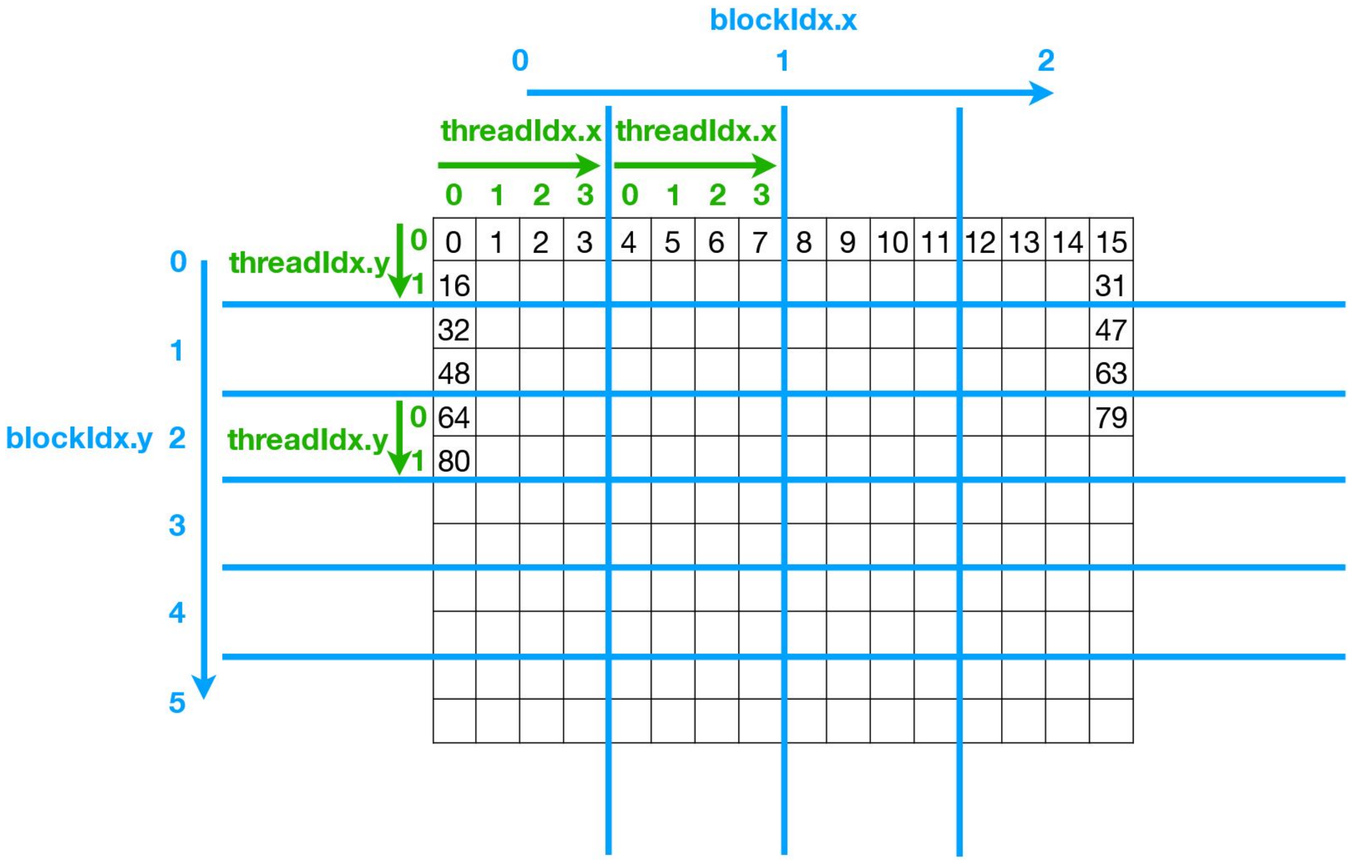
column_t = **blockIdx.x** * **blockDim.x** + **threadIdx.x**
row_t = **blockIdx.y** * **blockDim.y** + **threadIdx.y**
aisle_t = ~~**blockIdx.z** * **blockDim.z** + **threadIdx.z**~~
0 * 1 + 0 = 0

These three values are sometimes enough on their own, but can be combined with the array dimensions to calculate the GLA of a thread:

GLA = column_t + row_t * **arrayDim.x** + ~~aisle_t * **arrayDim.y** * **arrayDim.x**~~

This becomes much simpler when fewer dimensions are used (e.g. a matrix).

Uniquely Identifying a Thread - An Example



If we are working on a **16x16** matrix and the threads per block is chosen as:

blockDim {x, y} = {4, 2} giving **gridDim** {x, y} = {4, 6}

what thread is uniquely identified by
threadIdx {x, y} = {3, 1} **blockIdx** {x, y} = {1, 2} ?

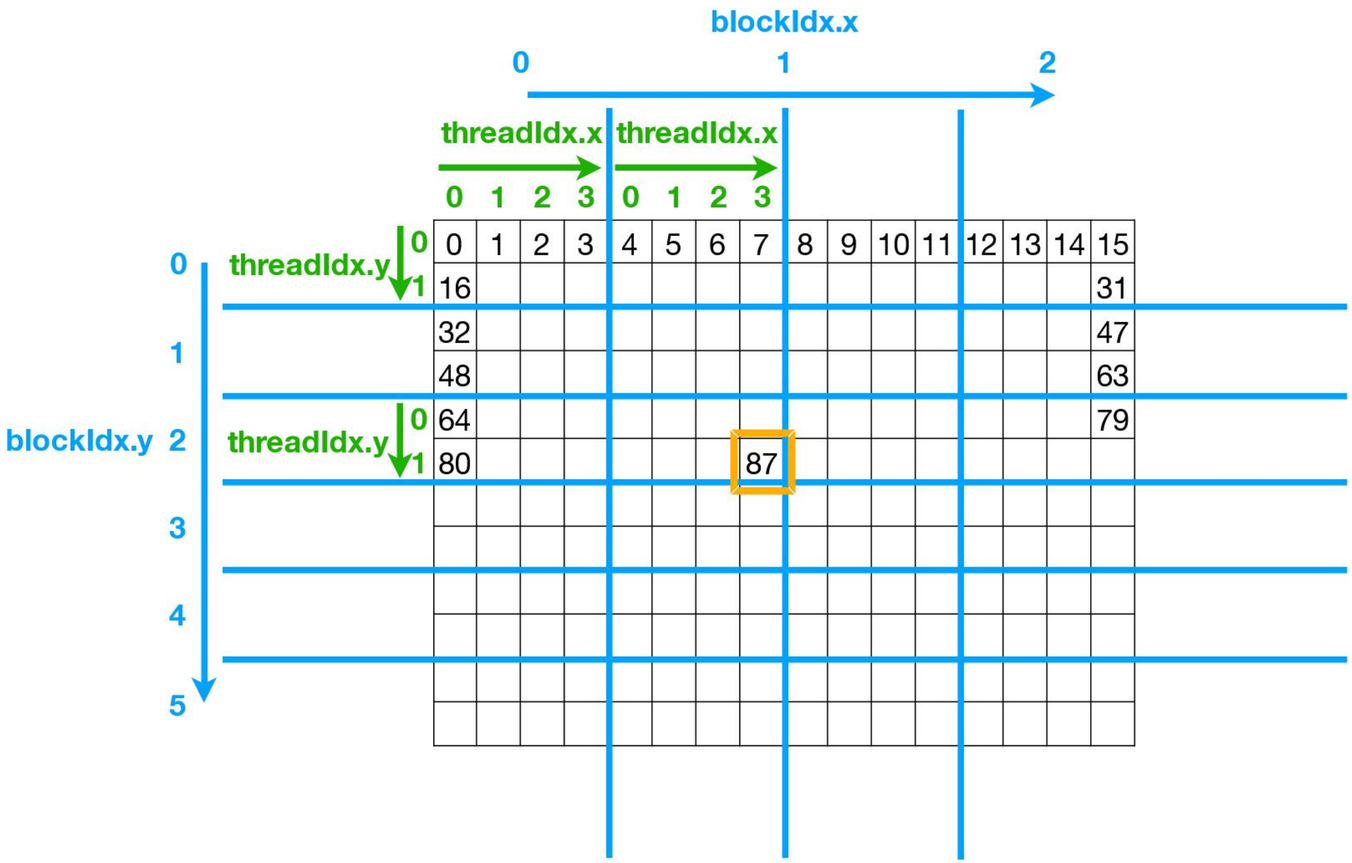
Calculation

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

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

GLA = column_t + row_t * **arrayDim.x**

Uniquely Identifying a Thread - An Example



If we are working on a **16x16** matrix and the threads per block is chosen as:

blockDim {x, y} = {4, 2} giving **gridDim** {x, y} = {4, 6}

what thread is uniquely identified by
threadIdx {x, y} = {3, 1} **blockIdx** {x, y} = {1, 2} ?

Calculation

column_t = (**blockIdx.x** * **blockDim.x**) + **threadIdx.x**
= (1 * 4) + 3 = 7

row_t = (**blockIdx.y** * **blockDim.y**) + **threadIdx.y**
= (2 * 2) + 1 = 5

GLA = column_t + row_t * **arrayDim.x**
= 7 + (5 * 16) = 87

Naive Matrix Multiplication with CUDA

Naive Matrix Multiplication with CUDA - The Host Function

To utilize the GPU, the CPU first calls the `__host__` function and - after some setup in the `__host__` function - calculation will take place when the `__global__` function is called and the CUDA kernel is launched.

In the `__host__` function:

1. Allocate memory on the device for the matrices that will be used in the matrix multiplication.

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

2. Copy the values from the matrix variables on the host to the matrix variables on the device

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

3. Choose the block dimensions and compute the grid dimensions (m is the # of rows, q is # of columns).

```
//calculate grid and block dimensions
unsigned int grid_rows = (m + block_size - 1) / block_size;
unsigned int grid_cols = (q + block_size - 1) / block_size;
dim3 grid(grid_cols, grid_rows);
dim3 block(block_size, block_size);
```

4. Launch the `__global__` function that will carry out the GPU matrix multiplication on the device. Note the presence of grid and block dimensions in the kernel call.

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

5. Pause the host until the device is done carrying out the matrix multiplication.

```
cudaDeviceSynchronize();
```

Naive Matrix Multiplication with CUDA - The CUDA Kernel

In the `__global__` function:

6. Calculate the row and column indexes based on the **blockIdx**, **blockDim** and the **threadIdx**

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

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

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

8. 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 - The Host Function

Returning to the `__host__` function:

9. After the CUDA Kernel returns, copy 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);
```

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

```
cudaFree(d_A);
```

Go to
Breakout Session 1

Return to main room:
~30 mins

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.

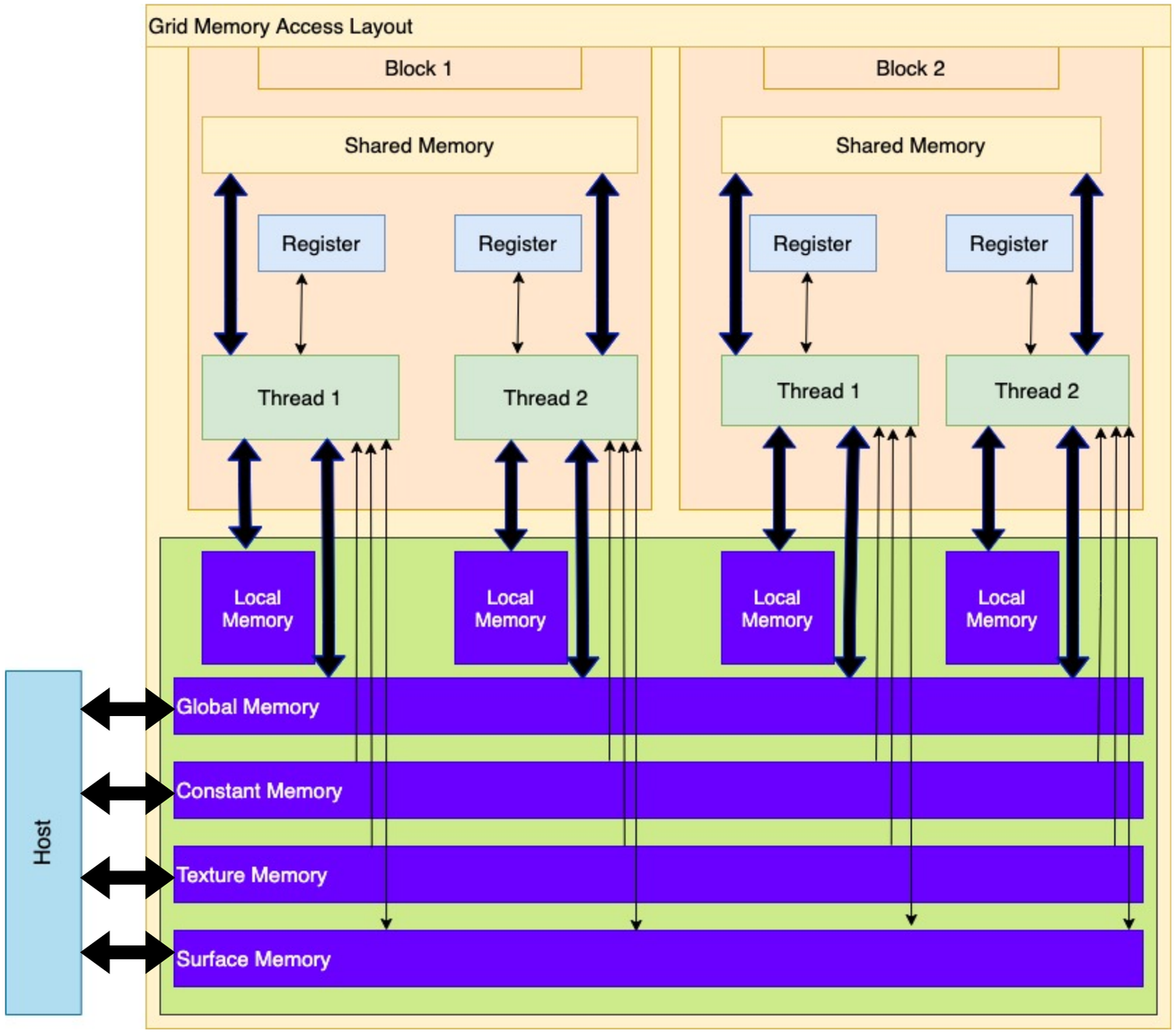
Exercise 4 : CUDA Matrix Multiplication Check

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?

Shared Memory
Matrix Multiplication with CUDA

GPU Memory Architecture



For K2200 GPU

Memory Type	Size	Latency (clock cycles)
Register	64 kB	1
Shared Memory	16 to 48 kB	1 to 32
Global Memory	4 GB	400 to 600

GPU Memory Architecture - Another View

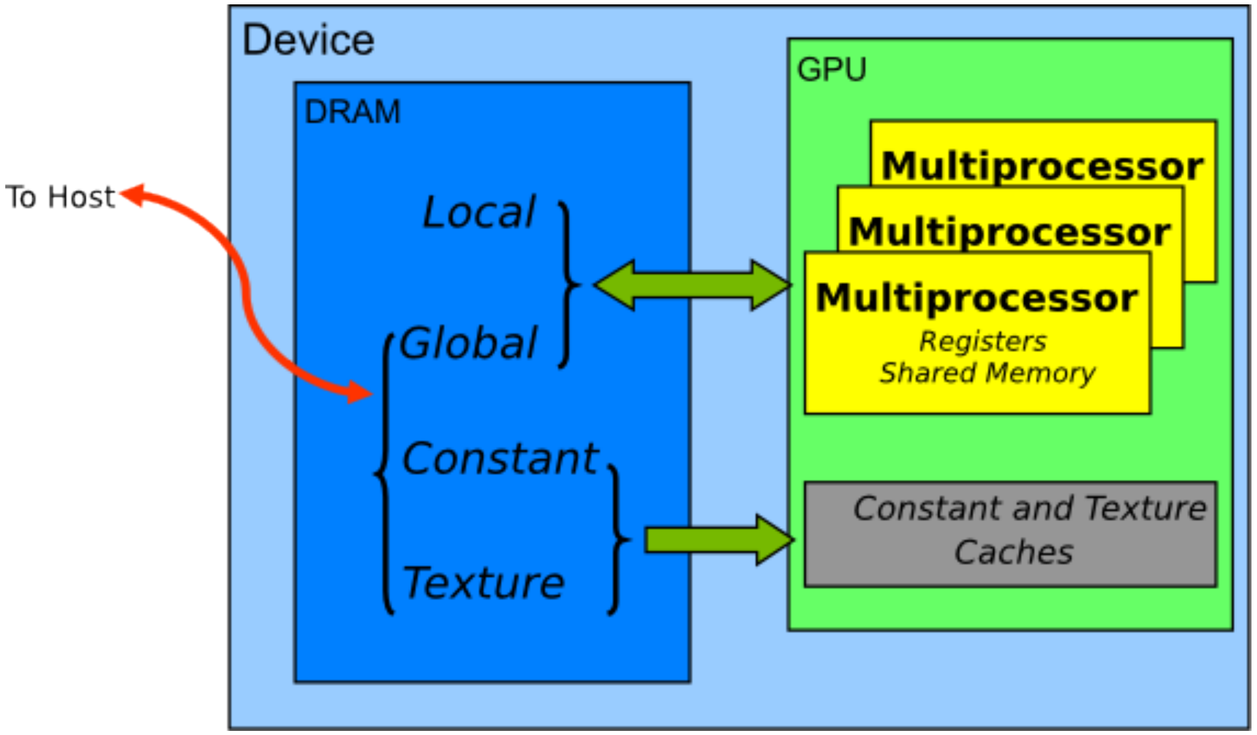


Table 1. Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

Source: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

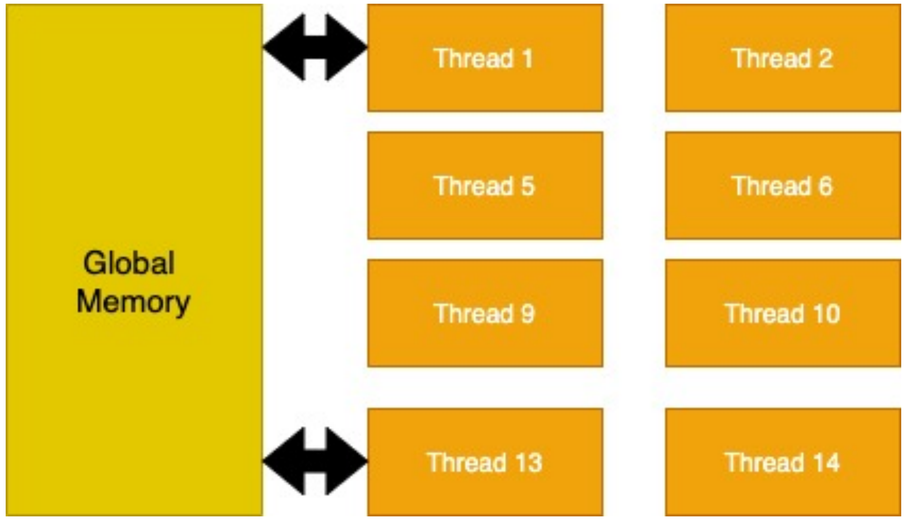
GPU Memory Hierarchy

Latency in clock cycles		Size
1	Register Memory (eg: SRAM)	64 KB
1 to 32	Shared Memory (eg: SRAM)	16 - 48 KB
Varies	Cache (eg: SRAM)	64 KB
400 to 600	Global Memory (eg DRAM)	4 GB
	Main Memory (e.g: DRAM)	
	Disk Memory (e.g: Magnetic Disk, Flash drive)	

Speed ↑ ↓ **Size**

Latency and size are for K2200 GPU

Global Memory



Size typically ranges from 4 up to 32 GB

What? Memory communication data between multiprocessors and device

How? __global__ float commondata[maxMem];

Who? All threads on the device, host, other GPUs (UVA)

Scope? Lifetime of application

Access? R+W

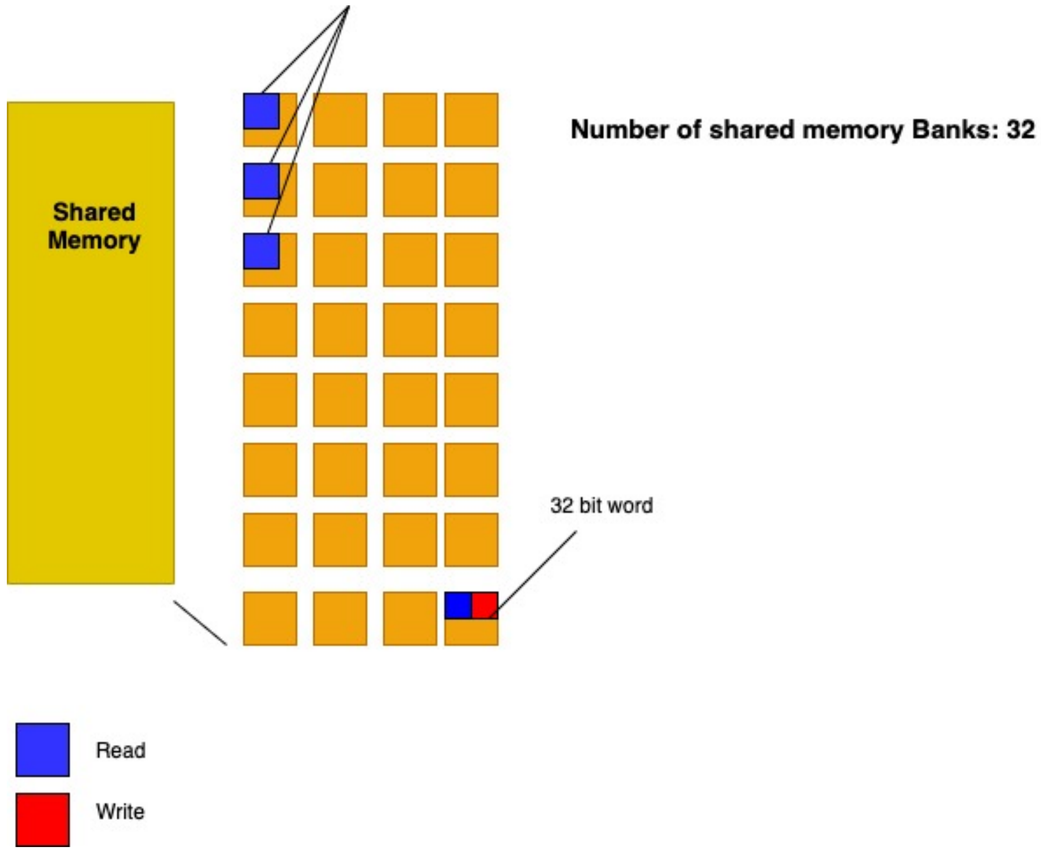
Problems? Bad alignment of data can slow down even further

Registers and Local Memory

Number of 32-bit registers per multiprocessor: 64k (except CC 3.7: 128k)

What?	Register is the memory for the ALUs, so it is on-chip and fast!
How?	Declare a variable, e.g: int counter;
Who?	A register is assigned to a single thread only
Scope?	Lifetime of thread
Access?	Read+Write, thread-private
Problems?	Affects occupancy and if a thread wants too many registers, they become spilled out to local memory (slow)

Shared Memory & L1 Cache



- Shared memory banks are equally sized memory modules, can be accessed simultaneously
- On the V100, the combined L1 cache and shared memory capacity is 128KB for each block
- Shared memory is configurable up to 96KB
- Programs that don't use shared memory can use all 128KB as L1 cache

Source: https://developer.nvidia.com/blog/inside-volta/

CUDA Shared Memory

Allocate shared memory with __**shared**__ keyword

```
// Statically allocate a tile of shared memory.
__shared__ float s_a[SHMEM_SIZE];
__shared__ float s_b[SHMEM_SIZE];
```

We want to make use of shared memory's speed for data that will be reused within a block. We also want to avoid inefficient memory access patterns from using global memory.

Data should be loaded into shared memory in a way that enables efficient memory accesses.

C/C++ stores data in "row-major" order
FORTRAN stores data in "column-major" order

Go to
Breakout Session 2

Return to main room:
~30 mins

Breakout rooms

Navigate to the **GPU_workshop/Lesson_4_SharedMem/CUDA/exercise** folder

Exercise: Matrix Multiplication with CUDA Shared Memory

Kernel Steps

- Compute each thread's global row and column index.
- Statically allocate a tile of shared memory. Tile size should equal the number of threads per block.
- Declare a temporary variable to accumulate calculated elements for the C matrix.
- Sweep tiles of size blockDim.x across matrices A and B. For matrix A, keep the row invariant and iterate through columns. For matrix B, keep the column invariant and iterate through rows.
- Load in elements from A and B into shared memory for each tile.
- Wait for tiles to be loaded in before doing computation.
- Do matrix multiplication on the small matrix within the current tile.
- Wait for all threads to finish using current tiles before loading in new ones.
- Write resulting calculation as an element of the C matrix.

To build use **./build.sh**

To submit use **sbatch submit.sh**

Exercise: Matrix Multiplication with CUDA Shared Memory Check

Compare the execution times of the different matrix multiplication implementations we've done so far.

Which implementation had the fastest GPU execution time for multiplying 1024 x 1024 matrices?

When might we see better performance using shared memory?

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