GPU Programming Workshop

Session 2: Introduction to CUDA

Special Technical Projects
&
Client Support Group

UCAR

April 13, 2021

Overview

Cheat Sheet (link in chat):
https://docs.google.com/document/d/1NCNkZM3reXjPaH5YL10CM2EcOH55hMucCWMXHe_2iAk/edit?usp=sharing

- Connect to Casper, clone and update your forked repository (instructions are on Cheat Sheet).
- 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 on local:
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>
```

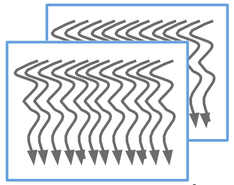
Find your instance IP and username here or ask your mentor later:
https://docs.google.com/spreadsheets/d/1m0W_u4XNAT2j6XjQr4hwElbft2OX_5xwE9zEuR91-3Q/edit?usp=sharing

Software Abstraction

Software term

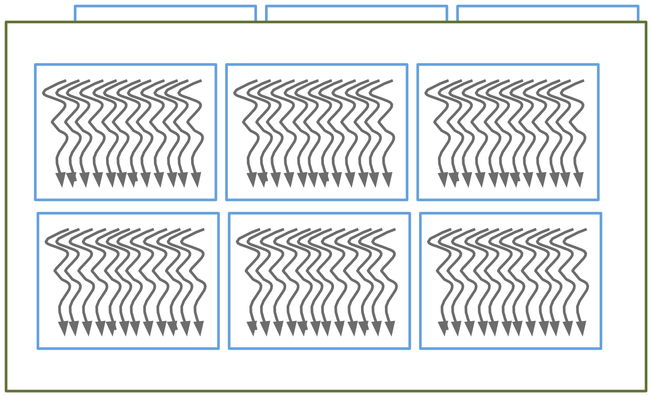
Threads

Blocks



512–1024 threads / block

Grids

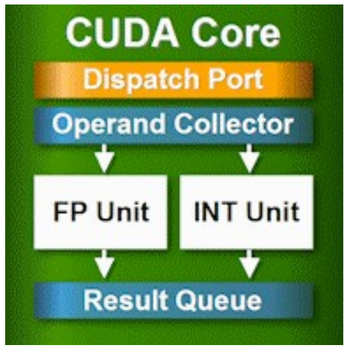


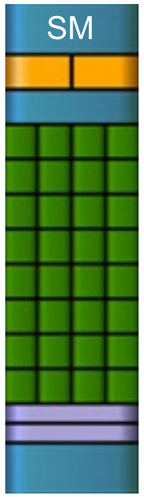
Kernel is executed by threads processed by CUDA Core

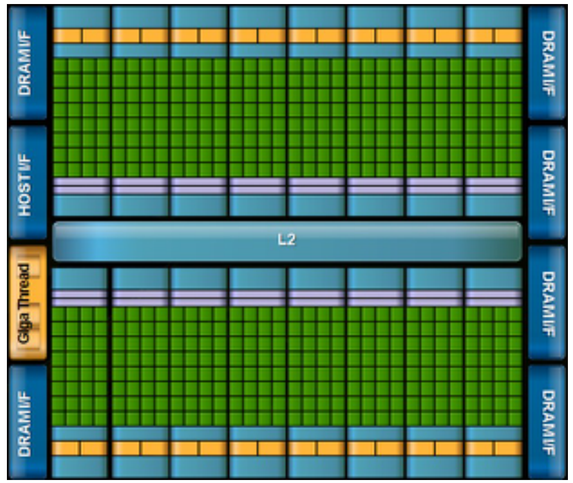
Maximum 8 blocks per SM
32 parallel threads are executed at the same time in a *WARP*

One grid per kernel with multiple concurrent kernels

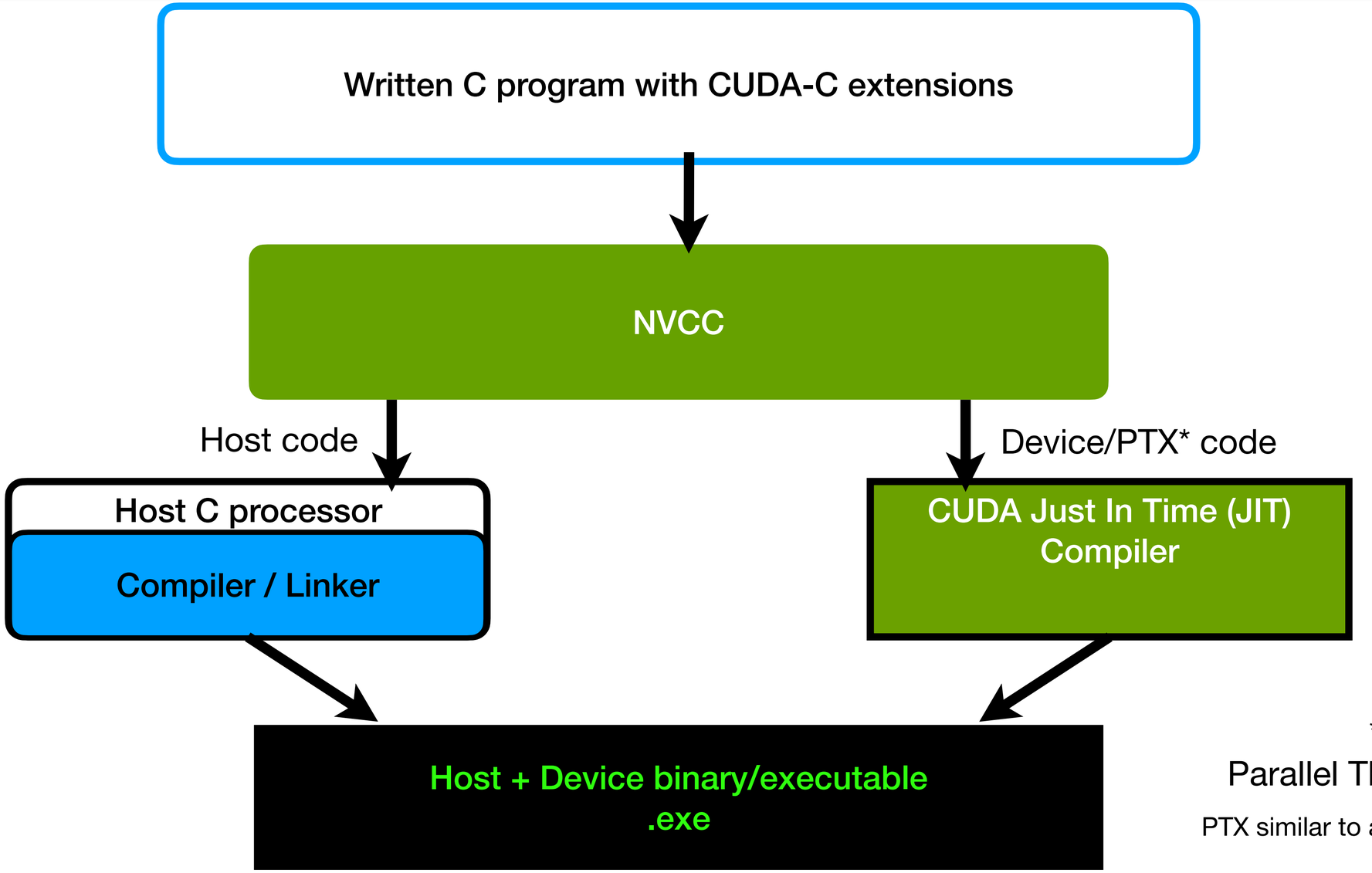
Hardware term







CUDA Compilation



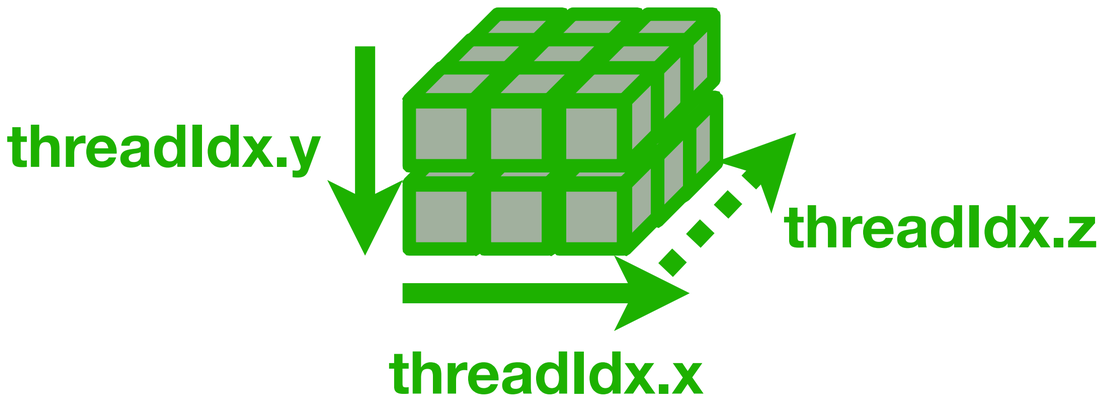
*PTX =
Parallel Thread Execution

PTX similar to assembly code on CPU

Understanding CUDA Threads Blocks and Grids

Threads, Blocks, and Grids

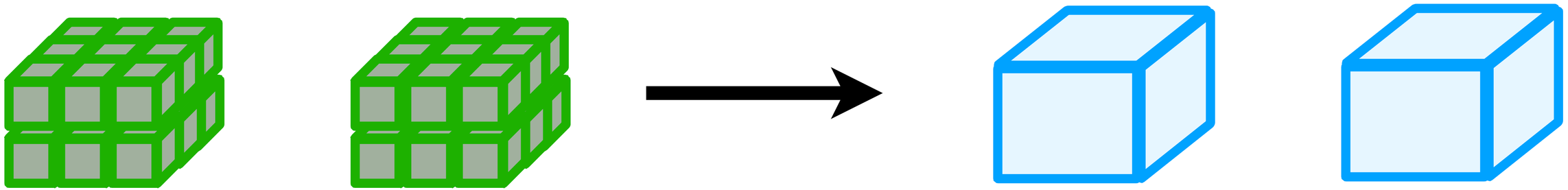
Threads on the GPU are grouped into thread **Blocks** and then a **Grid** of blocks. A thread is uniquely identified within a block based on its index in each dimension: **threadIdx.x**, **threadIdx.y**, and **threadIdx.z**.



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

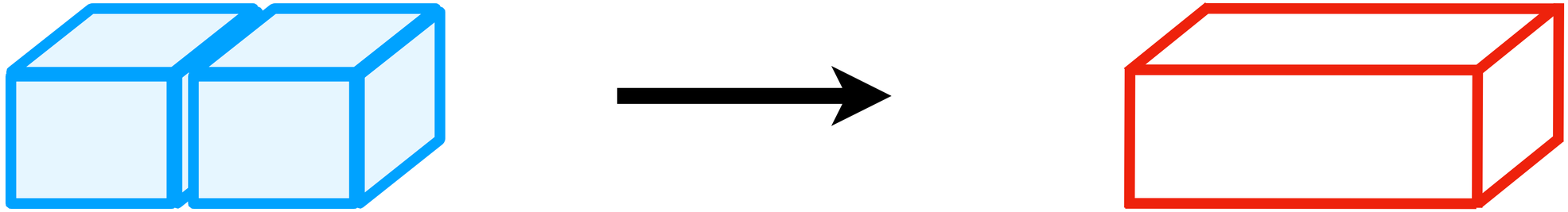
Becomes 2 blocks each with **blockDim** = **{x,y,z}** = {3,2,3}

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 **linear addresses** of the thread.

Threads, Blocks, and Grids - An Aside

threadIdx.x
threadIdx.y
threadIdx.z = 0

Defines what position a thread is in within a block

blockDim.x
blockDim.y
blockDim.z = 1

Defines how many threads there are within a block

blockIdx.x
blockIdx.y
blockIdx.z = 0

Defines what position a block is within a grid

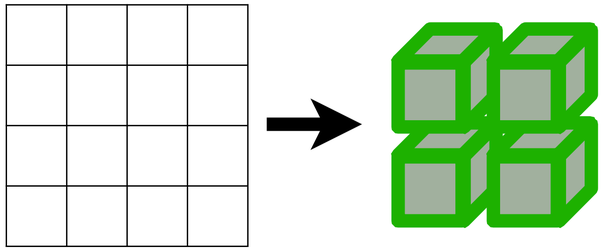
gridDim.x
gridDim.y
gridDim.z = 1

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 **block linear address** (BLA) is used to uniquely identify a thread within a block. Multiple threads can have the same BLA.

To uniquely identify a thread within a block **blockDim** needs to be combined with the **threadIdx**.

BLA = **threadIdx.x** + **blockDim.x** * **threadIdx.y** + **blockDim.x** * **blockDim.y** * **threadIdx.z**

E.g. **threadIdx** {x,y} = {3,4} with **blockDim** {x,y} = {5,3} has BLA = 3 + 5 * 4 = 23

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 typically 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**

.

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**~~

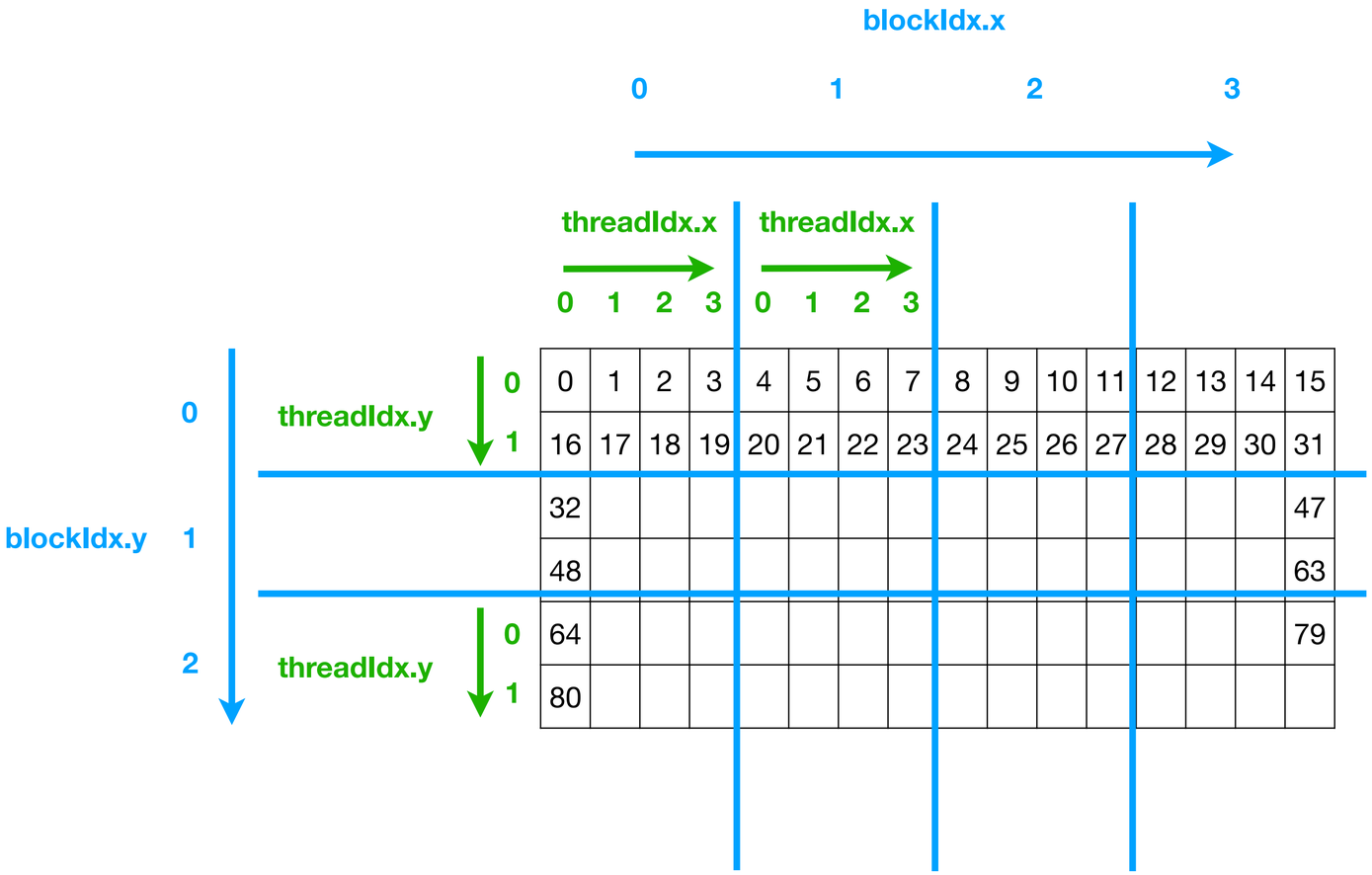
0 * 1 + 0 = 0

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 **6x16** matrix and the threads per block is chosen as:

blockDim {x, y} = {4, 2} giving **gridDim** {x, y} = {4, 3}

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

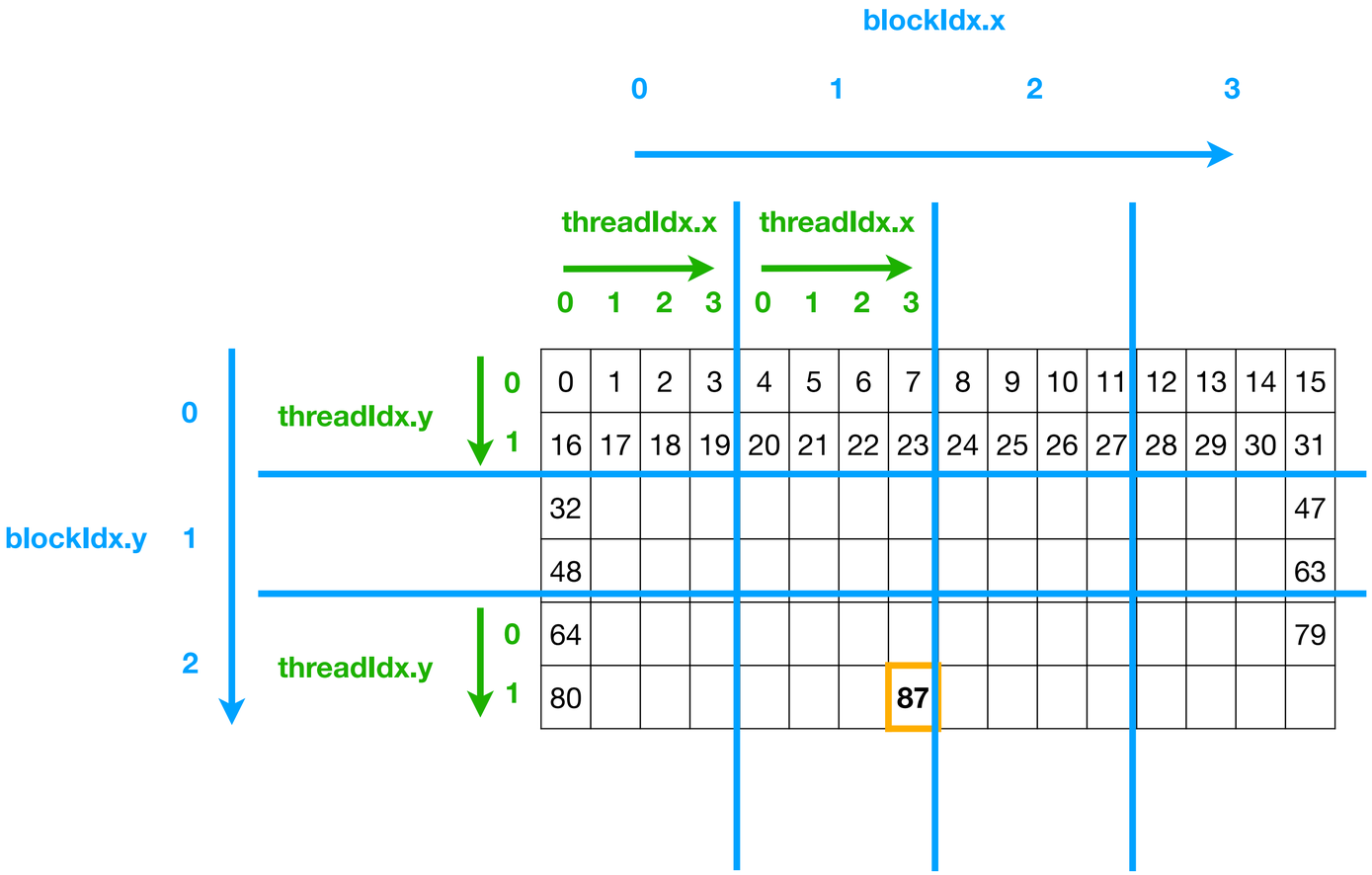
Calculation

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 **16x6** matrix and the threads per block is chosen as:

blockDim {x, y} = {4, 2} giving **gridDim** {x, y} = {4, 3}

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

Breakout Session 1

- **Task 2: Coding Practice - CUDA Matrix Multiplication**

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;
```

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

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

Shared Memory
Matrix Multiplication with CUDA

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**

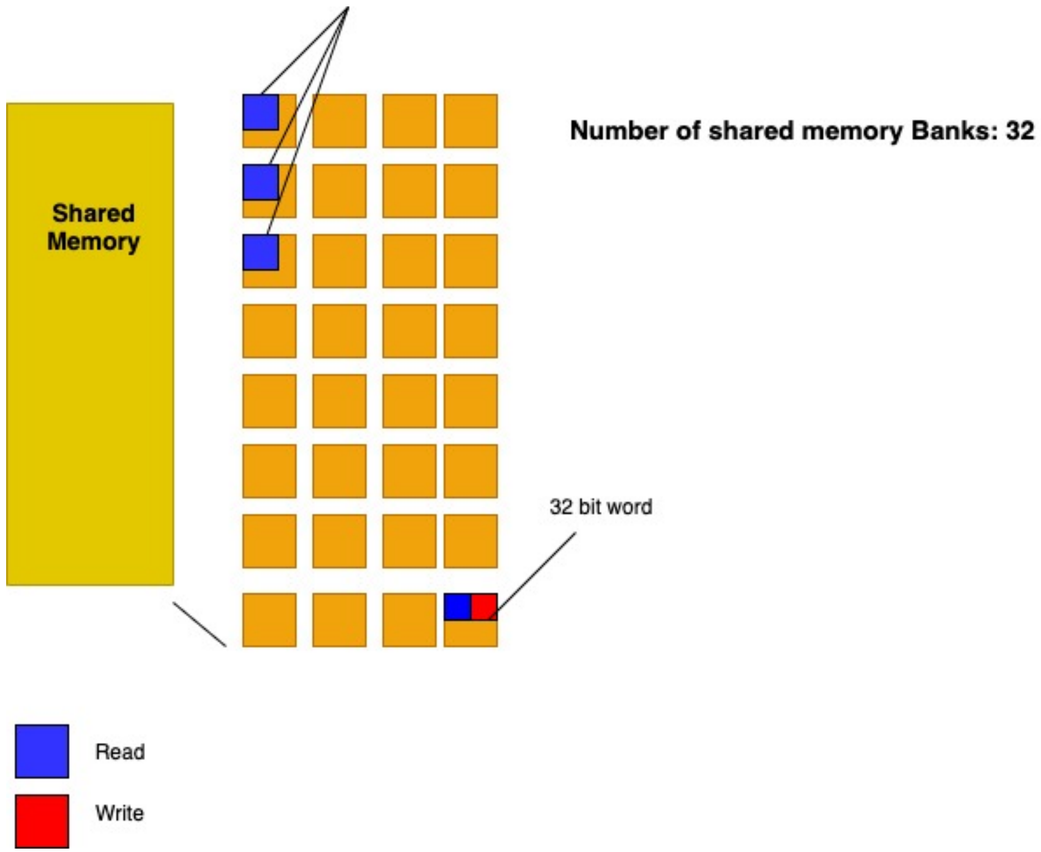
Latency and size are for K2200 GPU

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)

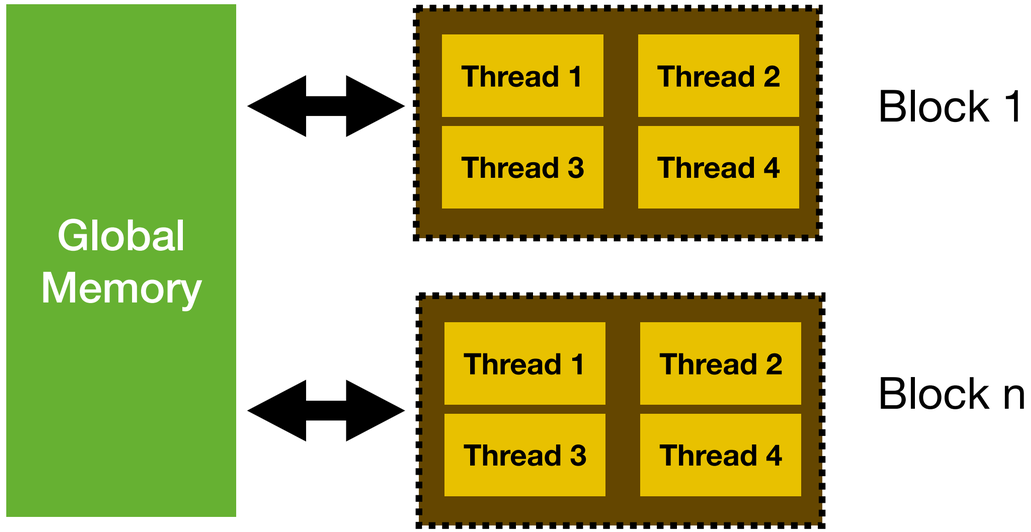
Shared Memory & L1 Cache



- 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/

Global Memory



Size typically ranges from 4 up to 32 GB

What?	Memory communication data between multiprocessors and device
How?	CudaMemAlloc( ) function
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

CUDA Shared Memory

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

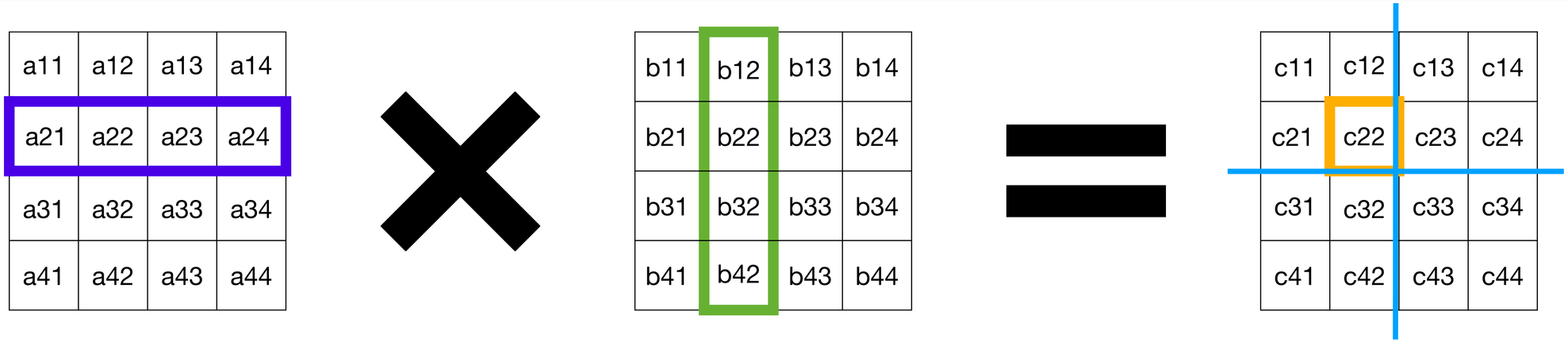
```
// Statically allocate a tile of shared memory.
__shared__ float s_a[SHMEM_SIZE];
__shared__ float s_b[SHMEM_SIZE];
```

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

Shared Memory Matrix Multiplication



A
MxP = 4x4

B
PxQ = 4x4

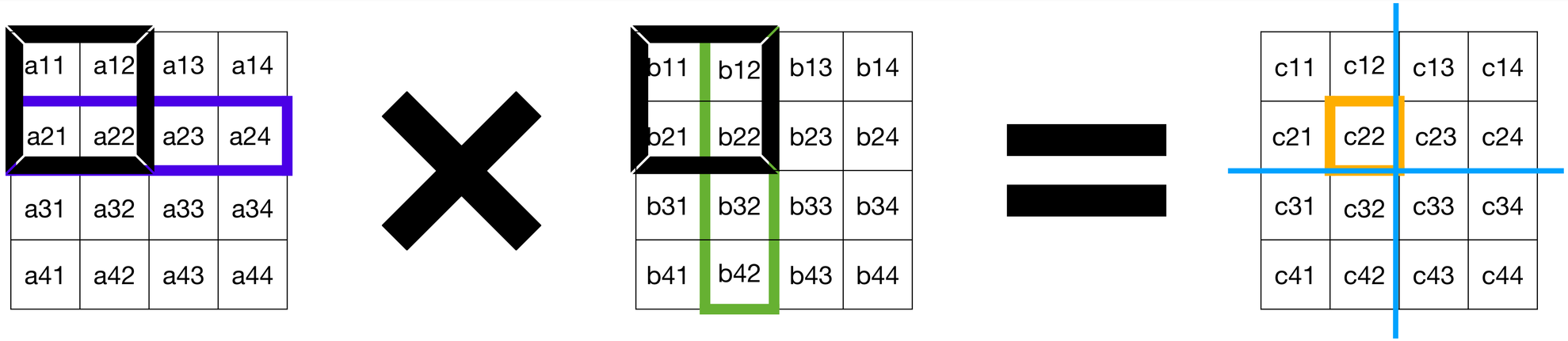
C
MxQ=4x4

If we want to find element c22, we look at a dot product between row 2 from matrix A and column 2 from matrix B

c22 = a21 * b12 + a22 * b22 + a23 * b32 + a24 * b42

We divide this problem up based on matrix C. If we use **blockDim** {x, y} = {2,2} for these 4x4 matrices we get this division.

Shared Memory Matrix Multiplication



A
MxP = 4x4

B
PxQ = 4x4

C
MxQ=4x4

If we want to find element c22, we look at a dot product between row 2 from matrix A and column 2 from matrix B

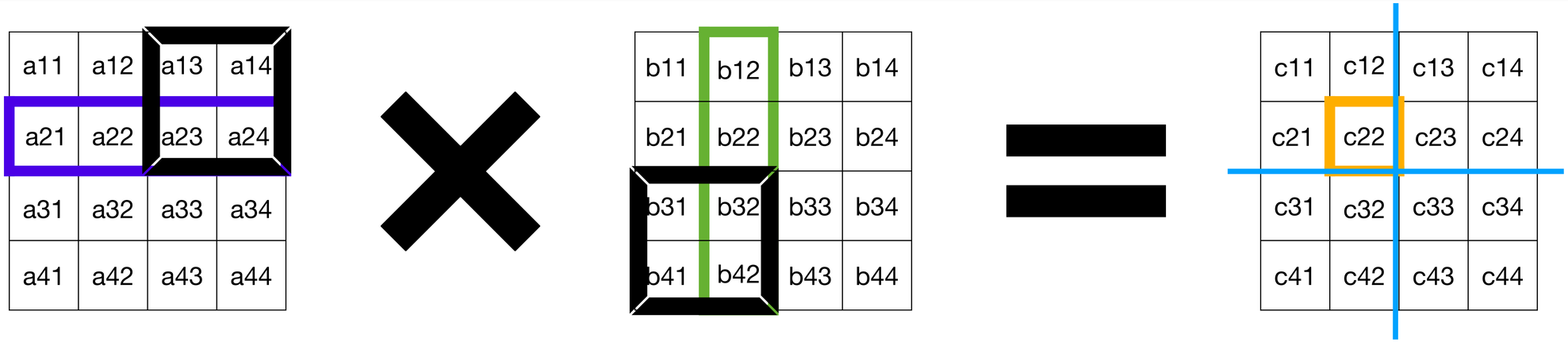
c22 = a21 * b12 + a22 * b22 + a23 * b32 + a24 * b42

We divide this problem up based on matrix C. If we use **blockDim** {x, y} = {2,2} for these 4x4 matrices we get this division.

We will "tile" across matrices A and B using tiles the same size as a block. This tiling will allow us to compute parts of the dot product and allow for some data reuse.

c22 = **[**a21 * b12 + a22 * b22**]** + ...

Shared Memory Matrix Multiplication



A
MxP = 4x4

B
PxQ = 4x4

C
MxQ=4x4

If we want to find element c22, we look at a dot product between row 2 from matrix A and column 2 from matrix B

c22 = a21 * b12 + a22 * b22 + a23 * b32 + a24 * b42

We divide this problem up based on matrix C. If we use **blockDim** {x, y} = {2,2} for these 4x4 matrices we get this division.

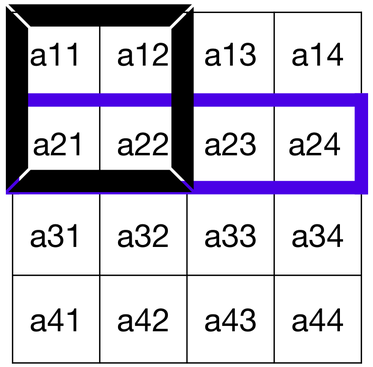
We will "tile" across matrices A and B uses tiles the same size as a block. This tiling will allow us to compute parts of the dot product and allow for some data reuse.

c22 = **[**a21 * b12 + a22 * b22**]** + ...
[a23 * b32 + a24 * b42**]**

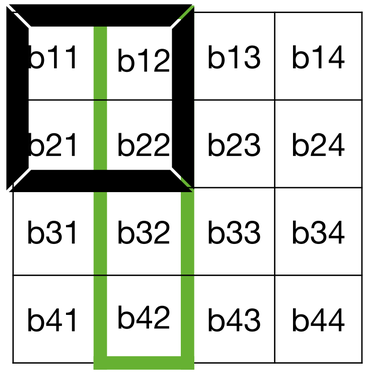
Shared Memory Matrix Multiplication - Our Kernel

The shared memory multiplication kernel involves 8 main steps

1. Tiles are created by each block in shared memory that will hold sub-matrices of A and B. Call these s_A and s_B. Each tile starts in the leftmost columns for s_A and in the topmost rows for s_B
2. Each block copies elements from A into its s_A and from B into its s_B
3. All threads in a block wait after Step 2 to ensure the data is loaded into the tile
4. Each thread calculates its part of the dot product using the tile data
5. Wait for all threads in a block to finish Step 5
6. The s_A tile for each block is shifted 1 tile to the right and s_B is shifted 1 tile down
7. Steps 2 through 6 are repeated until each part of the dot product is calculated
8. Each thread writes the completed dot product to the C matrix.

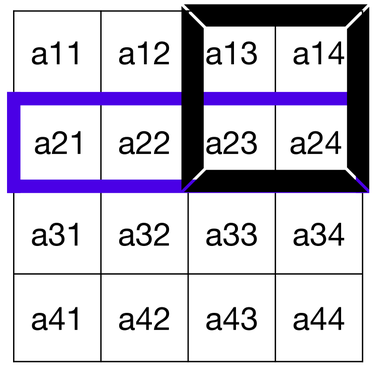


A
MxP = 4x4

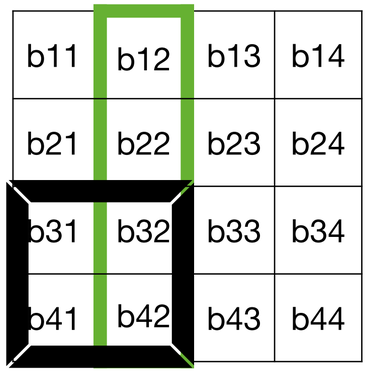


B
PxQ = 4x4

After an iteration
s_A tiles shift right
and
s_B tiles shift down



A
MxP = 4x4



B
PxQ = 4x4

Go to
Breakout Session 2

Return to main room:
~30 mins

References

- Participant Cheat Sheet:
https://docs.google.com/document/d/1NCNkZM3reXjPaH5YL10CM2EcOH55hMucCWMXHe_2iAk/edit?usp=sharing

- GitHub Repo:
https://github.com/NCAR/GPU_workshop.git

- CUDA Syntax Cheat Sheet:
https://kapeli.com/cheat_sheets/CUDA_C.docset/Contents/Resources/Documents/index