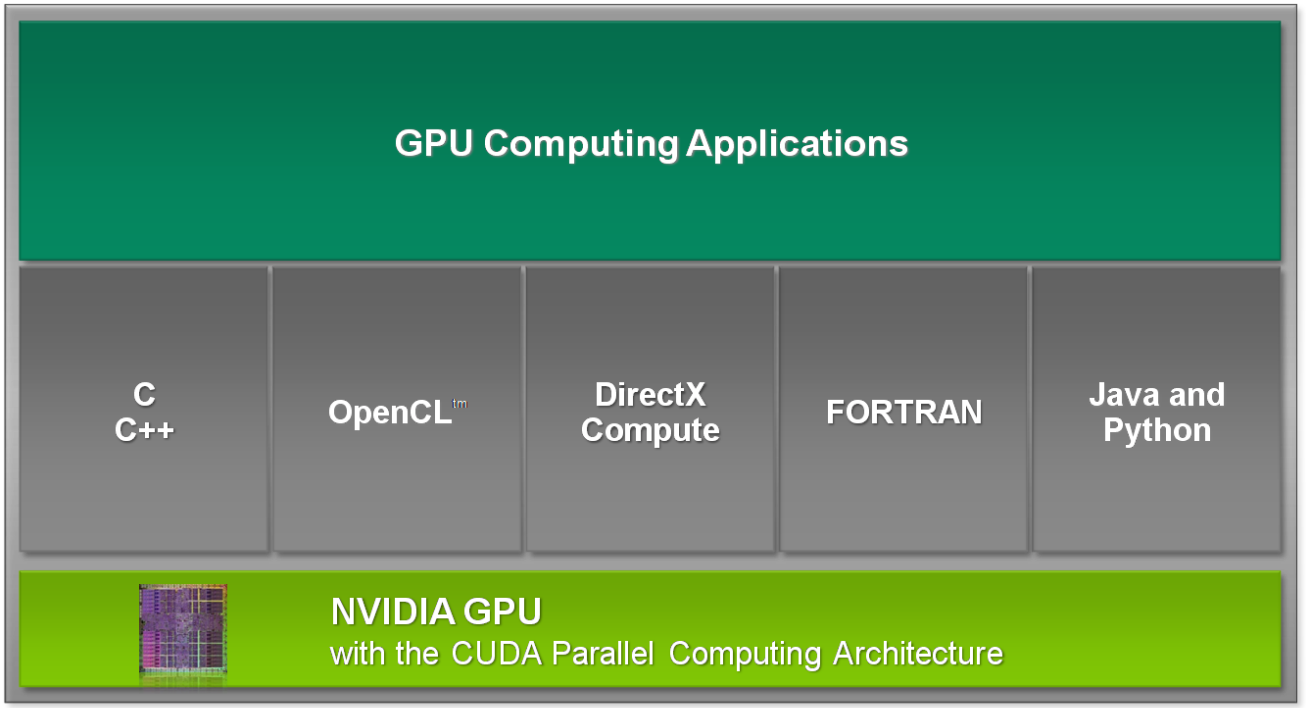
INTRODUCTION TO CUDA

- **CUDA is the acronym for Compute Unified Device Architecture**.

Enter CUDA

CUDA is NVIDIA's general purpose parallel computing architecture .

- designed for calculation-intensive computation on GPU hardware
- CUDA is not only a language, it is an API
- we will mostly concentrate on the C implementation of CUDA

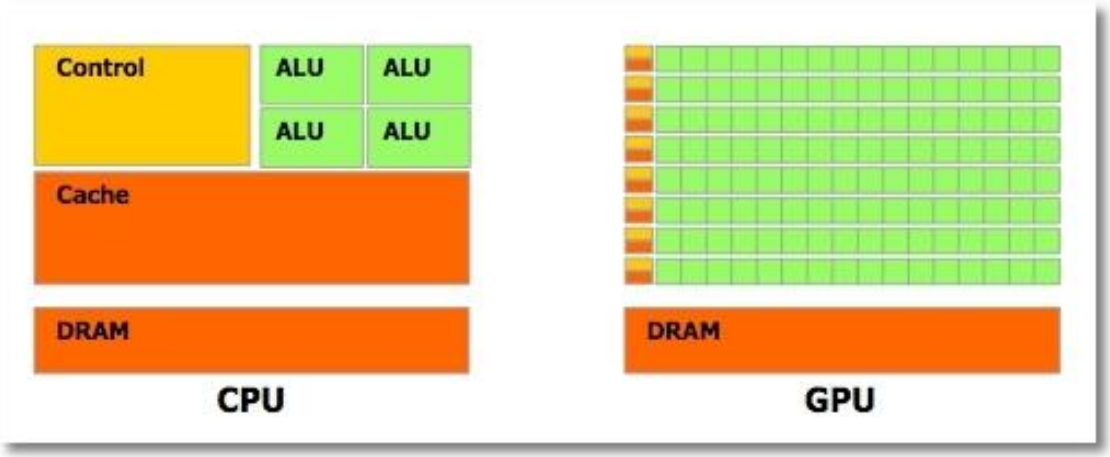


CUDA

- Programing system for machines with GPUs
  - Programming Language
  - Compilers
  - Runtime Environments
  - Drivers
  - Hardware

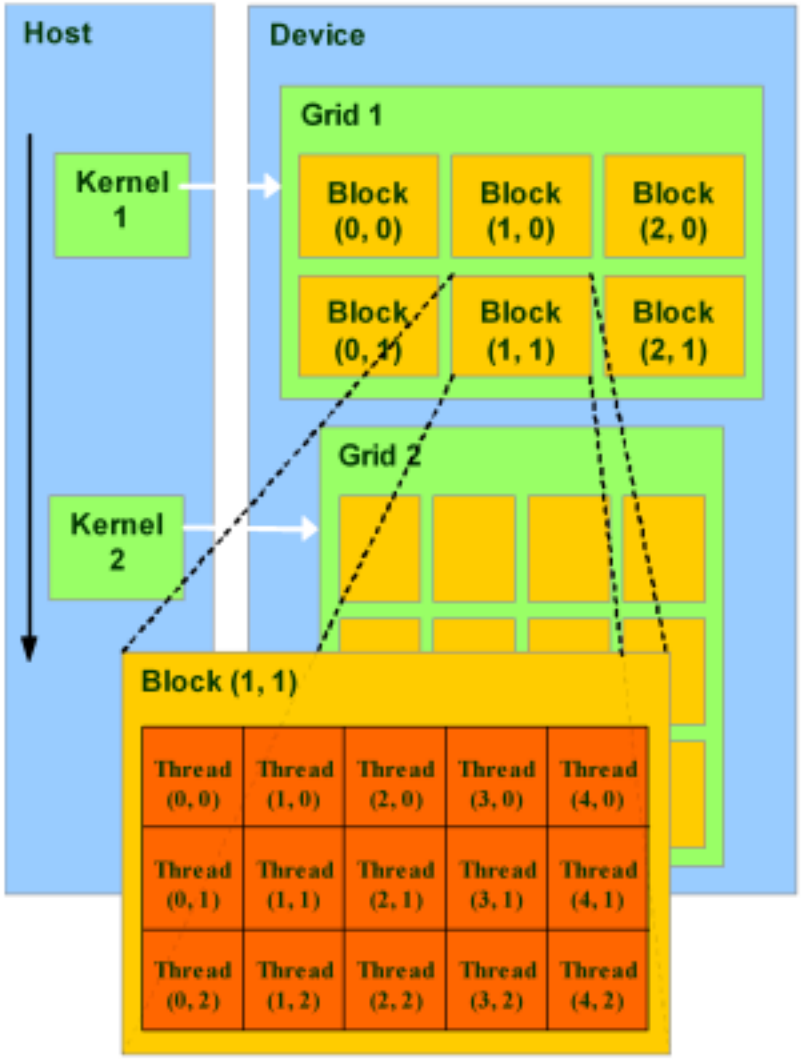
CUDA Goals

- Scale code to hundreds of cores running thousands of threads
- The task runs on the gpu independently from the cpu



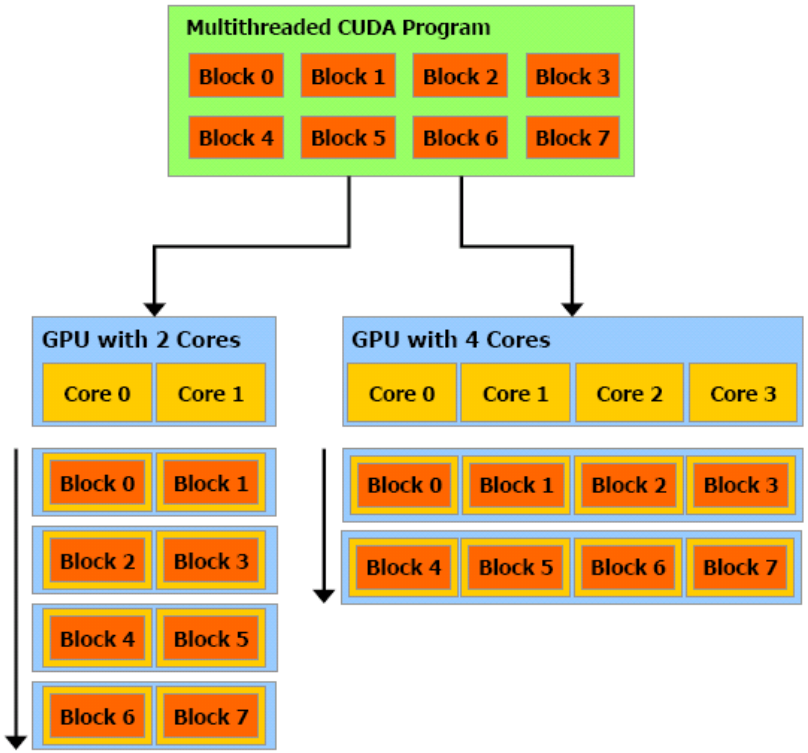
CUDA Structure

- Threads are grouped into thread blocks
- Blocks are grouped into a single grid
- The grid is executed on the GPU as a kernel



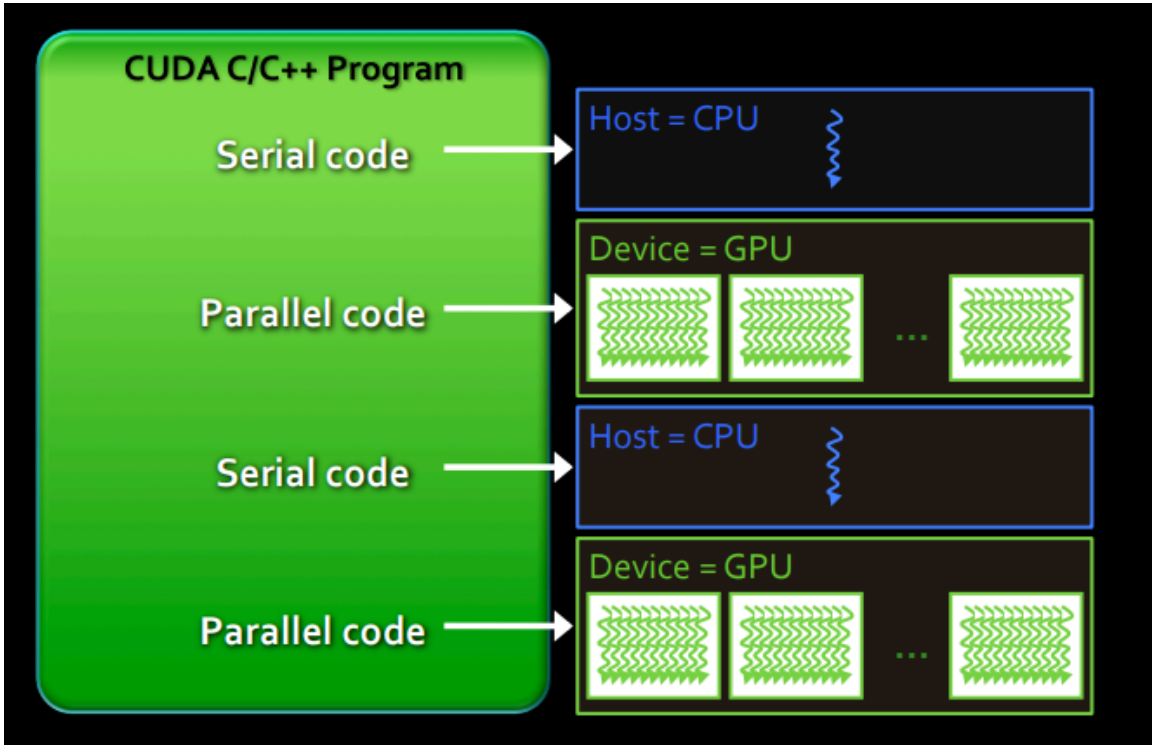
Scalability

- Blocks map to cores on the GPU
- Allows for portability when changing hardware

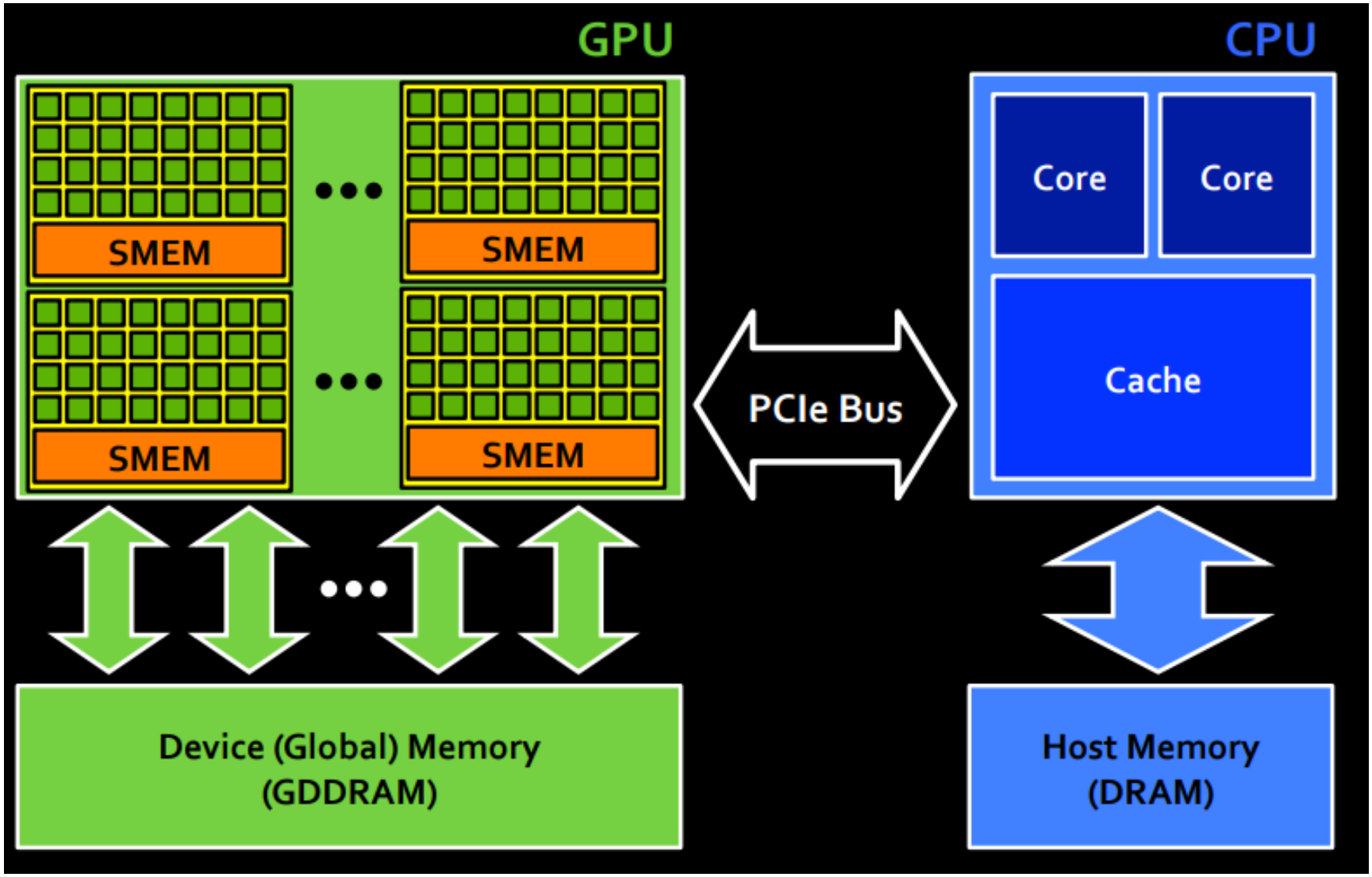


Behavior of CUDA program

- Serial code executes in Host (CPU) thread
- Parallel code executes in many concurrent Device (GPU) threads across multiple parallel processing elements



Execution flow



Terms and Concepts

Each block and thread has a unique id within a block.

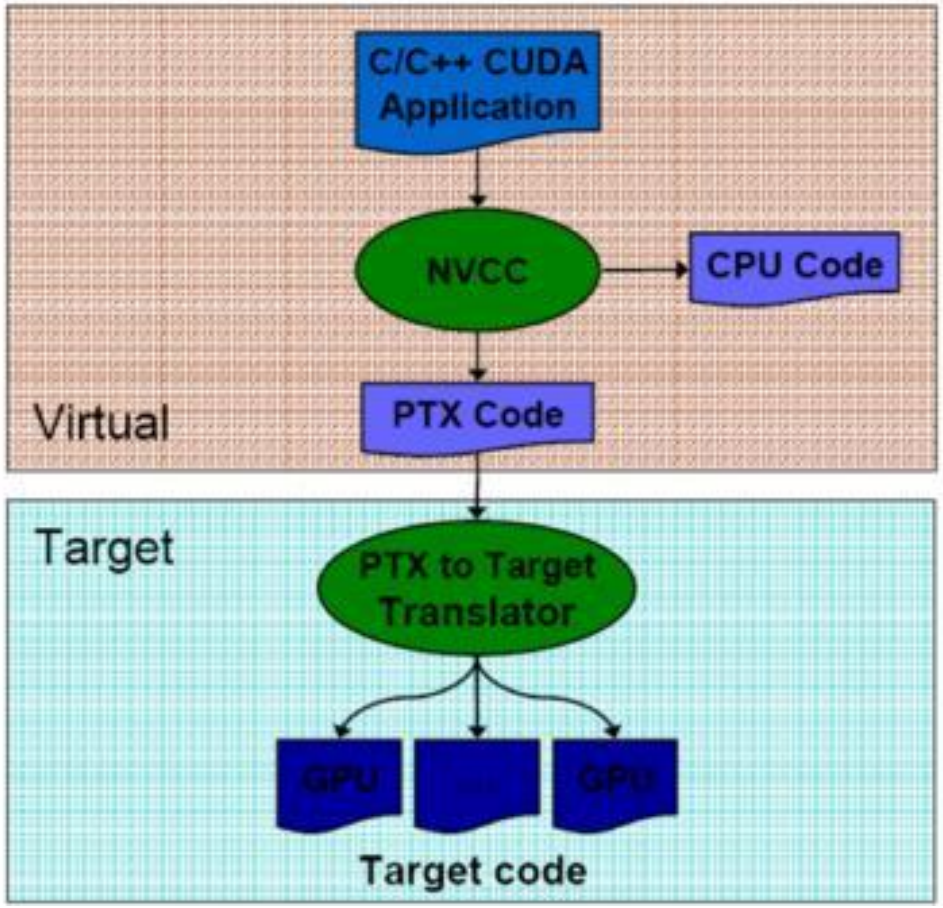
- threadIdx – identifier for a thread
- blockIdx – identifier for a block
- blockDim – size of the block

Unique thread id:

(blockIdx*blockDim)+threadIdx

NVCC compiler

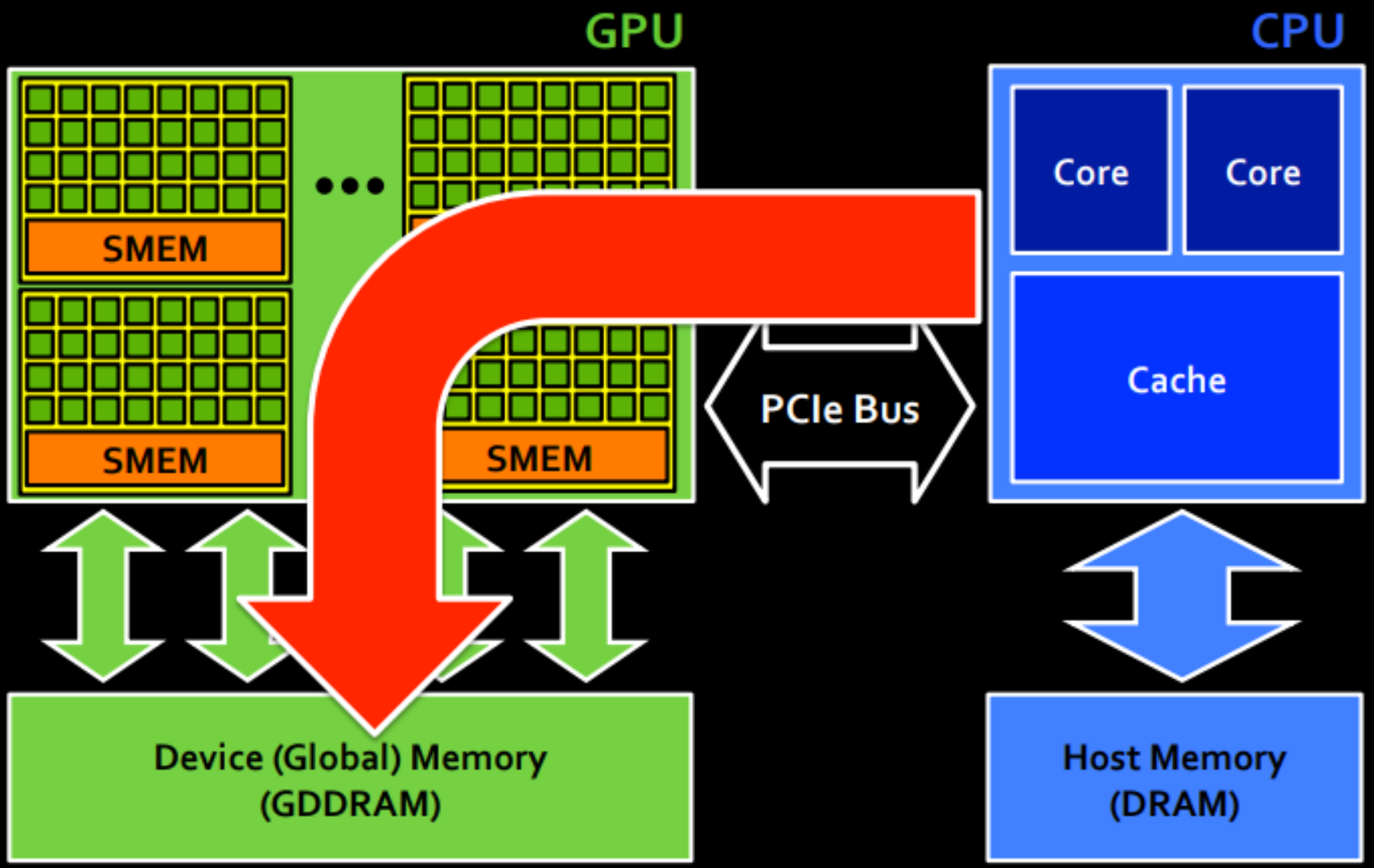
- Compiles C or PTX code (CUDA instruction set architecture)
- Compiles to either PTX code or binary (cubin object)



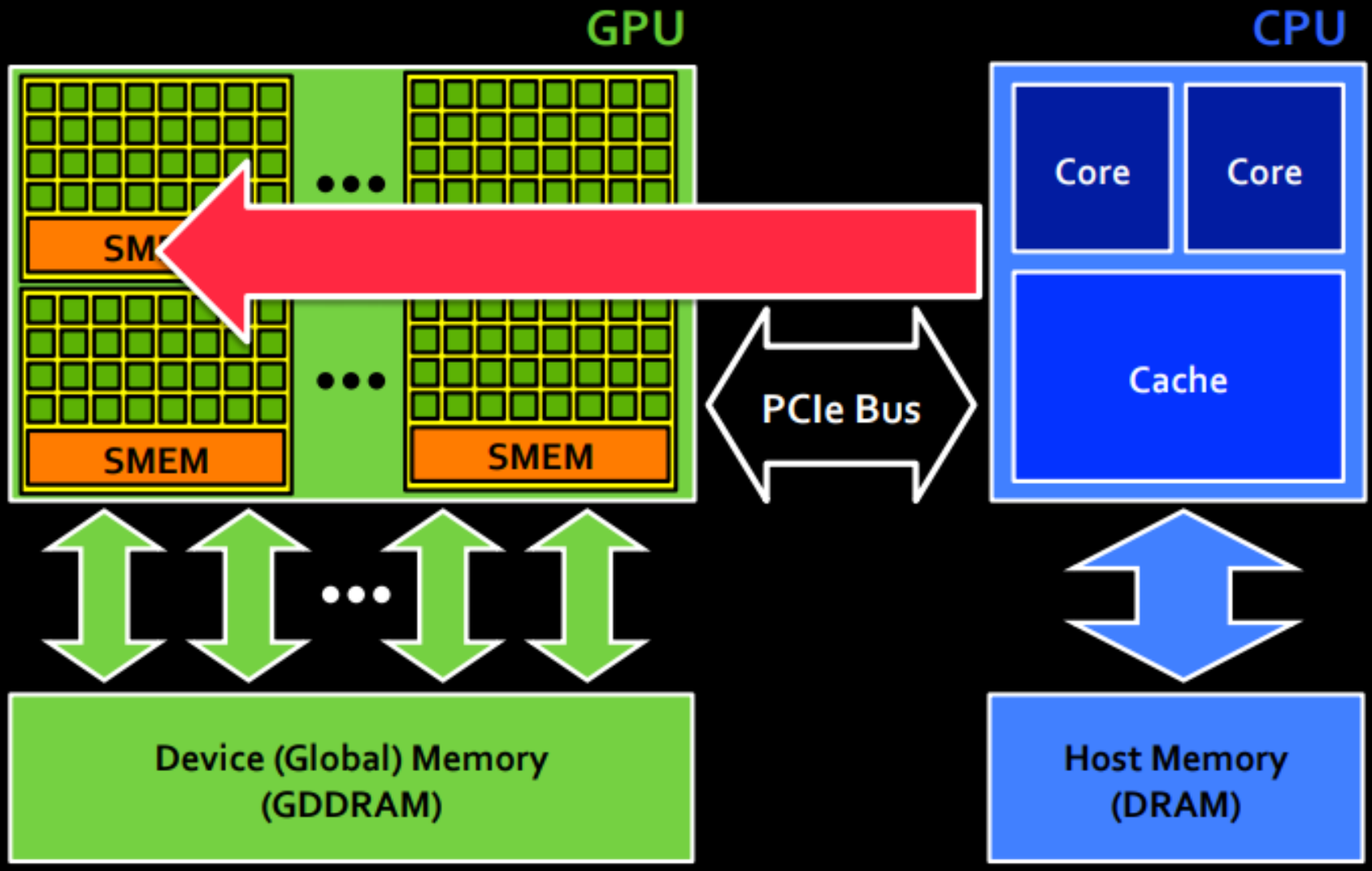
Development: Basic Idea

1. Allocate equal size of memory for both host and device
2. Transfer data from host to device
3. Execute kernel to compute on data
4. Transfer data back to host

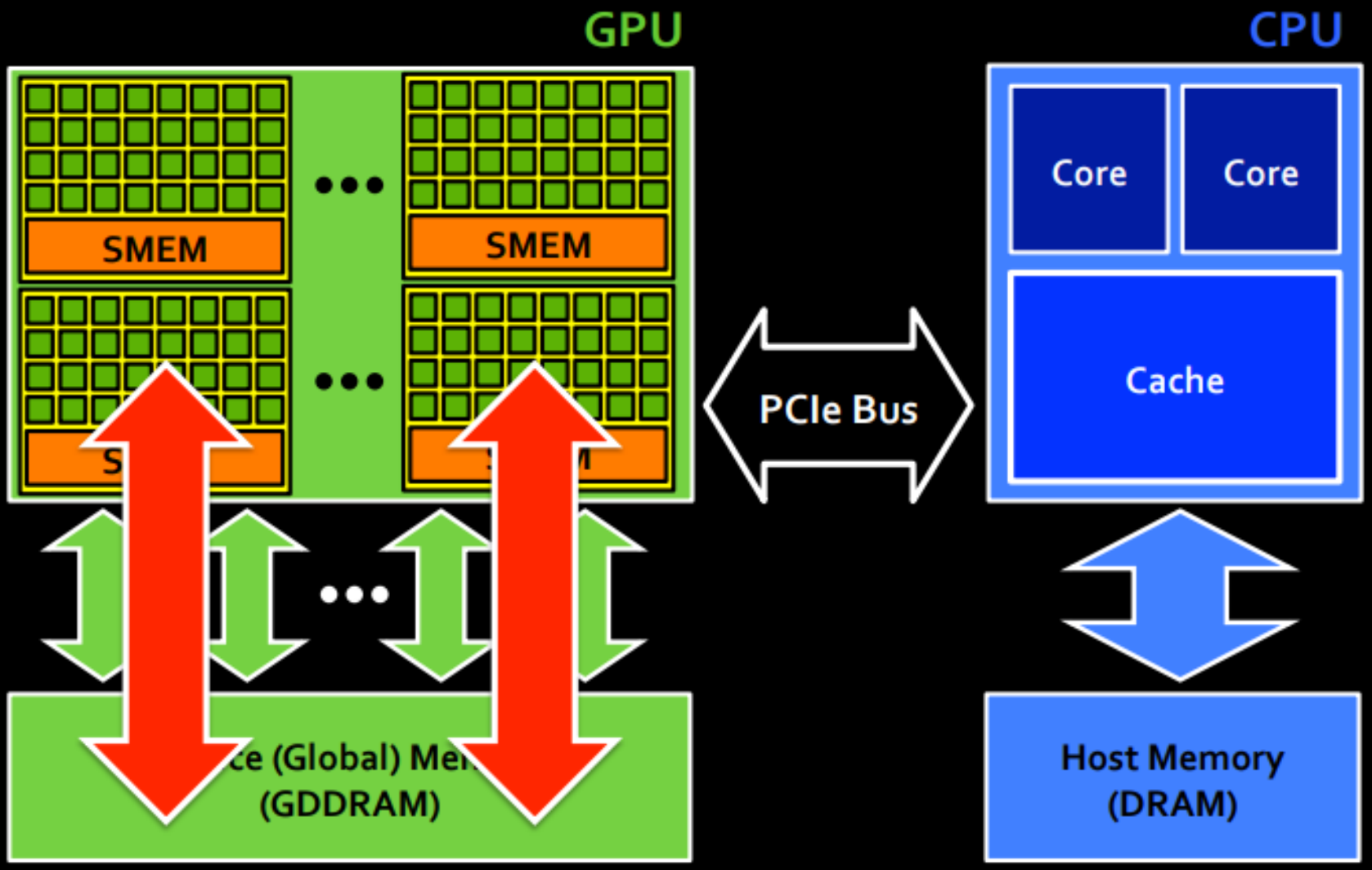
Step 1 – copy data to GPU memory



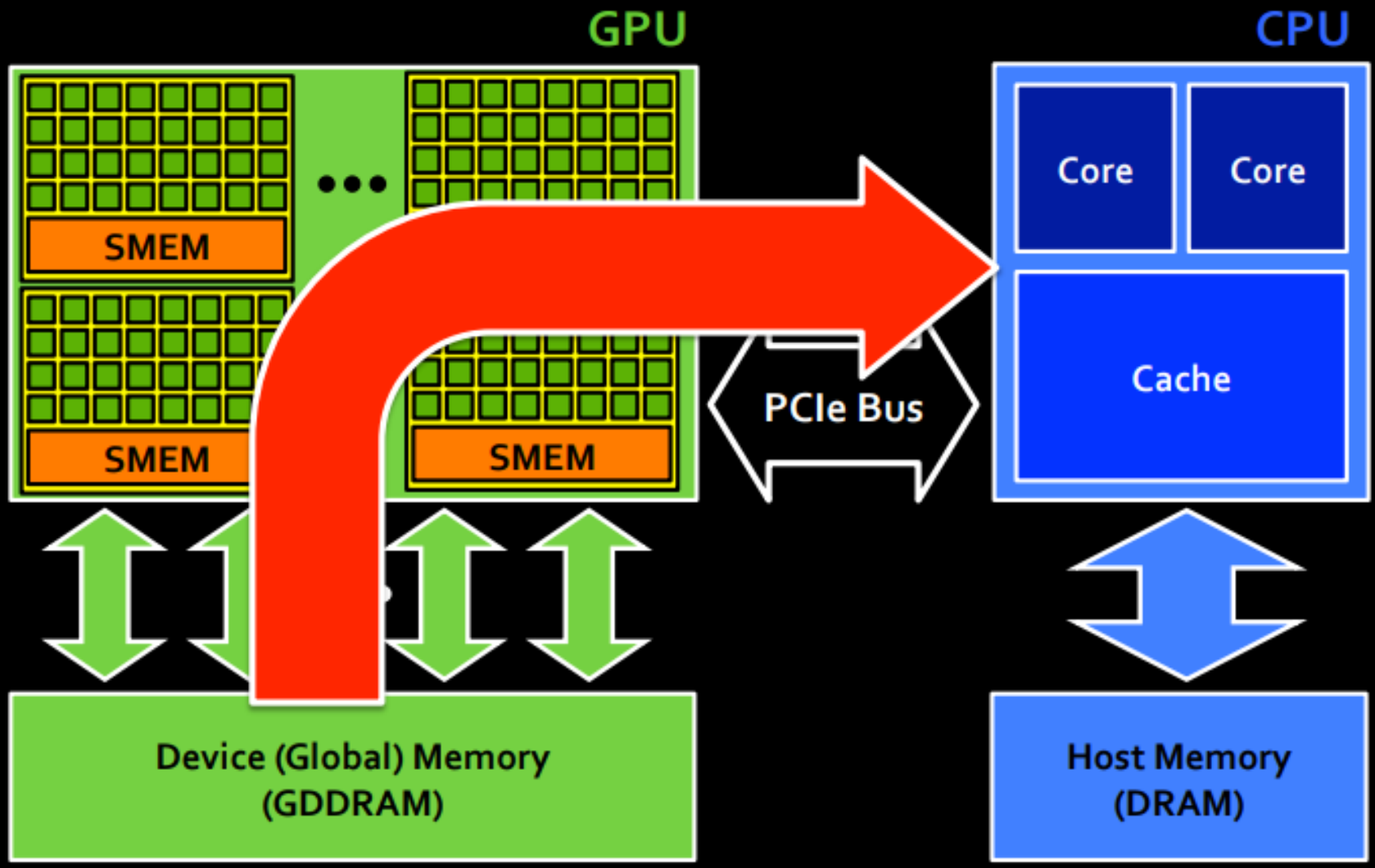
Step 2 – launch kernel on GPU



Step 3 – execute kernel on GPU

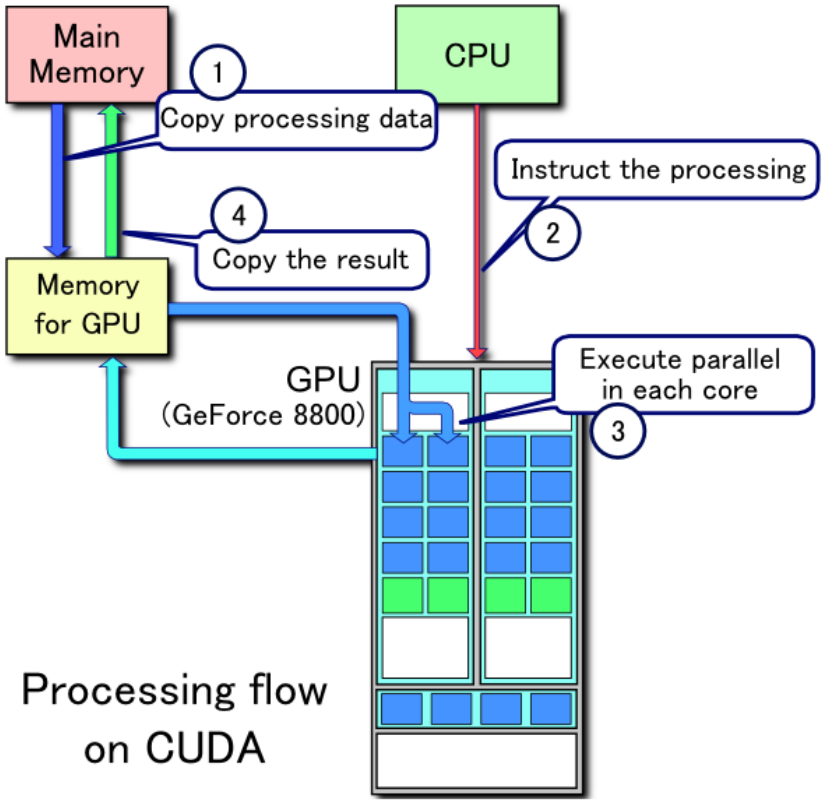


Step 4 – copy data to CPU memory



Processing Flow

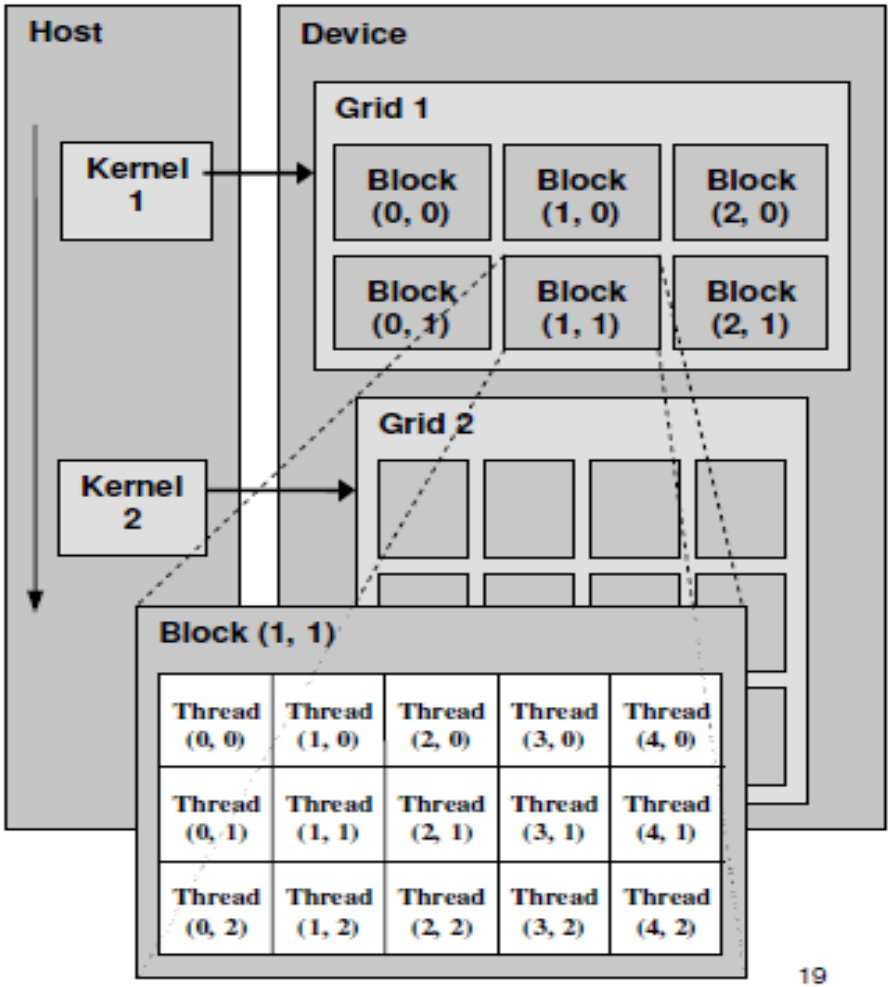
- Processing Flow of CUDA:
  - Copy data from main mem to GPU mem.
  - CPU instructs the process to GPU.
  - GPU execute parallel in each core.
  - Copy the result from GPU mem to main mem.



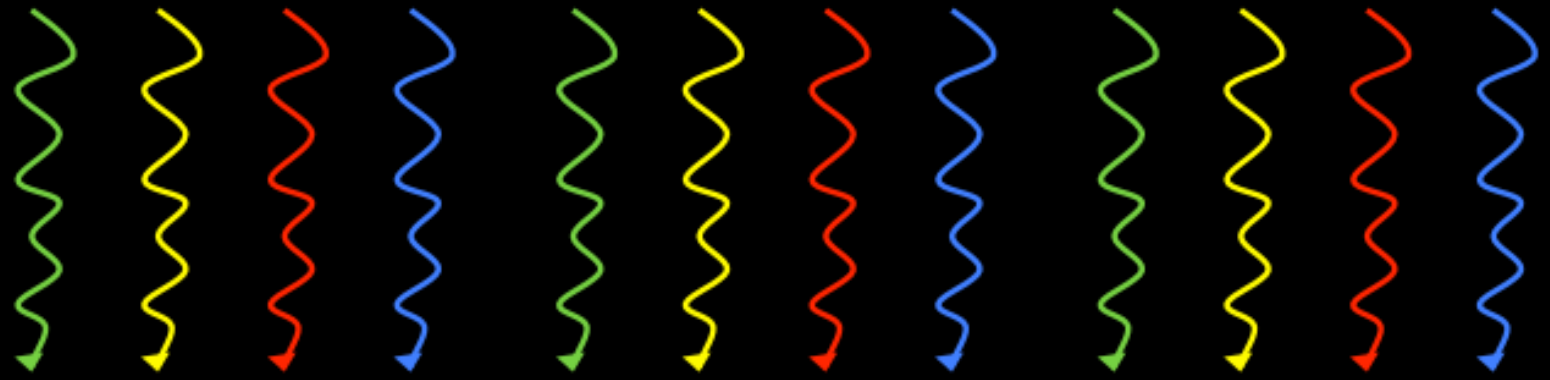
CUDA ARCHITECTURE

CUDA Programming Model

Device = GPU
Host = CPU
Kernel = function that runs on the device

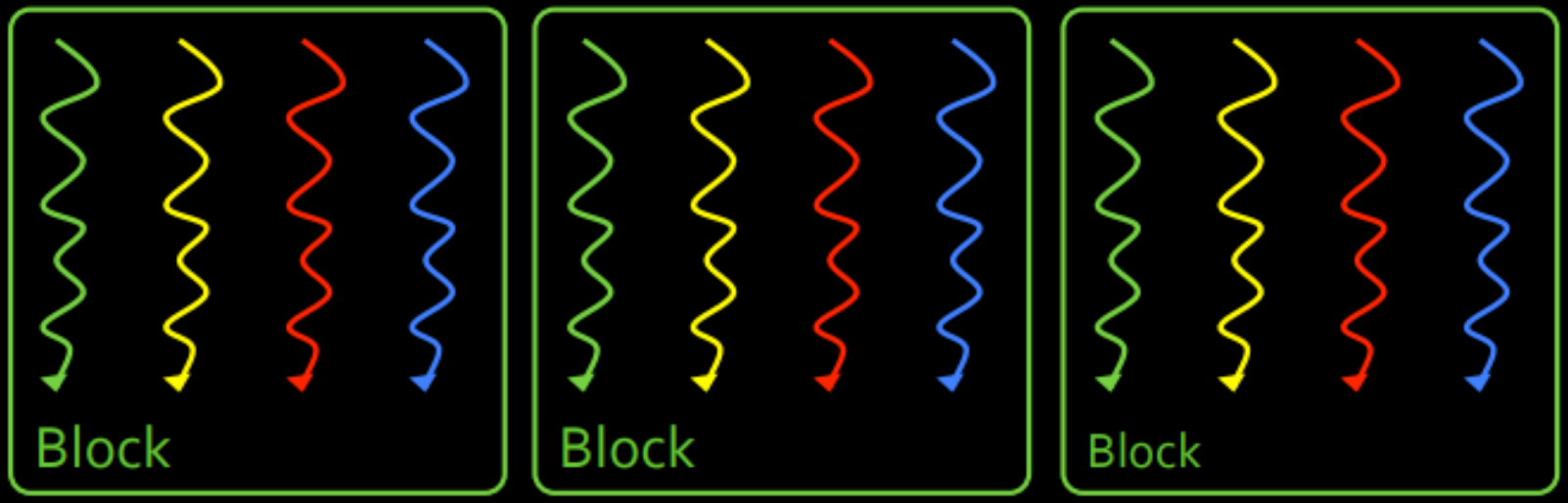


CUDA Thread Organization



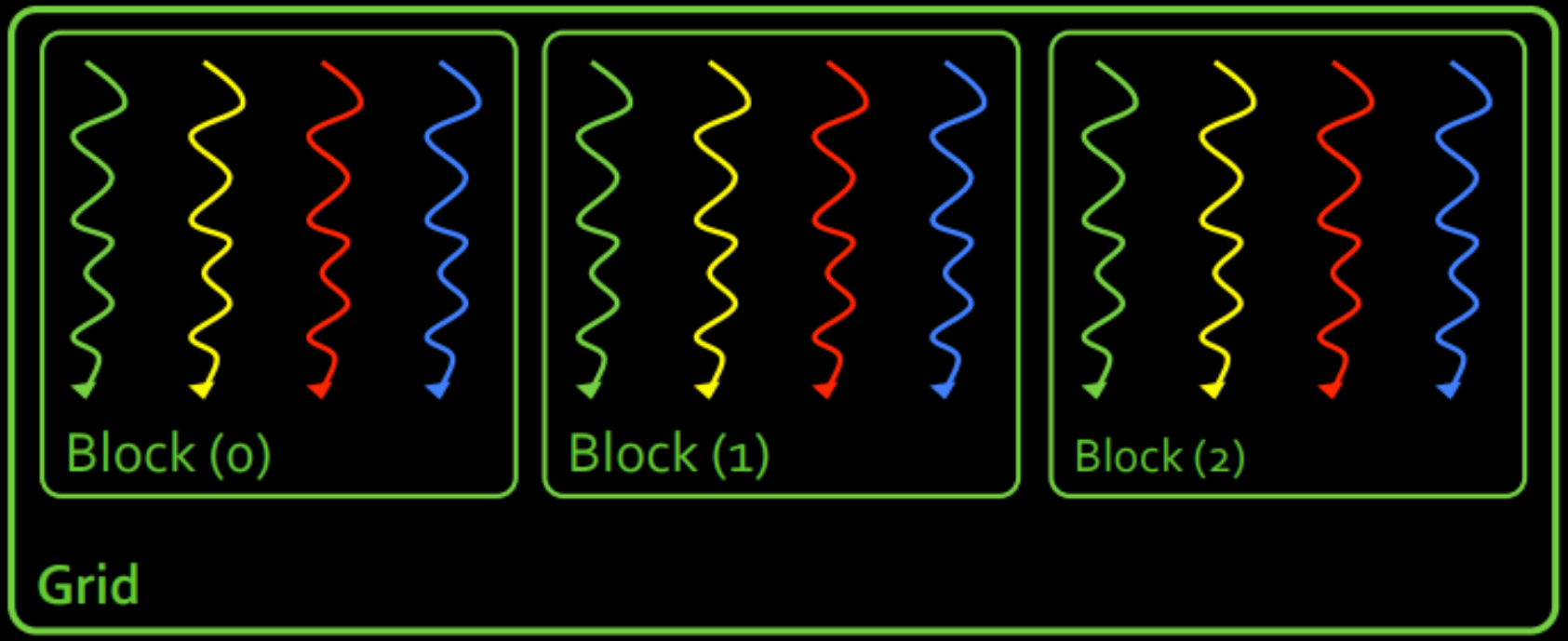
- GPUs can handle thousands of concurrent threads
- CUDA programming model supports even more
  - Allows a kernel launch to specify more threads than the GPU can execute concurrently
  - Helps to amortize kernel launch times

Blocks of threads



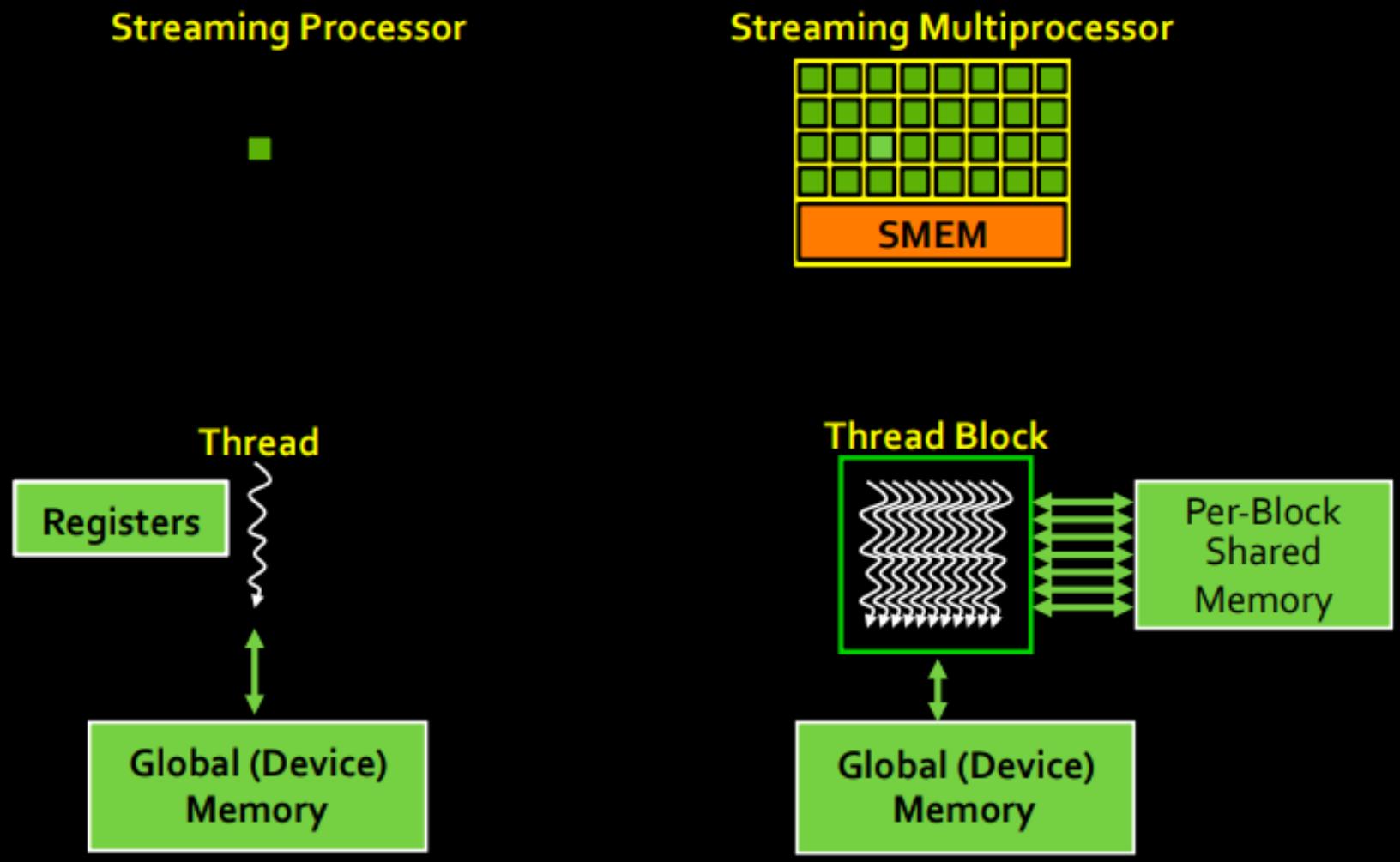
- Threads are grouped into blocks

Grids of blocks

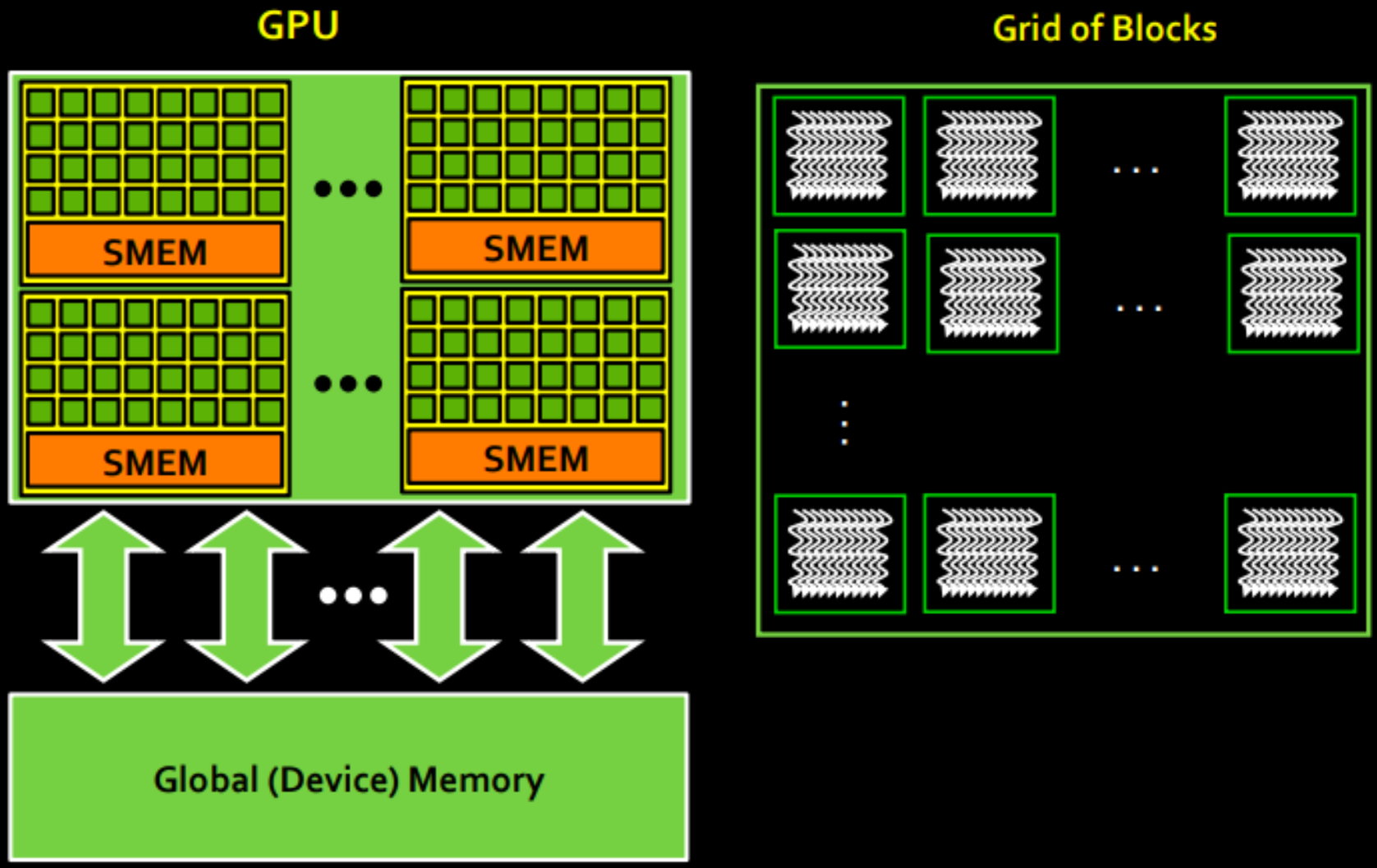


- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

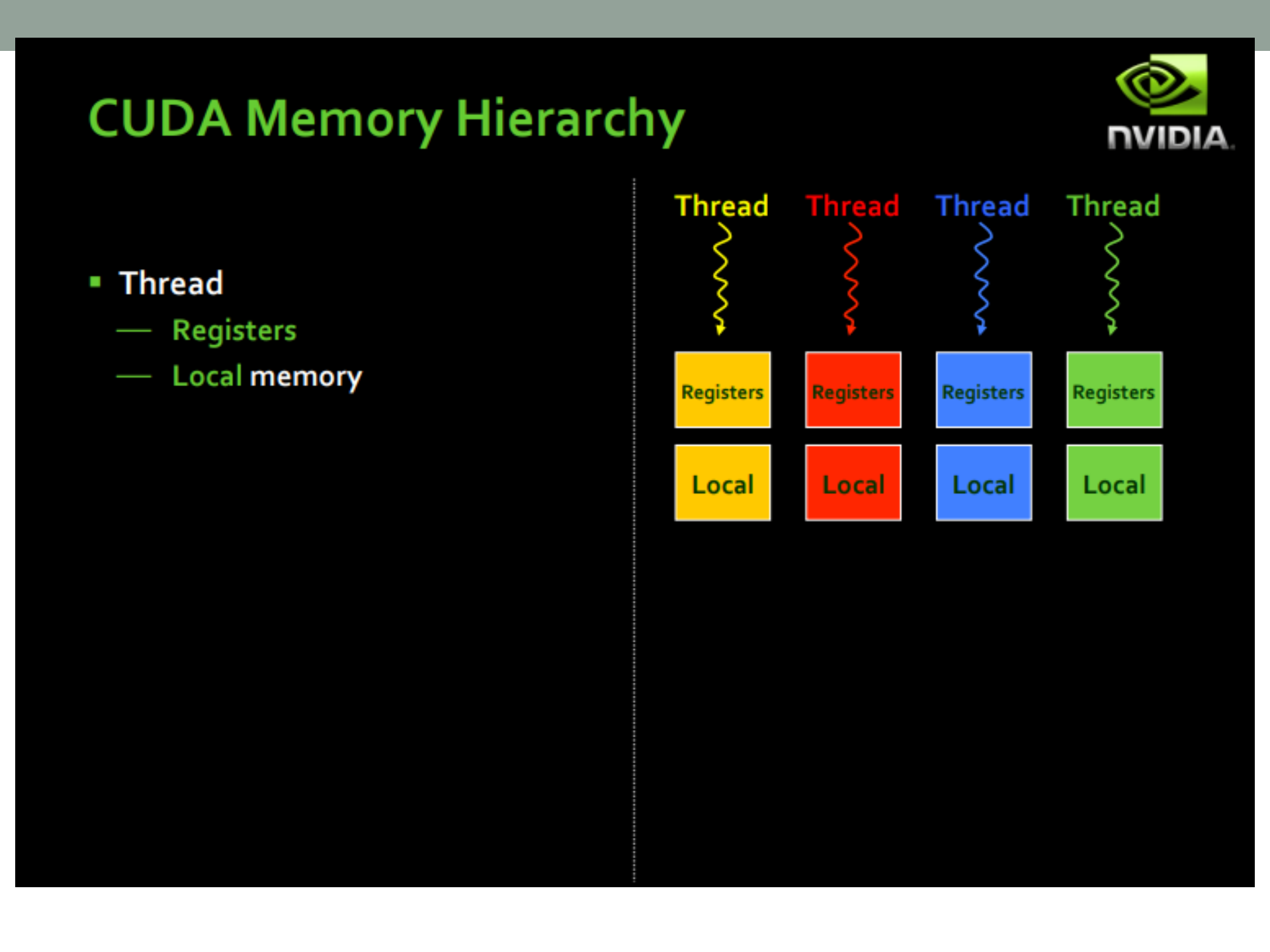
Blocks execute on Streaming Multiprocessors



Grids of blocks executes across GPU



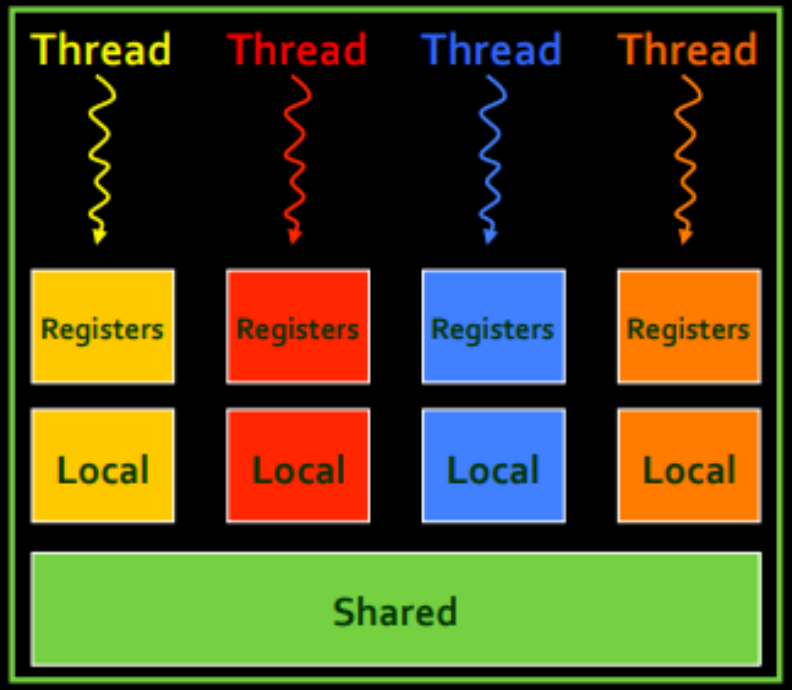
CUDA Memory Hierarchy



- Thread
  - Registers
  - Local memory

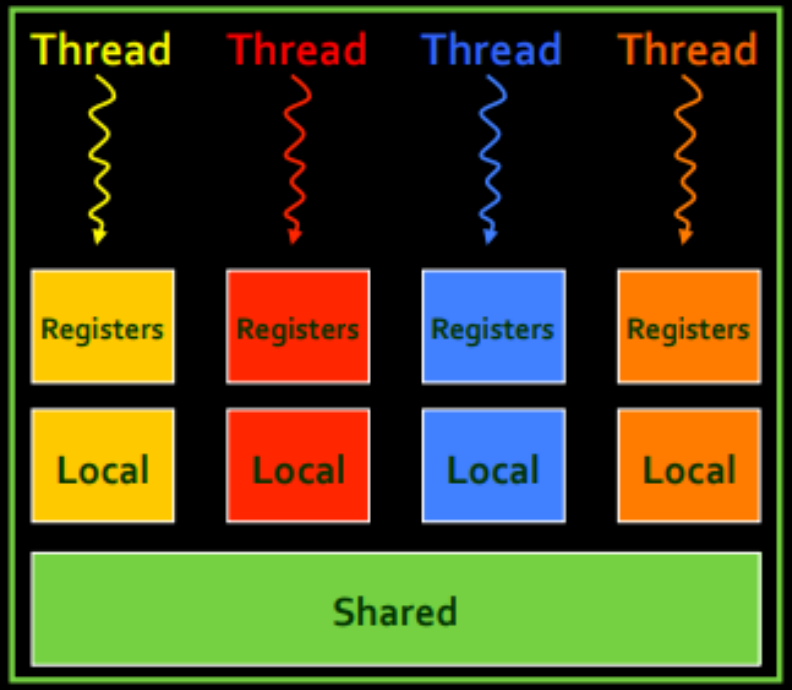
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory



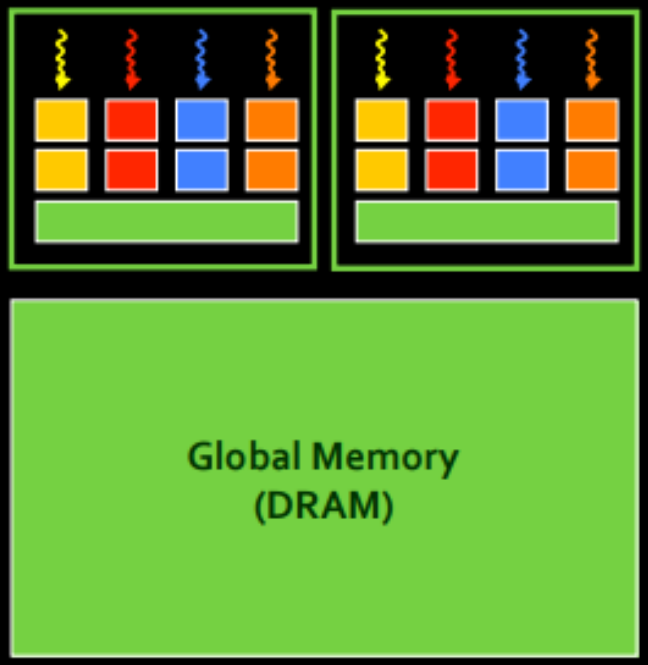
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory



CUDA Memory Hierarchy

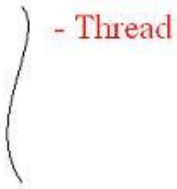
- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory
- All Thread Blocks
  - Global Memory



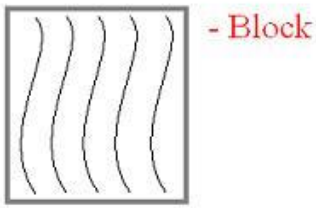
Thread Hierarchy

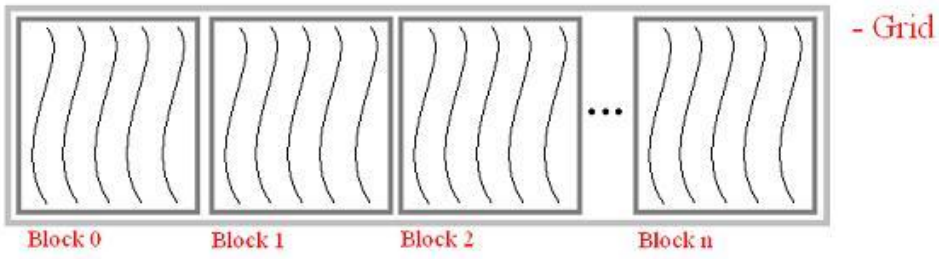
Thread – Distributed by the CUDA runtime
(identified by **threadIdx**)

Warp – A scheduling unit of up to 32 threads



Block – A user defined group of 1 to 512 threads.
(identified by **blockIdx**)

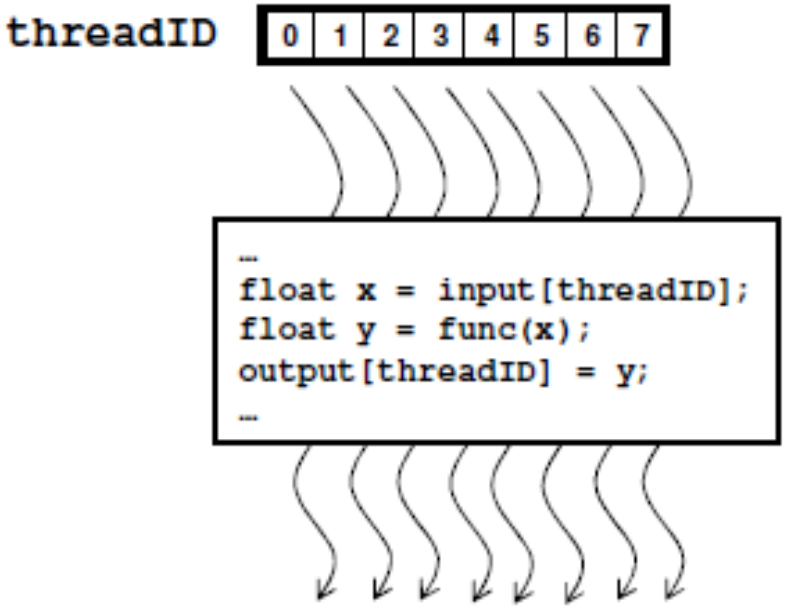




Grid – A group of one or more blocks. A grid is created for each CUDA kernel function

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



Cuda Programming

- Cuda Specifications
  - Function Qualifiers
  - CUDA Built-in Device Variables
  - Variable Qualifiers
- Cuda Programming and Examples
  - Compile procedure
  - Examples

Kernel Function Qualifiers

- __device__
- __global__
- __host__

Example in C:

CPU program

void increment_cpu(float *a, float b, int N)

CUDA program

__**global**__ void increment_gpu(float *a, float b, int N)

CPU program

```
void increment_cpu(float *a, float b, int N)
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
```

Function Qualifiers

- __global__ : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions,
  - cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host
- __host__ and __device__ qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

Variable Type Qualifiers

- Specify how a variable is stored in memory
- __device__
- __shared__
- __constant__

Example:

```
__global__ void increment_gpu(float *a, float b, int N)
{
  __shared__ float shared[];
}
```

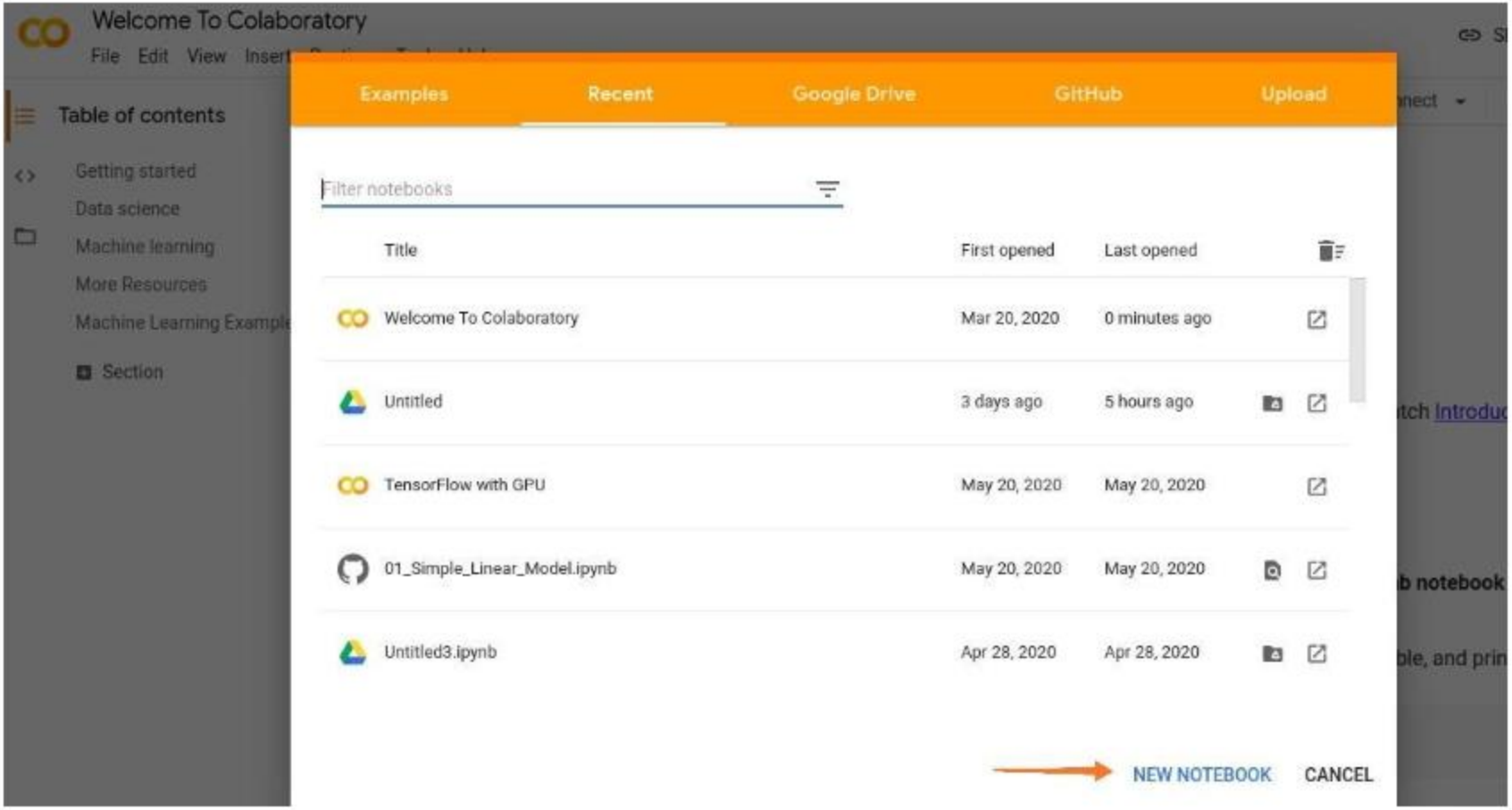
CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
- dim3 gridDim;
  - Dimensions of the grid in blocks (at most 2D)
- dim3 blockDim;
  - Dimensions of the block in threads
- dim3 blockIdx;
  - Block index within the grid
- dim3 threadIdx;
  - Thread index within the block

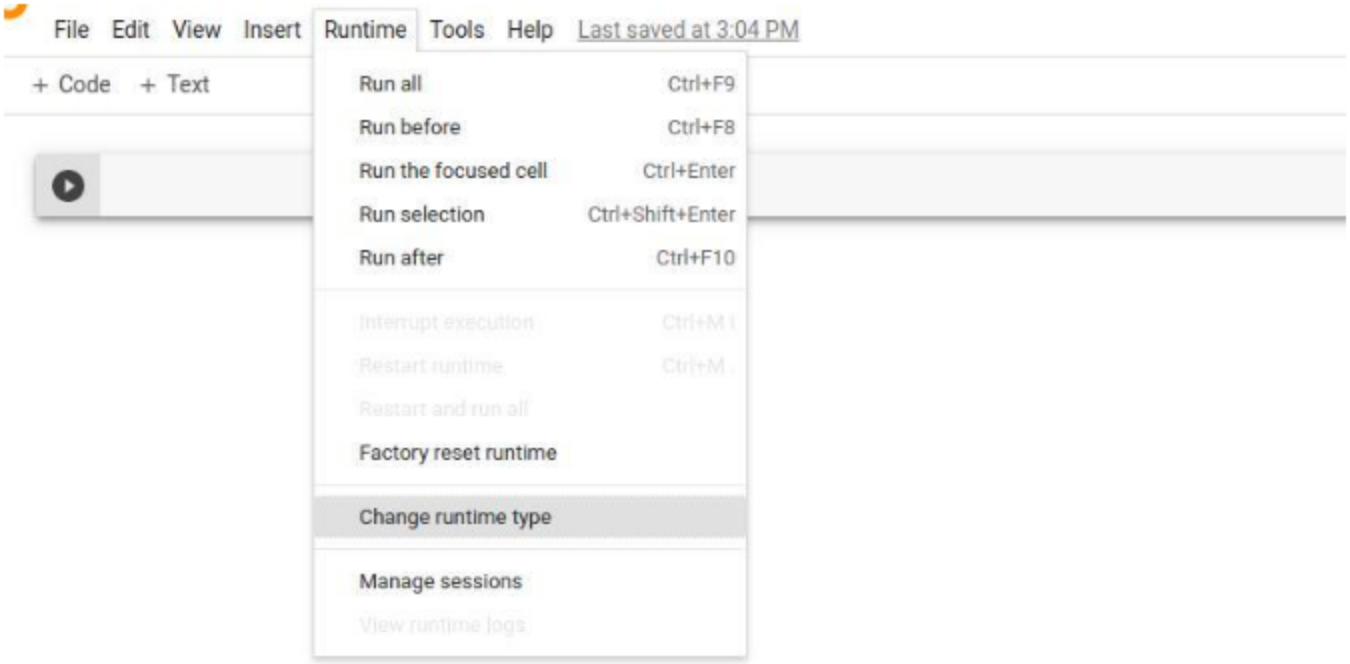
Variable Qualifiers (GPU code)

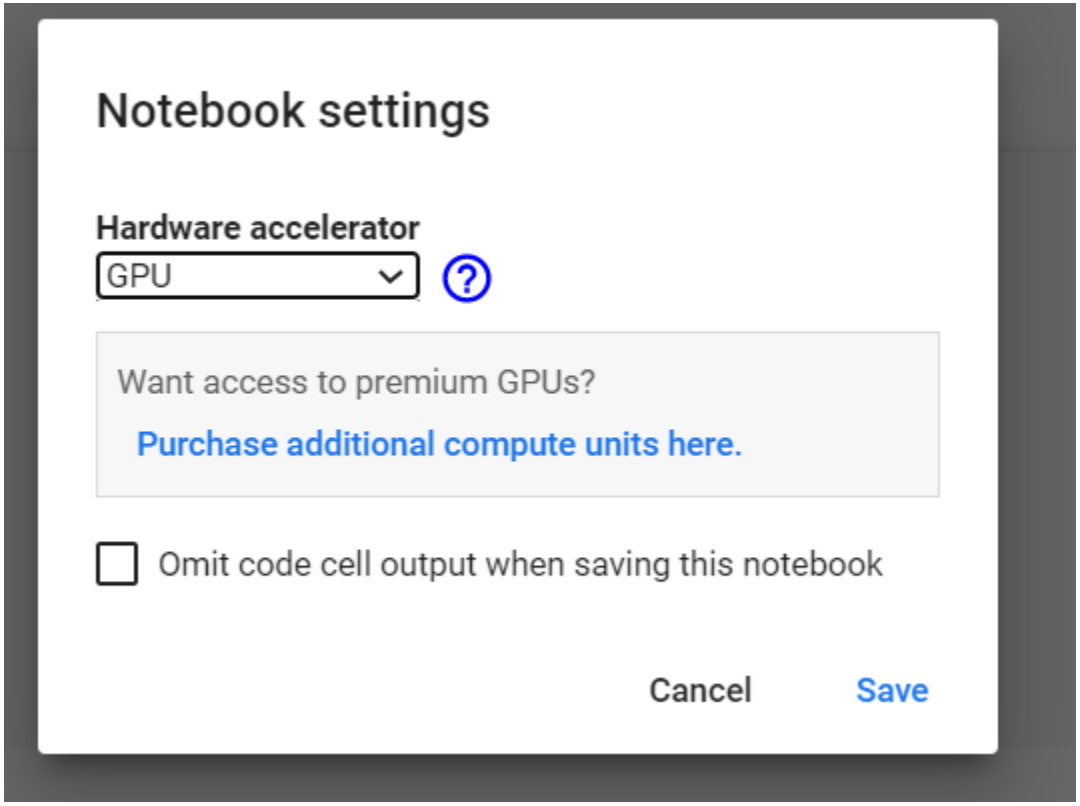
- __device__
  - Stored in device memory (large, high latency, no cache)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- __shared__
  - Stored in on-chip shared memory (very low latency)
  - Allocated by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: kernel execution
- Unqualified variables:
  - Scalars and built-in vector types are stored in registers
  - Arrays of more than 4 elements stored in device memory

Step 1: Go to https://colab.research.google.com in Browser and Click on New Notebook.



Step 2: We need to switch our runtime from CPU to GPU. Click on Runtime > Change runtime type > Hardware Accelerator > GPU > Save.




Notebook settings
Hardware accelerator
GPU
Want access to premium GPUs?
Purchase additional compute units here.
Omit code cell output when saving this notebook
Cancel
Save

- Every line that starts with ‘!’, it will be executed as a command line command.

```
!apt-get --purge remove cuda nvidia* libnvidia-*
!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --
purge
!apt-get remove cuda-*
!apt autoremove
!apt-get update
```

- **check your CUDA installation by running the command given below :**

```
!nvcc --version
```

Output will be something like this:

```
vcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Built on Wed_Apr_11_23:16:29_CDT_2018
Cuda compilation tools, release 9.2, V9.2.88
```

Step 6: Run the given command to install a small extension to run nvcc from the Notebook cells.

```
!pip install
git+https://github.com/andreinechaev/nvcc4jupyter.git
```

Step 7: Load the extension using the code given below:

```
%load_ext nvcc_plugin
```

- To run the code in your notebook, add the %%cu extension at the beginning of your code.

```
%%cu
#include <stdio.h>
__global__ void Hellokernel()
{

}
main()
{
Hellokernel<<<1, 1>>>();
 printf("Hello World\n");
return 0;
}
```

- A kernel is defined using the **__global__** declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new **<<<...>>>***execution configuration* syntax.
- Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through built-in variables.

Minimal Kernels

```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

Common Pattern!

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

int idx = blockDim.x * blockId.x + threadIdx.x;
will map from local index threadIdx to global index

NB: blockDim should be >= 32 in real code, this is just an example

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}

void main()
{
   .....
       increment_cpu(a, b, N);
}
```

CUDA program

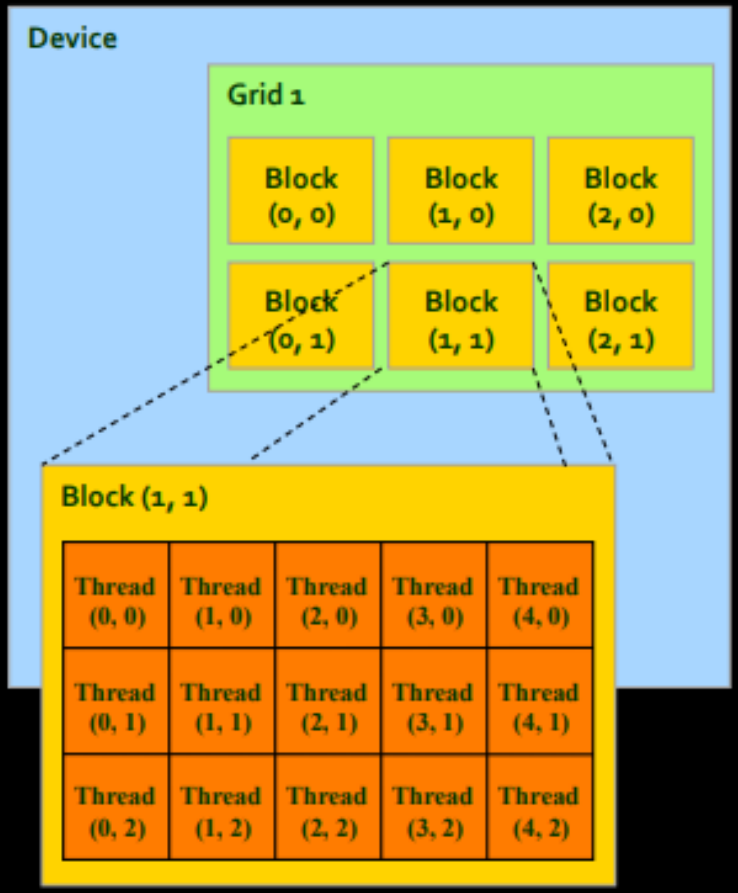
```
__global__ void increment_gpu(float *a, float b, int N)
{
       int idx = blockIdx.x * blockDim.x + threadIdx.x;
       if (idx < N)
           a[idx] = a[idx] + b;
}

void main()
{
    ...
       dim3 dimBlock (blocksize);
       dim3 dimGrid( ceil( N / (float)blocksize) );
       increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Drastic bandwidth reduction
- Thread cooperation is a powerful feature of CUDA

Thread and Block ID and Dimensions

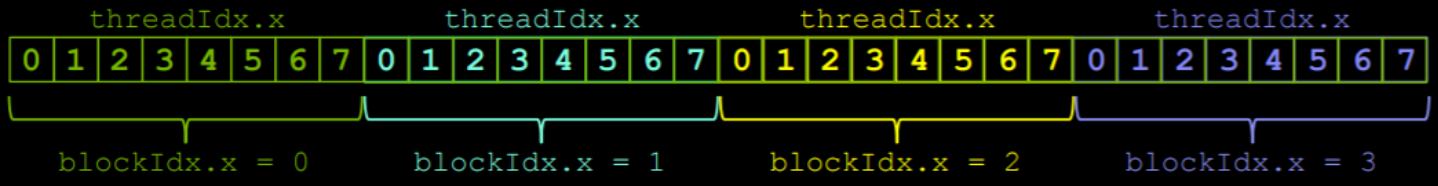
- Threads
  - 3D IDs, unique within a block
- Thread Blocks
  - 2D IDs, unique within a grid
- Dimensions set at launch
  - Can be unique for each grid
- Built-in variables
  - `threadIdx`, `blockIdx`
  - `blockDim`, `gridDim`
- Programmers usually select dimensions that simplify the mapping of the application data to CUDA threads



Indexing Arrays With Threads And Blocks

- No longer as simple as just using `threadIdx.x` or `blockIdx.x` as indices

- To index array with 1 thread per entry (using 8 threads/block)

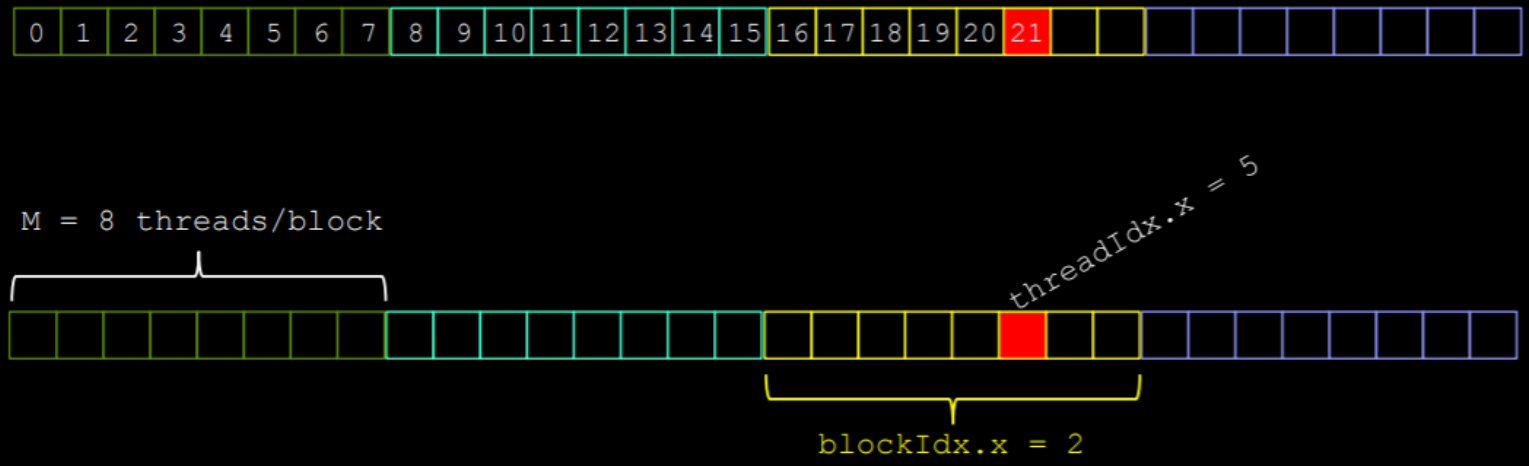


- If we have `M` threads/block, a unique array index for each entry given by

```
int index = threadIdx.x + blockIdx.x * M;

int index =       x     +     y      * width;
```

Indexing Arrays: Example

- In this example, the red entry would have an index of 21:




```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +      2      * 8;
          = 21;
```

CUDA C

```
#include <stdio.h>

__global__ void print_kernel() {
    printf("Hello from block %d, thread %d\n", blockIdx.x, threadIdx.x);
}

int main() {
    print_kernel<<<10, 10>>>();
    cudaDeviceSynchronize();
}
```

```
Hello from block 1, thread 0
Hello from block 1, thread 1
Hello from block 1, thread 2
Hello from block 1, thread 3
Hello from block 1, thread 4
Hello from block 1, thread 5
....
Hello from block 8, thread 3
Hello from block 8, thread 4
Hello from block 8, thread 5
Hello from block 8, thread 6
Hello from block 8, thread 7
Hello from block 8, thread 8
Hello from block 8, thread 9
```

Parallel Programming in CUDA C

- With `add()` running in parallel...let's do vector addition
- Terminology: Each parallel invocation of `add()` referred to as a *block*
- Kernel can refer to its block's index with the variable `blockIdx.x`
- Each block adds a value from `a[]` and `b[]`, storing the result in `c[]`:

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index arrays, each block handles different indices

Parallel Programming in CUDA C

- We write this code:

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- This is what runs in parallel on the device:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

Parallel Addition: `main()`

```
#define N  512
int main( void ) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;     // device copies of a, b, c
    int size = N * sizeof( int );   // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

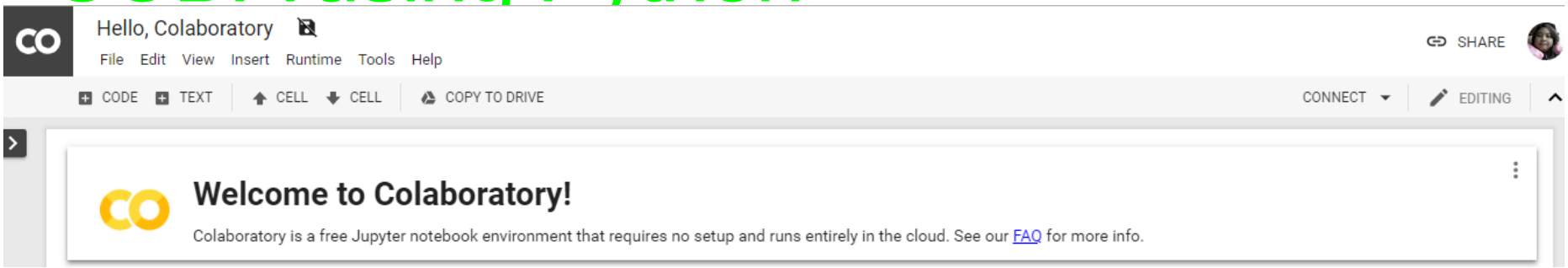
Parallel Addition: `main()` (cont)

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with N parallel blocks
    add<<< N, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

CUDA using Python

- Anaconda/Python 3.6.1/Jupyter notebook
- CUDA Toolkit
- Numba package

CUDA using Python




```
!nvidia-smi
```

```
Tue Oct 30 12:11:30 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.44                 Driver Version: 396.44                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   72C    P8    33W / 149W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

CUDA using Python

```
!apt-get install nvidia-cuda-toolkit
```

```
  libvdpau-va-gl1 nvidia-vdpau-driver nvidia-legacy-340xx-vdpau-driver
  mesa-utils
Recommended packages:
  libnvcuvid1
The following NEW packages will be installed:
  adwaita-icon-theme at-spi2-core ca-certificates-java cpp-6
  dconf-gsettings-backend dconf-service fontconfig fonts-dejavu-core
```

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Built on Tue_Jun_12_23:07:04_CDT_2018
Cuda compilation tools, release 9.2, V9.2.148
```

```
!pip3 install numba
```

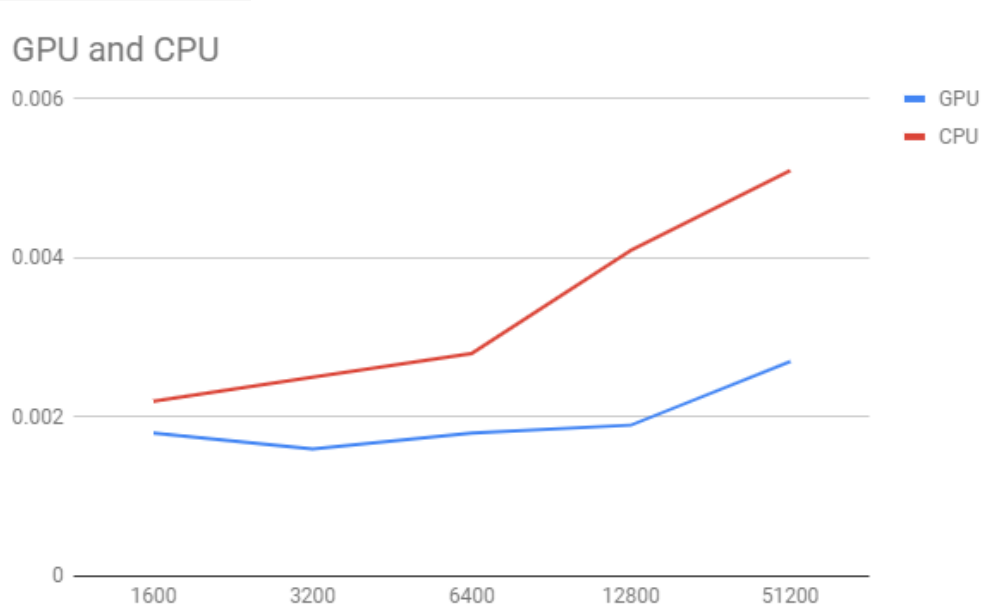
```
Collecting numba
  Downloading https://files.pythonhosted.org/packages/42/45/8d5fc45e5f760ac65906ba48dec98e99e7920c96783ac7248c5e31c9464e/numba-0.40.1-cp36-cp36m-manylinux1_x8
    100% |████████████████████████████████| 3.2MB 8.3MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from numba) (1.14.6)
Collecting llvmlite>=0.25.0dev0 (from numba)
  Downloading https://files.pythonhosted.org/packages/34/fb/f9c2e9e0ef2b54c52f0b727cf6af75b68c3d7ddb6d88c8d557b1b16bc1ab/llvmlite-0.25.0-cp36-cp36m-manylinux1
    100% |████████████████████████████████| 16.1MB 2.8MB/s
Installing collected packages: llvmlite, numba
Successfully installed llvmlite-0.25.0 numba-0.40.1
```

Vector add GPU

```
from __future__ import print_function
from timeit import default_timer as time
import numpy as np
from numba import cuda

@cuda.jit('(f4[:], f4[:], f4[:])')
def cuda_sum(a, b, c):
    i = cuda.grid(1)
    c[i] = a[i] + b[i]

griddim = 50, 1
blockdim = 32, 1, 1
N = griddim[0] * blockdim[0]
print("N", N)
cuda_sum_configured = cuda_sum.configure(griddim, blockdim)
a = np.array(np.random.random(N), dtype=np.float32)
b = np.array(np.random.random(N), dtype=np.float32)
c = np.empty_like(a)

ts = time()
cuda_sum_configured(a, b, c)
te = time()
print(te - ts)


assert (a + b == c).all()
print(c)
```

Vector add CPU

```
from timeit import default_timer as time
import numpy as np
N = 1600
def cpu_sum(a, b, c):
    for i in range(0, N):
      c[i] = a[i] + b[i]


a = np.array(np.random.random(N), dtype=np.float32)
b = np.array(np.random.random(N), dtype=np.float32)
c = np.empty_like(a)

ts = time()
cpu_sum(a, b, c)
te = time()
print(te - ts)

print(c)
```



LAB 14 CUDA

A First Program

```
/* Cuda Hello, World, hello.cu”
#include <stdio.h>

__global__ void mykernel(void) {
}

int main(void){
   mykernel<<<1,1>>>();
   printf(“Hello, World\n”);
   return 0;
}
```

__global__

__global__ :

1. A qualifier added to standard C. This alerts the compiler that a function should be compiled to run on a device (GPU) instead of host (CPU).
2. Function mykernel() is called from host code.

Addition

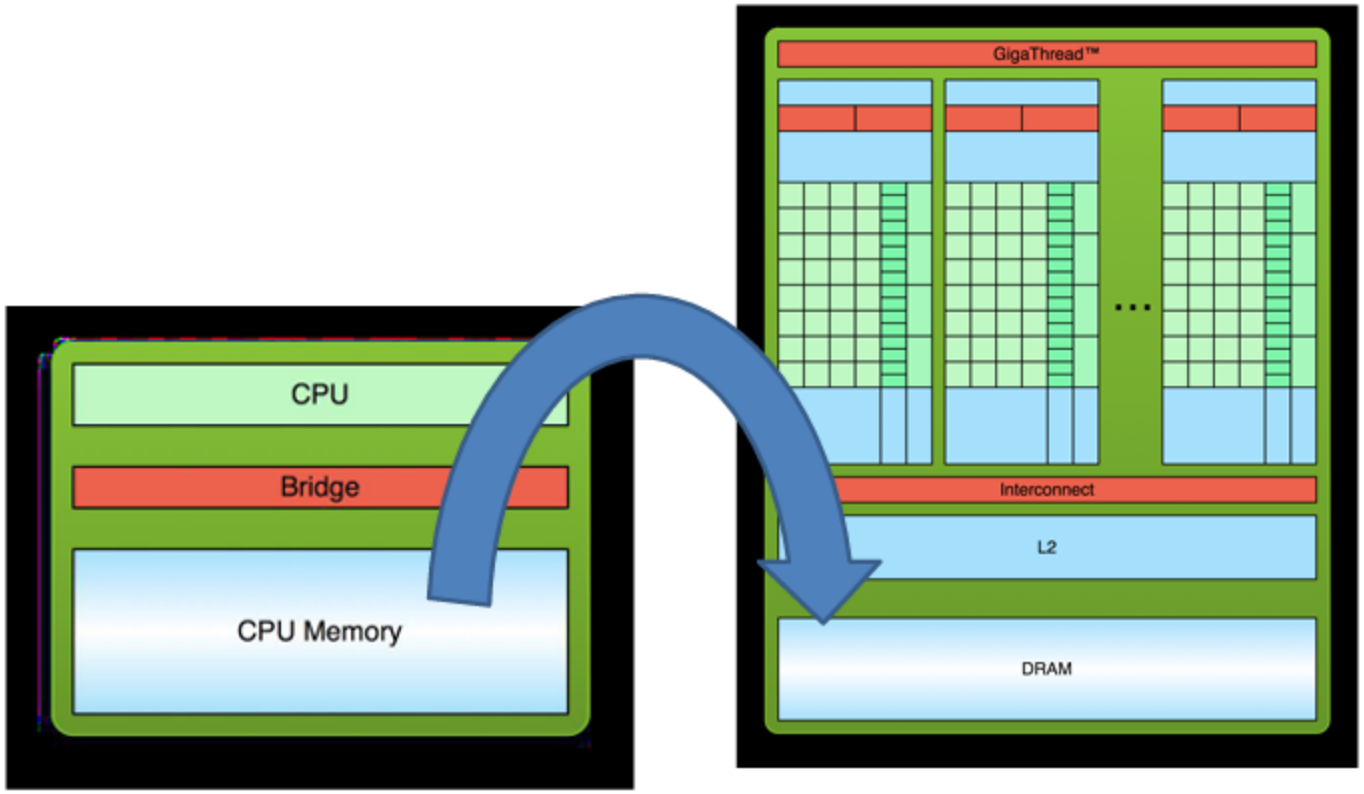
```
%%cu
#include <stdio.h>
__global__ void add(int a, int b, int *c)
{
*c = a + b;
}
```

```
int main(void)
{
int c;                          // host copies of c variable
int *dev_c;                     // device copies of c variable
cudaMalloc((void**)&dev_c, sizeof(int));
add << <1, 1 >> > (2, 7, dev_c);
cudaMemcpy(&c, dev_c, sizeof(int),
cudaMemcpyDeviceToHost);
printf("2 + 7 = %d\n", c);
cudaFree(dev_c);
return 0;
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

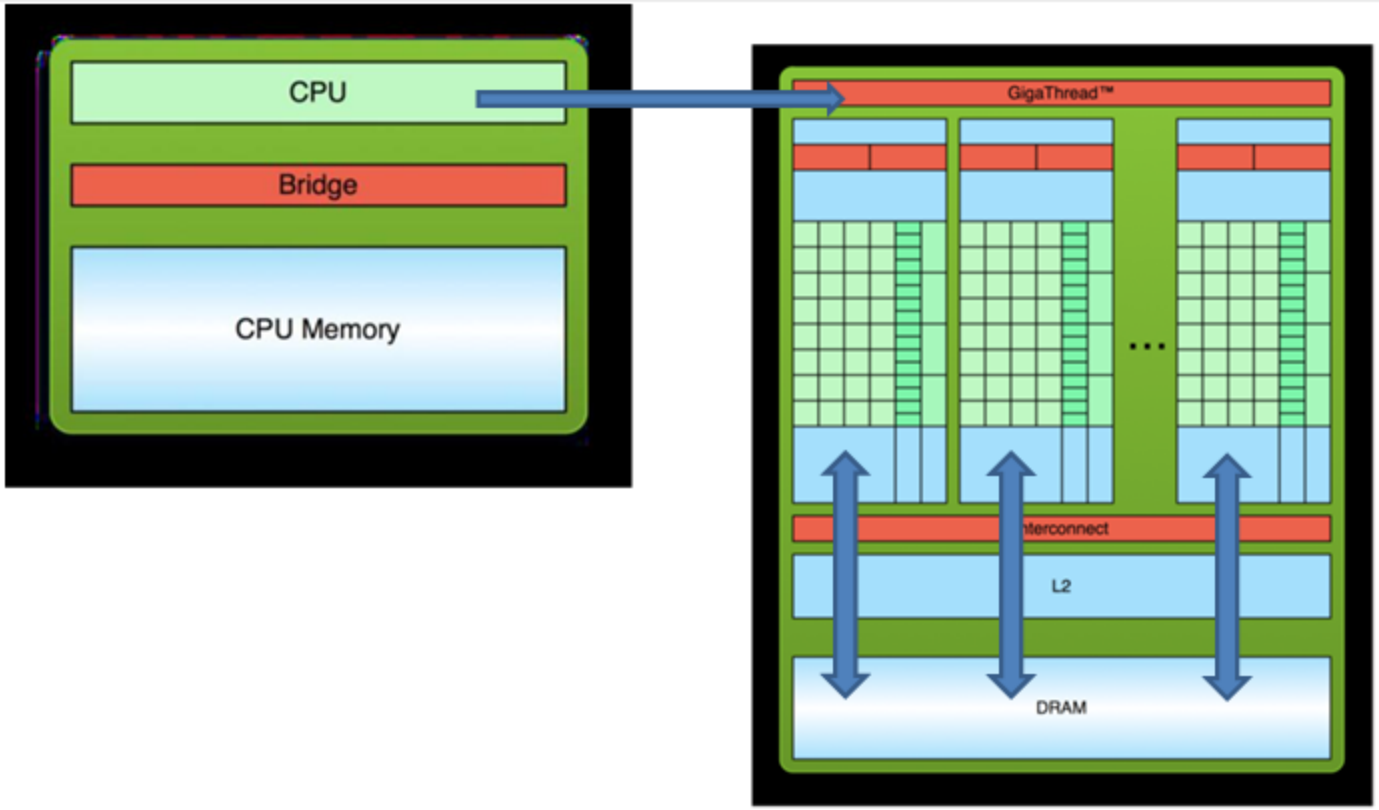
- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

Processing Flow



1. Copy input data from CPU memory to GPU memory and allocate memory

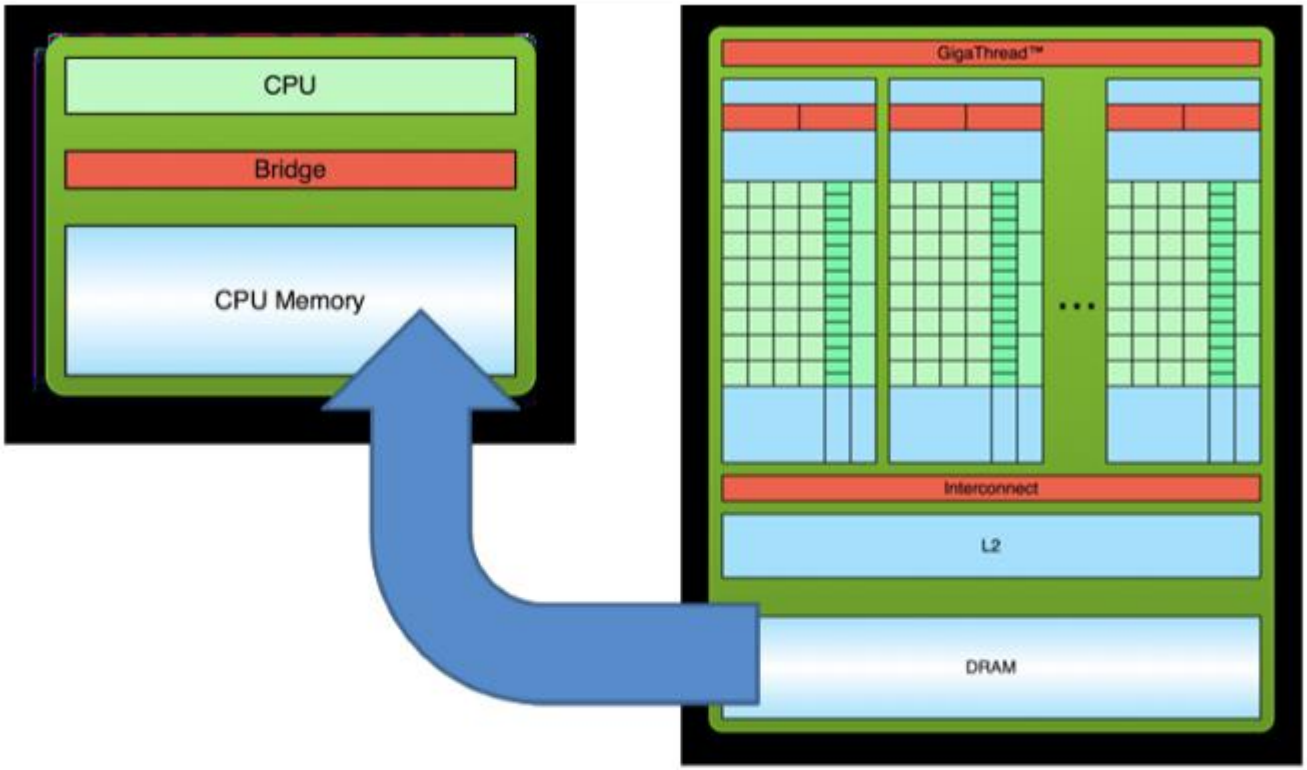
```
// cudaMalloc((void**)&device_c, sizeof(int));
```



2. Load GPU program and execute,

Caching data on chip for performance

//add<<<1, 1>>>(2, 7, device_c);



3. Copy results from GPU memory to CPU memory

//cudaMemcpy(&c, device_c, sizeof(int), cudaMemcpyDeviceToHost);

Passing Parameters & Data Transfer

```
// File name: add.cu
#include <stdio.h>

__global__ void add(int a, int b, int *c){
   *c = a+b;
}

int main(void){
   int c;
   int  *device_c;

   cudaMalloc((void**)&device_c, sizeof(int));
   add<<<1, 1>>>(2, 7, device_c);
   cudaMemcpy(&c, device_c, sizeof(int), cudaMemcpyDeviceToHost);

   printf("2+7 = %d\n", c);
   cudeFree(device_c);
   return 0;
}
```

- Can pass parameters to a kernel as with C function
- Need to allocate memory to do anything useful on a device, such as return values to the host.

- add() runs on the device, so device_*c* must point to the device memory
  - This is why we call cudaMalloc() to allocate memory on the device
- Do not deference the pointer returned by cudaMalloc() from code that executes on the host. Host code may pass this pointer around, perform arithmetic on it. But we can not use it to read or write from memory.
  - Its C equivalent is malloc().
- We can access memory on a device through calls to cudaMemcpy() from host code.
  - Its C equivalent is memcpy().

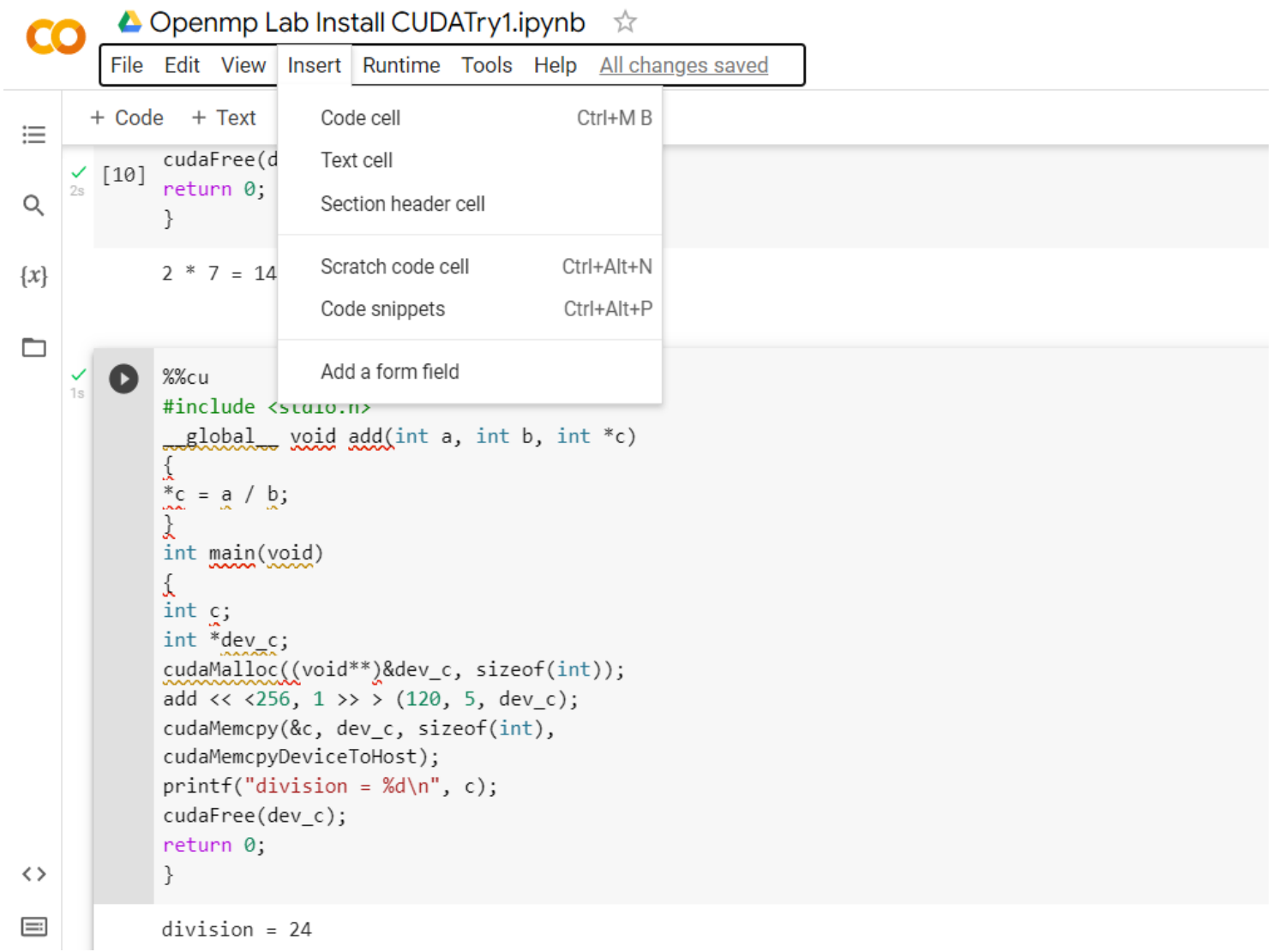
Parallel Computing

- How do we run code in parallel on the device?

add<<< 256, 1>>>(2, 7, device_c);

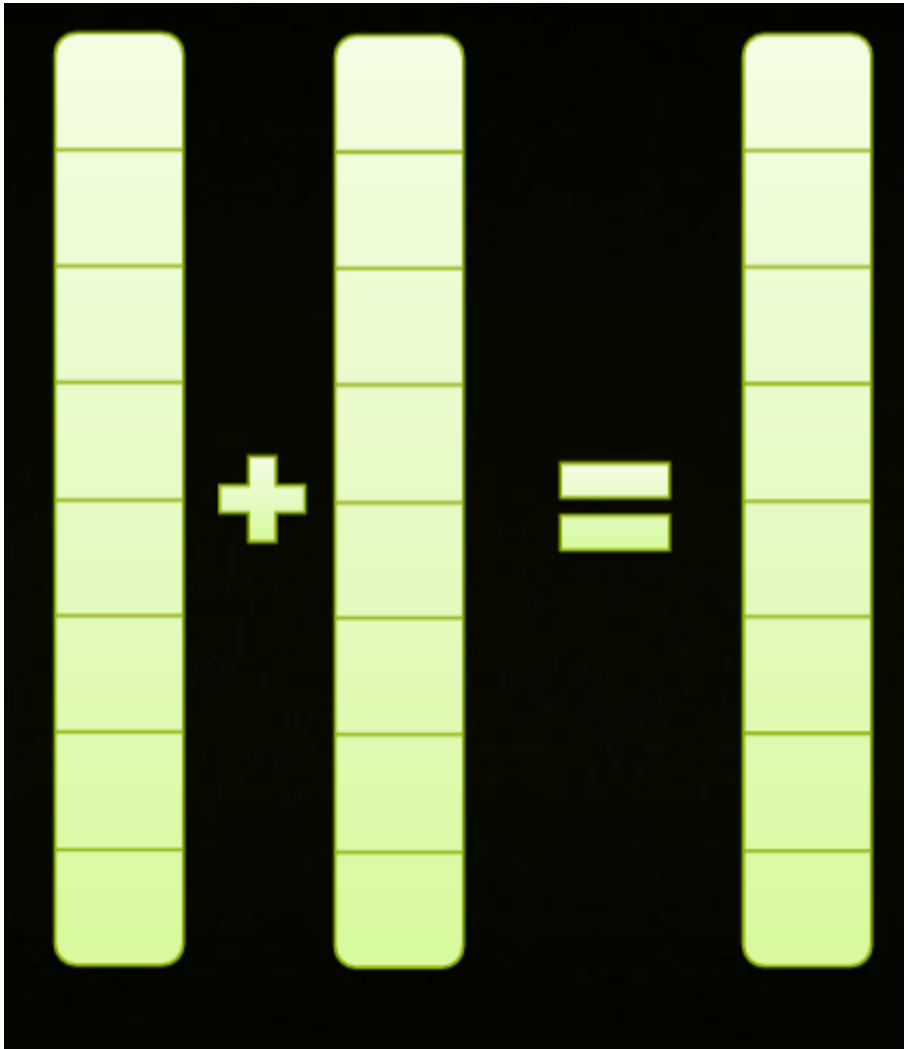
  - Instead of executing add() once, execute 256 times in parallel
- <<<N,1>>>();
  - The number “N” represents the number of parallel blocks (of threads) in which we would like the GPU to execute our kernel.
  - add<<< 256, 1>>>() can be thought as that the runtime creates 256 copies of the kernel and runs them in parallel.

Insert→ Code CELL



Arithmetic Operations

vector addition



- **blockIdx.x**, **blockIdx.y**, **blockIdx.z** are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.
- **threadIdx.x**, **threadIdx.y**, **threadIdx.z** are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.
- **blockDim.x**, **blockDim.y**, **blockDim.z** are built-in variables that return the "block dimension" (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis).

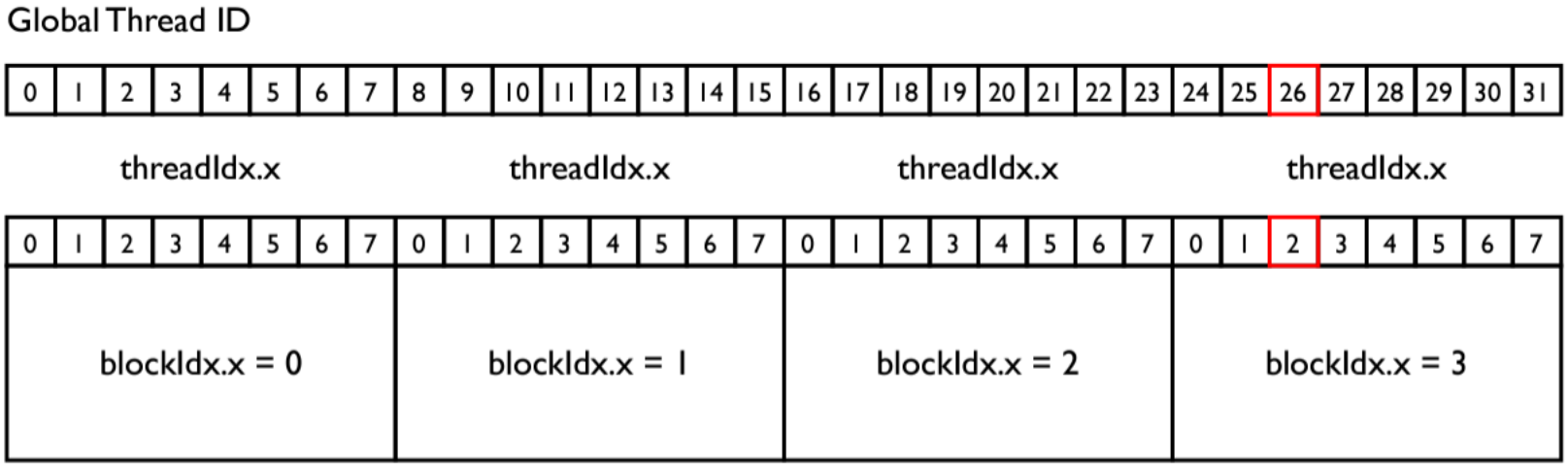
So, you can express your collection of blocks, and your collection of threads within a block, as a 1D array, a 2D array or a 3D array.

These can be helpful when thinking of your data as 2D or 3D.

The full global thread ID in x dimension can be computed by:

- x = blockIdx.x * blockDim.x + threadIdx.x;

Thread Identification Example: x-direction



- Assume a hypothetical 1D grid and 1D block architecture: 4 blocks, each with 8 threads.

- For Global Thread ID 26:
  - gridDim.x = 4 x 1
  - blockDim.x = 8 x 1
  - Global Thread ID = blockIdx.x * blockDim.x + threadIdx.x
  - = 3 x 8 + 2 = 26

- uint3 and dim3 are CUDA-defined structures of unsigned integers: x, y, and z.
  - struct uint3 {x; y; z;};
  - struct dim3 {x; y; z;};
- The unsigned structure components are automatically initialized to 1.
- These vector types are mostly used to define grid of blocks and threads.

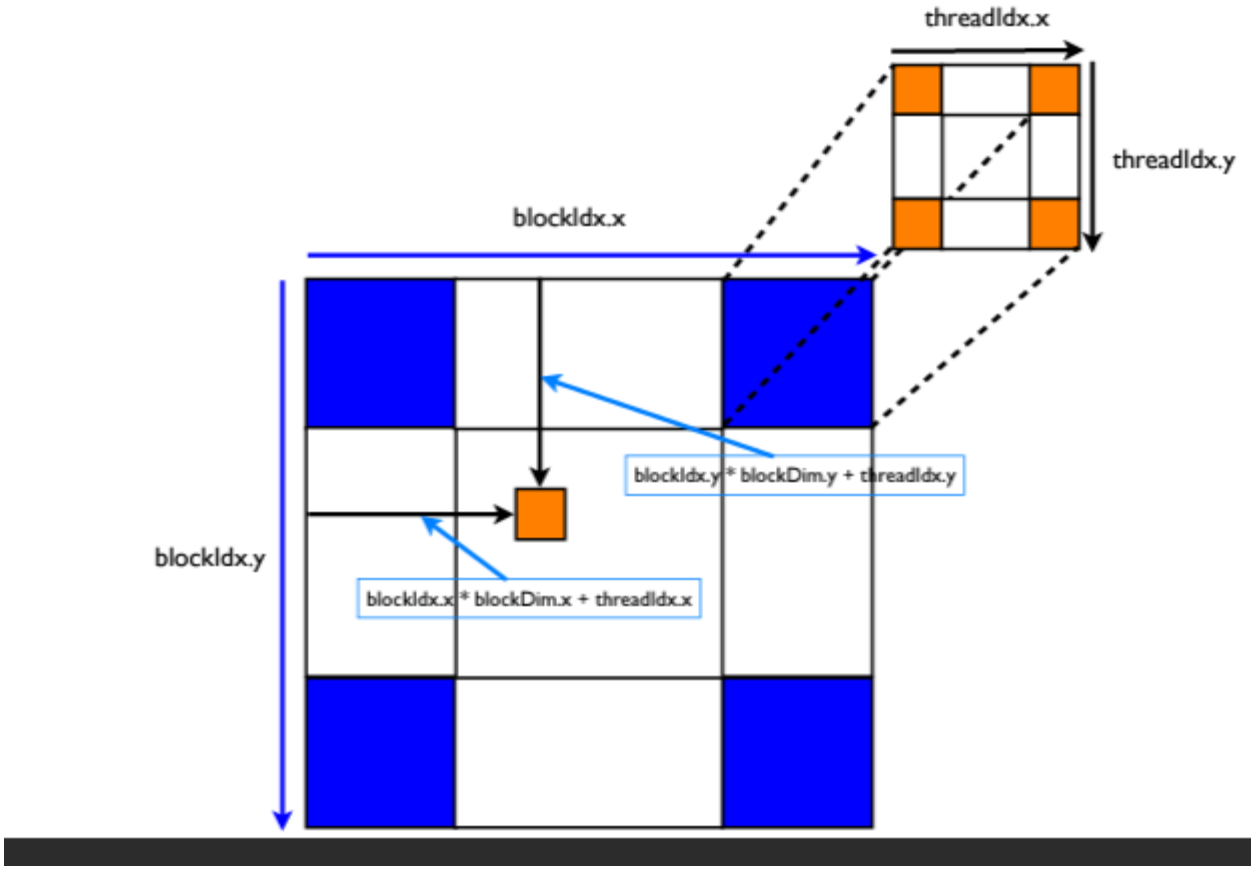
- To set dimensions:

```
dim3 grid(16,16);        // grid = 16 x 16 blocks
dim3 block(32,32);       // block = 32 x 32 threads
myKernel<<<grid, block>>>(...);
```

- which sets:

```
grid.x = 16;
grid.y = 16;
block.x = 32;
block.y = 32;
block.z = 1;
```

2D Grids and 2D Blocks



```
%%cu
#include <stdio.h>
__global__ void vector_add(int *out_d, int *a, int *b, int n)
  {
  int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
int row = by*blockDim.y + ty;
int col = bx*blockDim.x + tx;
int dim = gridDim.x*blockDim.x;
int i = row*dim + col;
out_d[i] = a[i] + b[i];

   }
```

```
int main()
  {
  int *a, *b, *out_d,*out;
   int *d_a, *d_b;
   int N=6;
  int i;
   a = (int*)malloc(sizeof(int) * N);
   b = (int*)malloc(sizeof(int) * N);
      out = (int*)malloc(sizeof(int) * N);
  for (i=0;i<N;i++)
    {
    a[i]=i;
    b[i]=i*2;
    }
```

```
cudaMalloc((void**)&d_a, sizeof(int) * N);
 cudaMalloc((void**)&d_b, sizeof(int) * N);
 cudaMalloc((void**)&out_d, sizeof(int) * N);
 cudaMemcpy(d_a, a, sizeof(int) * N, cudaMemcpyHostToDevice);
 cudaMemcpy(d_b, b, sizeof(int) * N, cudaMemcpyHostToDevice);
 vector_add<<<2,4>>>(out_d, d_a, d_b, N);
   cudaMemcpy(out, out_d, sizeof(int) * N, cudaMemcpyDeviceToHost);
 printf("Success");
   for (i=0;i<N;i++)
   {
   printf("%d\n",out[i]);
   }
 cudaFree(d_a);
 cudaFree(d_b);
   cudaFree(out_d);
 free(a);
 free(b);
   free(out);
 return 0;
 }
```

- Assume a hypothetical 1D grid and 1D block architecture: 4 blocks, each with 8 threads.

- For Global Thread ID 26:
  - gridDim.x = 4 x 1
  - blockDim.x = 8 x 1
  - Global Thread ID = blockIdx.x * blockDim.x + threadIdx.x
  - = 3 x 8 + 2 = 26

- uint3 and dim3 are CUDA-defined structures of unsigned integers: x, y, and z.
  - struct uint3 {x; y; z;};
  - struct dim3 {x; y; z;};
- The unsigned structure components are automatically initialized to 1.
- These vector types are mostly used to define grid of blocks and threads.

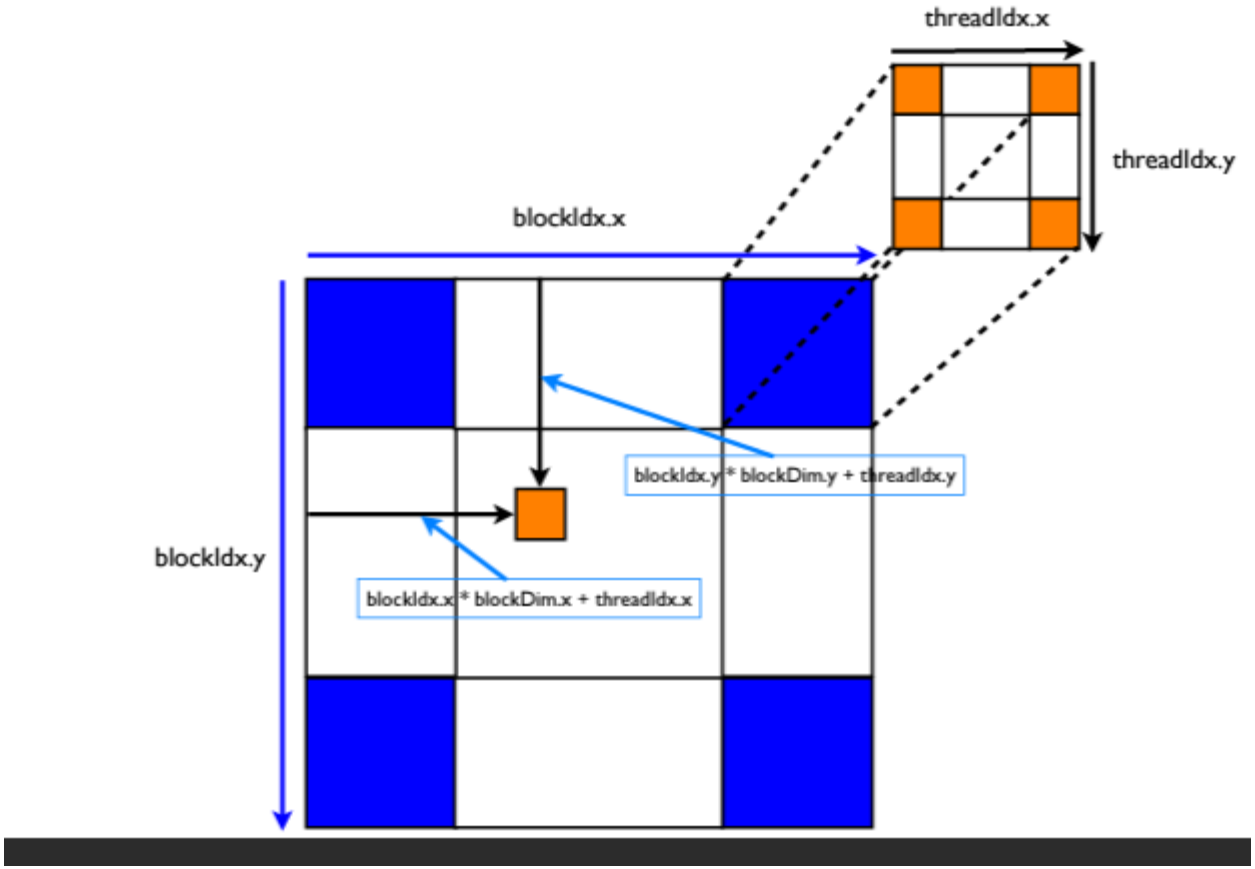
- To set dimensions:

```
dim3 grid(16,16);        // grid = 16 x 16 blocks
dim3 block(32,32);       // block = 32 x 32 threads
myKernel<<<grid, block>>>(...);
```

- which sets:

```
grid.x = 16;
grid.y = 16;
block.x = 32;
block.y = 32;
block.z = 1;
```

2D Grids and 2D Blocks



Parallel Programming in CUDA C

- With `add()` running in parallel...let's do vector addition
- Terminology: Each parallel invocation of `add()` referred to as a *block*
- Kernel can refer to its block's index with the variable `blockIdx.x`
- Each block adds a value from `a[]` and `b[]`, storing the result in `c[]`:

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index arrays, each block handles different indices

Parallel Programming in CUDA C

- We write this code:

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- This is what runs in parallel on the device:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

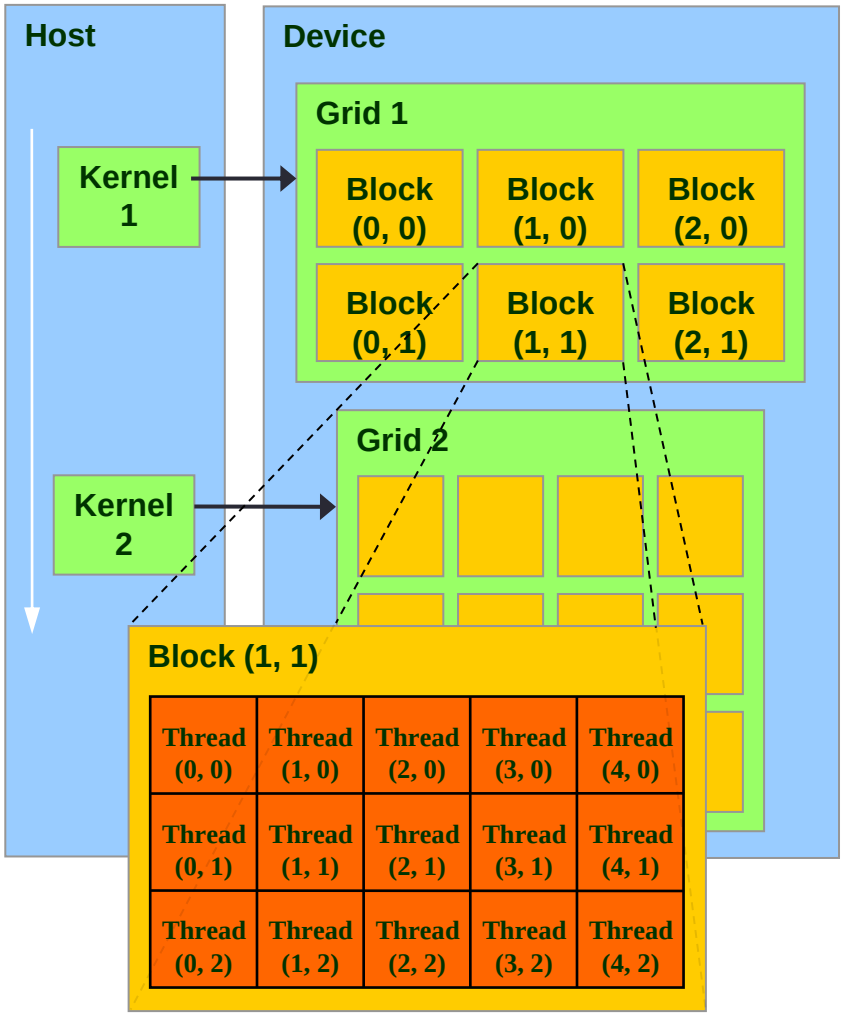
```
c[3] = a[3] + b[3];
```

CUDA Programming Model

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
    - Hardware switching between threads (in 1 cycle) on long-latency memory reference
    - Overprovision (1000s of threads) → hide latencies
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

- Kernel executed as a grid of thread blocks
  - All threads share data memory space
- Thread block is a batch of threads, can cooperate with each other by:
  - Synchronizing their execution: For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
  - (Unless thru slow global memory)
- Threads and blocks have IDs



Courtesy: NDVIA

CUDA Function Declarations

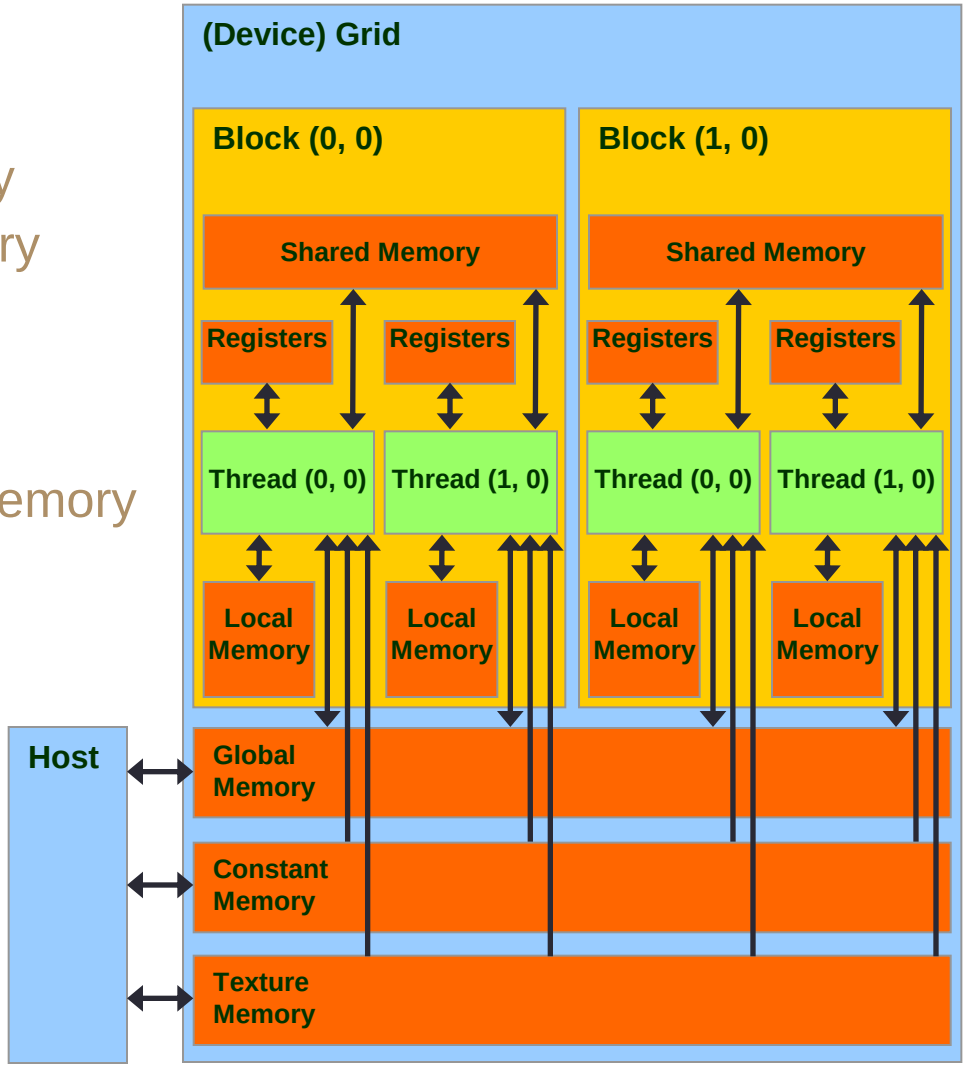
	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	Host
`__host__ float HostFunc()`	host	Host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

CUDA Device Memory Space Overview

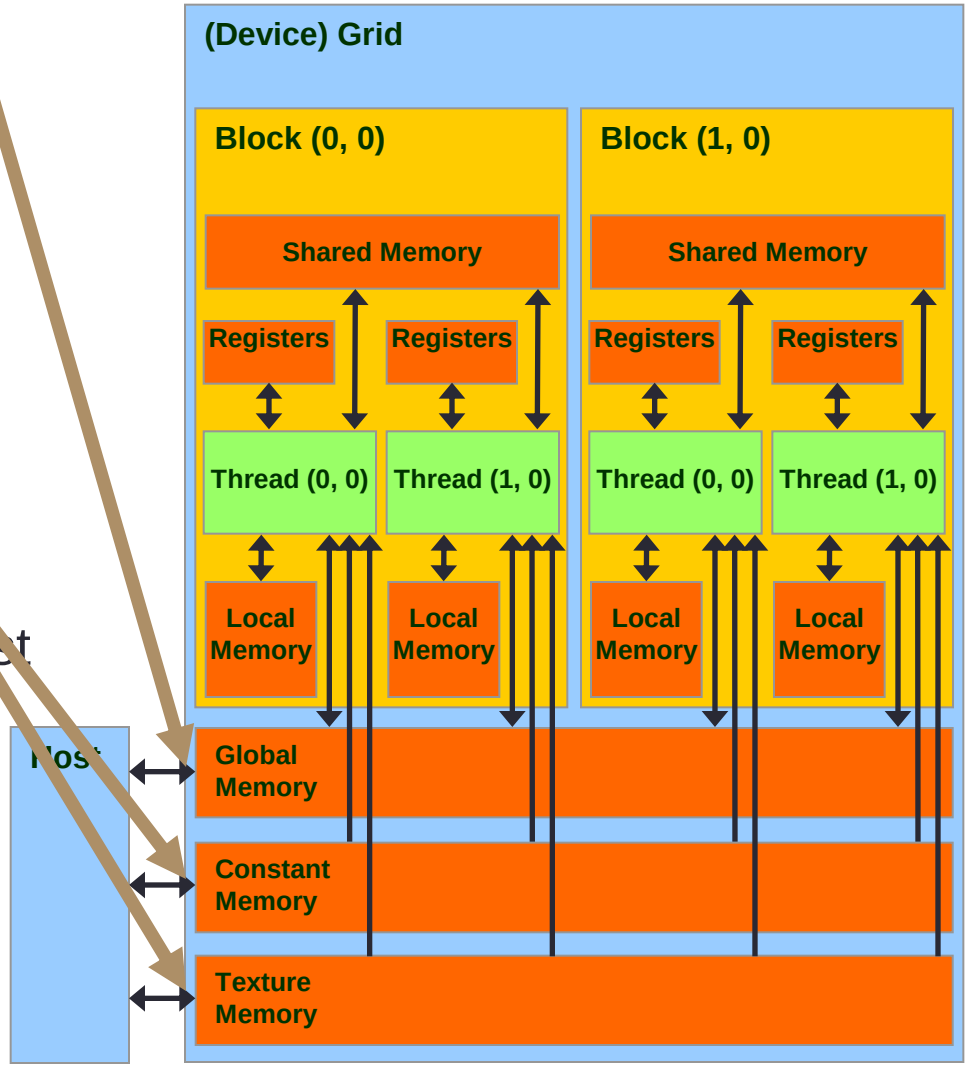
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



Courtesy: NDVIA

Methods for operations on Device Memory

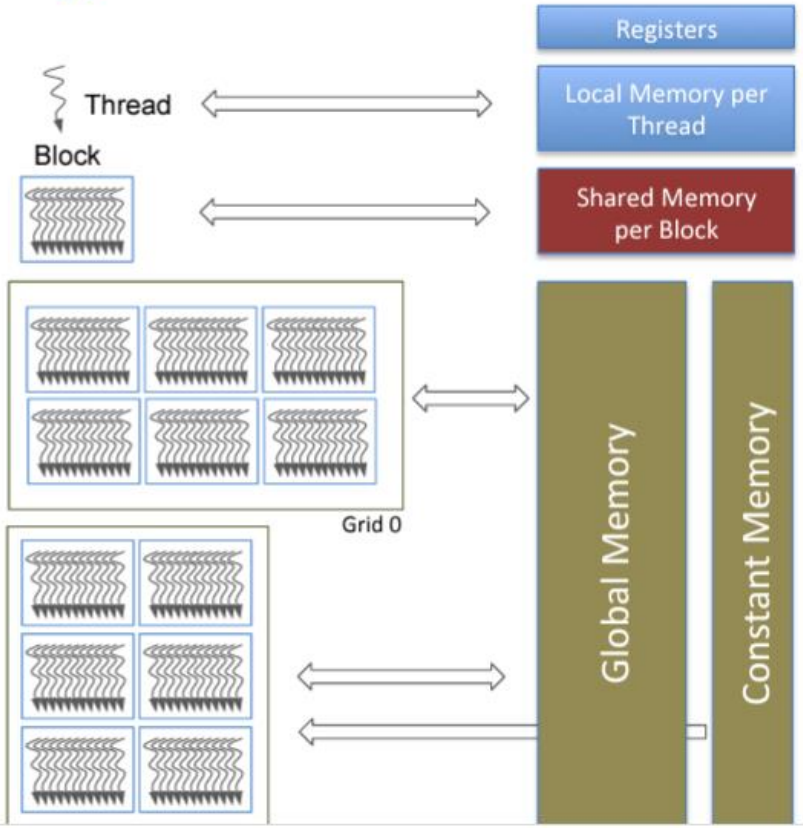
Memory Hierarchy

Private memory
Visible only to the thread

Shared memory
Visible to all the threads in a block

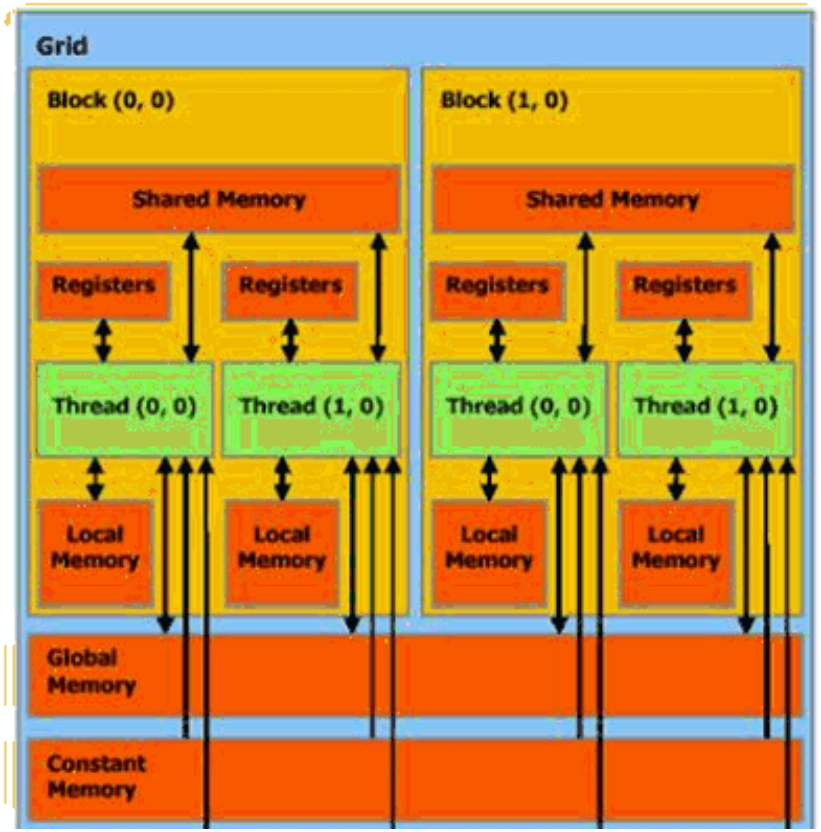
Global memory
Visible to all the threads
Visible to host
Accessible to multiple kernels
Data is stored in row major order

Constant memory (Read Only)
Visible to all the threads in a block



Memory Hierarchy

- Shared memory much much faster than global
- Don’t trust local memory
- Global, Constant memory available to both host and cpu



Thread Organization

Minimal Kernels

```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

Common Pattern!

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

int idx = blockDim.x * blockId.x + threadIdx.x;
will map from local index threadIdx to global index

NB: blockDim should be >= 32 in real code, this is just an example

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}

void main()
{
   .....
       increment_cpu(a, b, N);
}
```

CUDA program

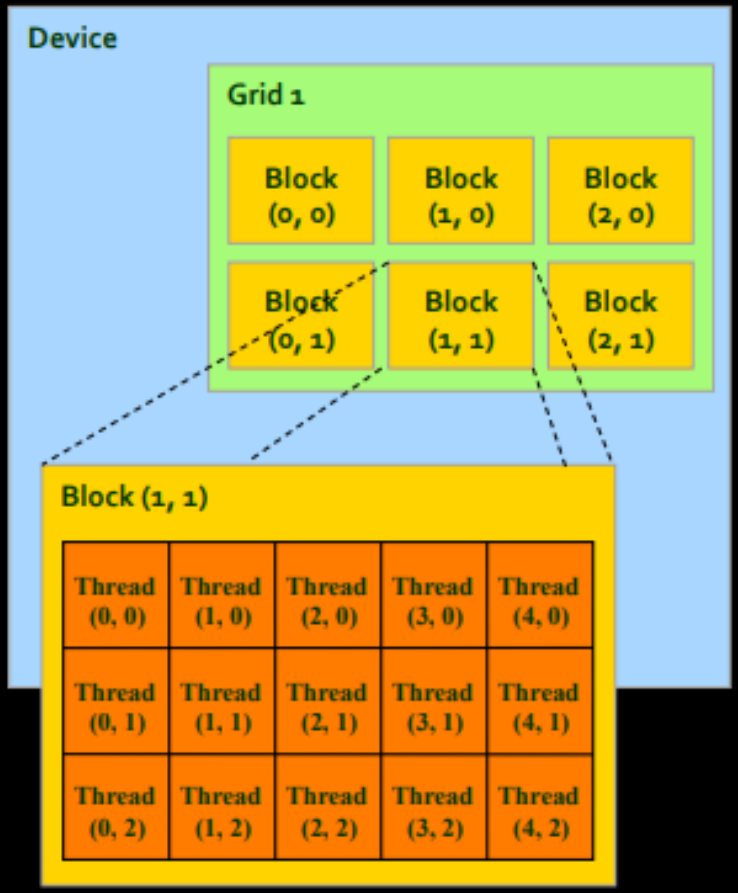
```
__global__ void increment_gpu(float *a, float b, int N)
{
       int idx = blockIdx.x * blockDim.x + threadIdx.x;
       if (idx < N)
           a[idx] = a[idx] + b;
}

void main()
{
    ...
       dim3 dimBlock (blocksize);
       dim3 dimGrid( ceil( N / (float)blocksize) );
       increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Drastic bandwidth reduction
- Thread cooperation is a powerful feature of CUDA

Thread and Block ID and Dimensions

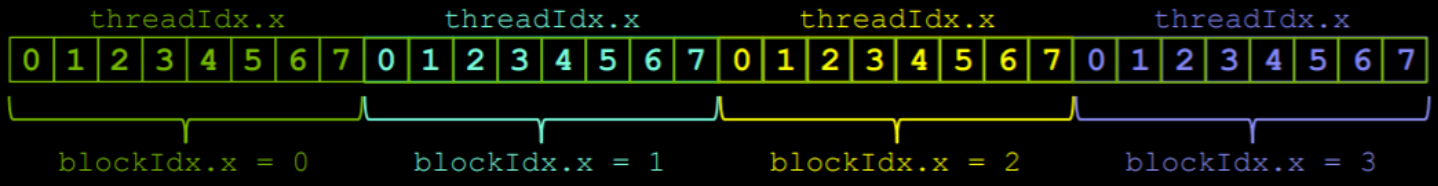
- Threads
  - 3D IDs, unique within a block
- Thread Blocks
  - 2D IDs, unique within a grid
- Dimensions set at launch
  - Can be unique for each grid
- Built-in variables
  - `threadIdx`, `blockIdx`
  - `blockDim`, `gridDim`
- Programmers usually select dimensions that simplify the mapping of the application data to CUDA threads



Indexing Arrays With Threads And Blocks

- No longer as simple as just using `threadIdx.x` or `blockIdx.x` as indices

- To index array with 1 thread per entry (using 8 threads/block)

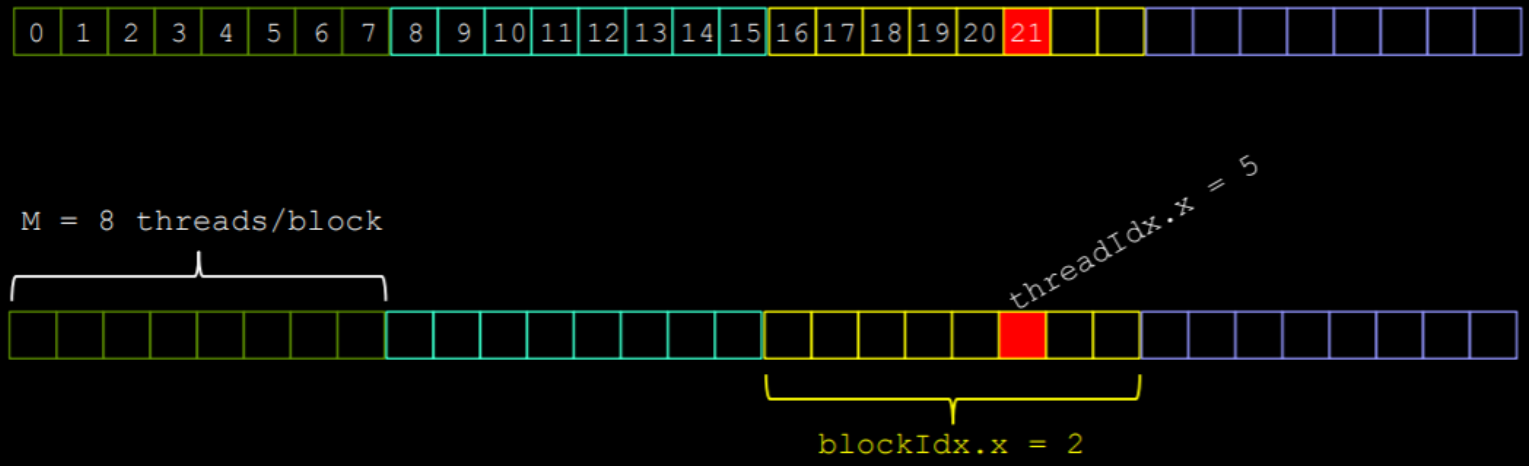


- If we have `M` threads/block, a unique array index for each entry given by

```
int index = threadIdx.x + blockIdx.x * M;

int index =       x     +     y      * width;
```

Indexing Arrays: Example

- In this example, the red entry would have an index of 21:




```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +      2      * 8;
          = 21;
```

Thread Indices

Array traversal

```
int index = threadIdx.x + blockDim.x * blockIdx.x;
```

```
blockDim.x  = 4              blockDim.x  = 4
blockIdx.x  = 0              blockIdx.x  = 1
threadIdx.x = 0, 1, 2, 3     threadIdx.x = 0, 1, 2, 3
Index       = 0, 1, 2, 3     Index       = 4, 5, 6, 7
```

The Big Difference

CPU program

```
void increment_cpu(float *a, float b, int
   N)
{
 for (int idx = 0; idx<N; idx++)
a[idx] = a[idx] + b;
}

void main()
{
…
increment_cpu(a, b, N);
}
```

GPU program

```
__global__ void increment_gpu(float *a, float b,
   int N)
 {
int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if( idx < N)                a[idx] = a[idx] + b;
}
void main() {
…..
dim3 dimBlock (blocksize);
 dim3 dimGrid( ceil( N / (float)blocksize)  )
   increment_gpu<<<dimGrid, dimBlock>>>(a,
   b, N);
}
```

to print from CUDA

```
%%cu
#include <stdio.h>

__global__ void print_kernel() {
    printf("Hello from block %d, thread %d\n", blockIdx.x, thre
adIdx.x);
}

int main() {
    print_kernel<<<10, 10>>>();
    cudaDeviceSynchronize();
}
```

An important thing to note is that *every* CUDA thread will call `printf`. In this example, we'll see 100 lines of output!

```
Hello from block 1, thread 0
Hello from block 1, thread 1
Hello from block 1, thread 2
Hello from block 1, thread 3
Hello from block 1, thread 4
Hello from block 1, thread 5
....
Hello from block 8, thread 3
Hello from block 8, thread 4
Hello from block 8, thread 5
Hello from block 8, thread 6
Hello from block 8, thread 7
Hello from block 8, thread 8
Hello from block 8, thread 9
```

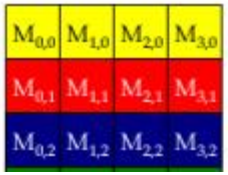
An important thing to note is that this list does *not* include program exit. If the call to cudaDeviceSynchronize() was removed from the example program above, the we would see no output.

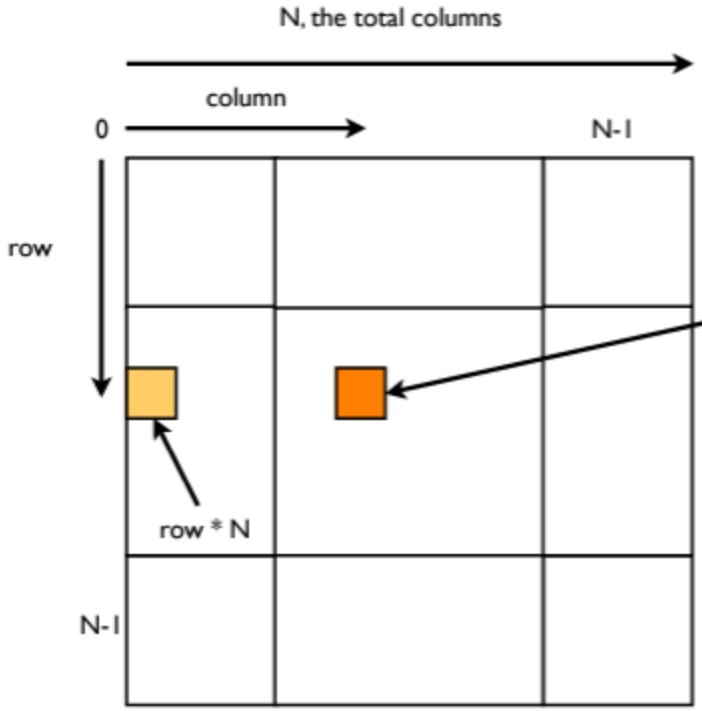
Flatten Matrices into Linear Memory

- Generally memory allocated dynamically on device (GPU) and we cannot not use two-dimensional indices (e.g. A[row][column]) to access matrices.

- We will need to know how the matrix is laid out in memory and then compute the distance from the beginning of the matrix.

- C uses **row-major** order --- rows are stored one after the other in memory, i.e. row 0 then row 1 etc.





a[row][column] = a[offset]

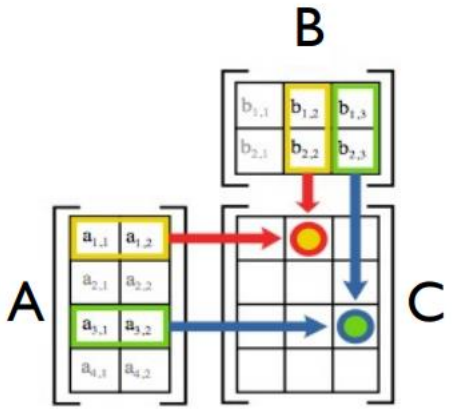
offset = column + row * N

In CUDA:

```
int col = blockIdx.x*blockDim.x+threadIdx.x;
int row = blockIdx.y*blockDim.y+threadIdx.y;
int index = col + row * N;
A[index] = …
```

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

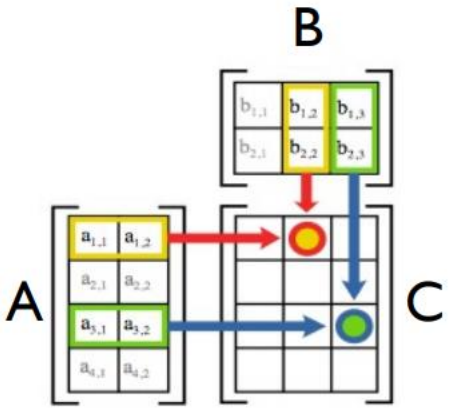
$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

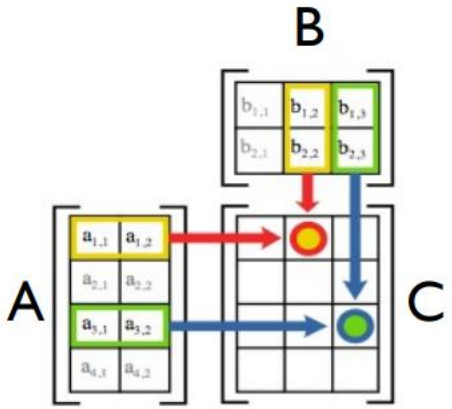
$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

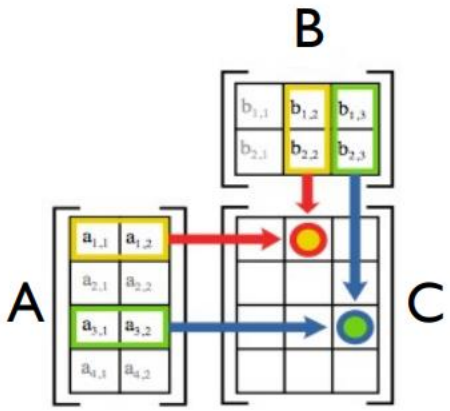
$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

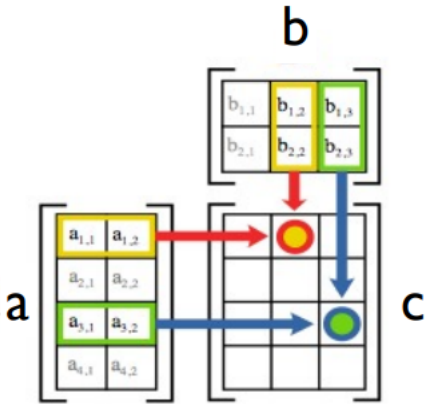
Step 1: Write the Kernel

C Function

```
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width)
{
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < width; j++) {
      int sum = 0;
      for (int k = 0; k < width; k++) {
        int m = a[i][k];
        int n = b[k][j];
        sum += m * n;
      }
      c[i][j] = sum;
    }
  }
}
```

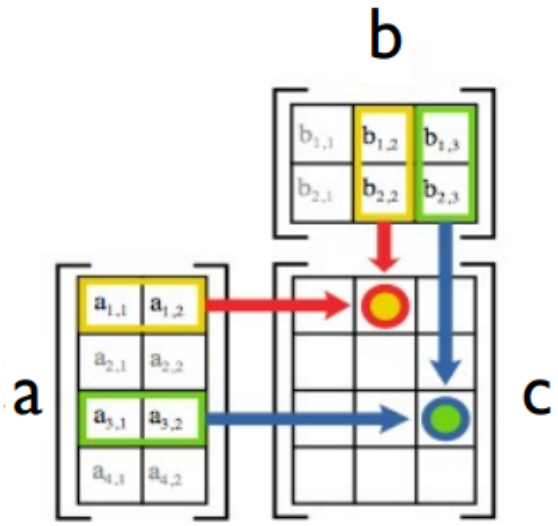
CUDA Kernel

```
__global__ void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;

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

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }
}
```



Step 2: Do the Rest

```
#define N 16
#include <stdio.h>

__global__ void matrixMult (int *a, int *b, int *c, int width);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 // initialize matrices a and b with appropriate values

 int size = N * N * sizeof(int);
 cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
 cudaMalloc((void **) &dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimGrid(1, 1);
 dim3 dimBlock(N, N);

 matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);

__global__ void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;

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

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }
}
```



CUDA C

```
#include <stdio.h>

__global__ void print_kernel() {
    printf("Hello from block %d, thread %d\n", blockIdx.x, threadIdx.x);
}

int main() {
    print_kernel<<<10, 10>>>();
    cudaDeviceSynchronize();
}
```

```
Hello from block 1, thread 0
Hello from block 1, thread 1
Hello from block 1, thread 2
Hello from block 1, thread 3
Hello from block 1, thread 4
Hello from block 1, thread 5
....
Hello from block 8, thread 3
Hello from block 8, thread 4
Hello from block 8, thread 5
Hello from block 8, thread 6
Hello from block 8, thread 7
Hello from block 8, thread 8
Hello from block 8, thread 9
```

Parallel Addition: `main()`

```
#define N  512
int main( void ) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;     // device copies of a, b, c
    int size = N * sizeof( int );   // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

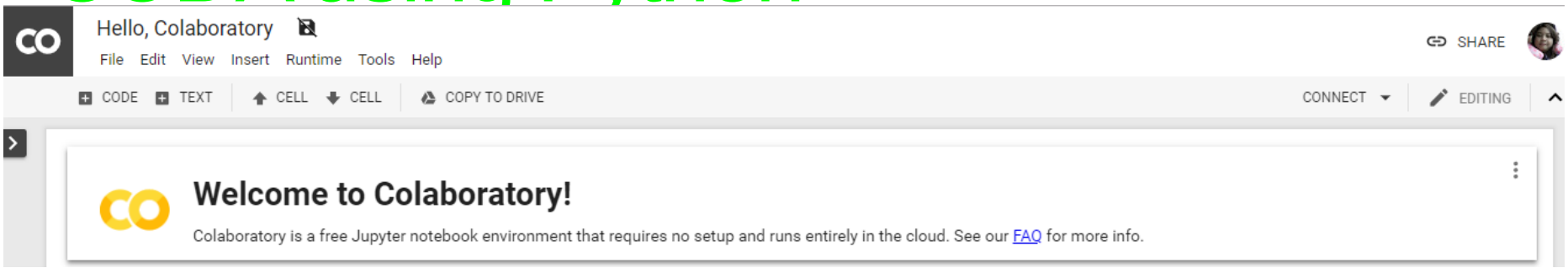
Parallel Addition: `main()` (cont)

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with N parallel blocks
    add<<< N, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

CUDA using Python

- Anaconda/Python 3.6.1/Jupyter notebook
- CUDA Toolkit
- Numba package

CUDA using Python




```
!nvidia-smi
```

```
Tue Oct 30 12:11:30 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.44                 Driver Version: 396.44                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   72C    P8    33W / 149W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

CUDA using Python

```
!apt-get install nvidia-cuda-toolkit
```

```
  libvdpau-va-gl1 nvidia-vdpau-driver nvidia-legacy-340xx-vdpau-driver
  mesa-utils
Recommended packages:
  libnvcuvid1
The following NEW packages will be installed:
  adwaita-icon-theme at-spi2-core ca-certificates-java cpp-6
  dconf-gsettings-backend dconf-service fontconfig fonts-dejavu-core
```

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Built on Tue_Jun_12_23:07:04_CDT_2018
Cuda compilation tools, release 9.2, V9.2.148
```

```
!pip3 install numba
```

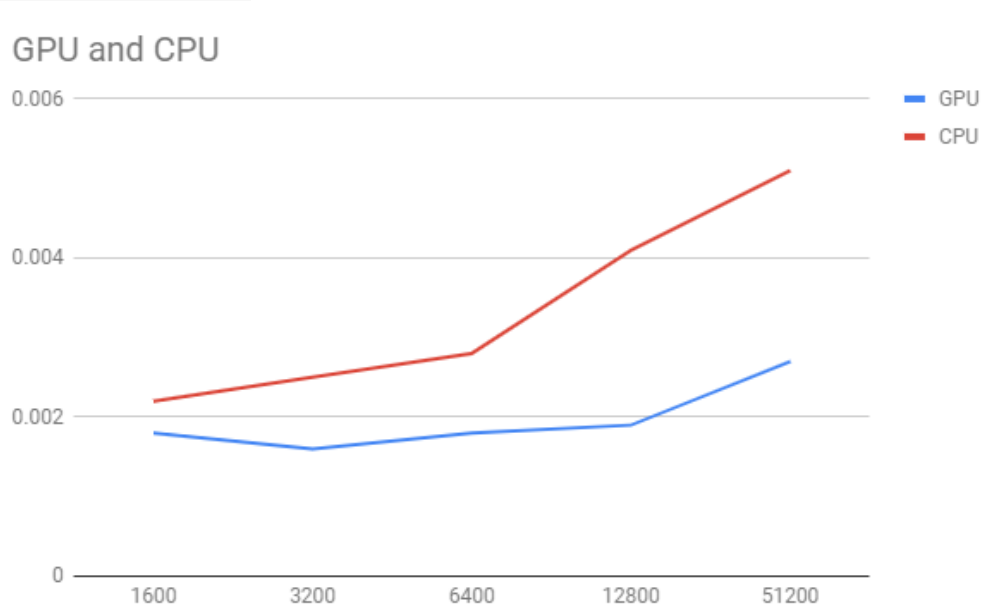
```
Collecting numba
  Downloading https://files.pythonhosted.org/packages/42/45/8d5fc45e5f760ac65906ba48dec98e99e7920c96783ac7248c5e31c9464e/numba-0.40.1-cp36-cp36m-manylinux1_x8
    100% |████████████████████████████████| 3.2MB 8.3MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from numba) (1.14.6)
Collecting llvmlite>=0.25.0dev0 (from numba)
  Downloading https://files.pythonhosted.org/packages/34/fb/f9c2e9e0ef2b54c52f0b727cf6af75b68c3d7ddb6d88c8d557b1b16bc1ab/llvmlite-0.25.0-cp36-cp36m-manylinux1
    100% |████████████████████████████████| 16.1MB 2.8MB/s
Installing collected packages: llvmlite, numba
Successfully installed llvmlite-0.25.0 numba-0.40.1
```

Vector add GPU

```
from __future__ import print_function
from timeit import default_timer as time
import numpy as np
from numba import cuda

@cuda.jit('(f4[:], f4[:], f4[:])')
def cuda_sum(a, b, c):
    i = cuda.grid(1)
    c[i] = a[i] + b[i]

griddim = 50, 1
blockdim = 32, 1, 1
N = griddim[0] * blockdim[0]
print("N", N)
cuda_sum_configured = cuda_sum.configure(griddim, blockdim)
a = np.array(np.random.random(N), dtype=np.float32)
b = np.array(np.random.random(N), dtype=np.float32)
c = np.empty_like(a)

ts = time()
cuda_sum_configured(a, b, c)
te = time()
print(te - ts)


assert (a + b == c).all()
print(c)
```

Vector add CPU

```
from timeit import default_timer as time
import numpy as np
N = 1600
def cpu_sum(a, b, c):
    for i in range(0, N):
      c[i] = a[i] + b[i]


a = np.array(np.random.random(N), dtype=np.float32)
b = np.array(np.random.random(N), dtype=np.float32)
c = np.empty_like(a)

ts = time()
cpu_sum(a, b, c)
te = time()
print(te - ts)

print(c)
```



LAB 15 CUDA- MATRIX

Minimal Kernels

```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

Common Pattern!

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

int idx = blockDim.x * blockId.x + threadIdx.x;
will map from local index threadIdx to global index

NB: blockDim should be >= 32 in real code, this is just an example

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}

void main()
{
   .....
       increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
       int idx = blockIdx.x * blockDim.x + threadIdx.x;
       if (idx < N)
           a[idx] = a[idx] + b;
}

void main()
{
    ...
       dim3 dimBlock (blocksize);
       dim3 dimGrid( ceil( N / (float)blocksize) );
       increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Drastic bandwidth reduction
- Thread cooperation is a powerful feature of CUDA

Thread Indices

Array traversal

```
int index = threadIdx.x + blockDim.x * blockIdx.x;
```

```
blockDim.x  = 4              blockDim.x  = 4
blockIdx.x  = 0              blockIdx.x  = 1
threadIdx.x = 0, 1, 2, 3     threadIdx.x = 0, 1, 2, 3
Index       = 0, 1, 2, 3     Index       = 4, 5, 6, 7
```

to print from CUDA

```
%%cu
#include <stdio.h>

__global__ void print_kernel() {
    printf("Hello from block %d, thread %d\n", blockIdx.x, threadIdx.x);
}

int main() {
    print_kernel<<<10, 10>>>();
    cudaDeviceSynchronize();
}
```

An important thing to note is that *every* CUDA thread will call `printf`. In this example, we'll see 100 lines of output!

```
Hello from block 1, thread 0
Hello from block 1, thread 1
Hello from block 1, thread 2
Hello from block 1, thread 3
Hello from block 1, thread 4
Hello from block 1, thread 5
....
Hello from block 8, thread 3
Hello from block 8, thread 4
Hello from block 8, thread 5
Hello from block 8, thread 6
Hello from block 8, thread 7
Hello from block 8, thread 8
Hello from block 8, thread 9
```

An important thing to note is that this list does *not* include program exit. If the call to cudaDeviceSynchronize() was removed from the example program above, the we would see no output.

CUDA C

```
#include <stdio.h>

__global__ void print_kernel() {
    printf("Hello from block %d, thread %d\n", blockIdx.x, threadIdx.x);
}

int main() {
    print_kernel<<<10, 10>>>();
    cudaDeviceSynchronize();
}
```

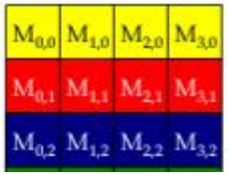
```
Hello from block 1, thread 0
Hello from block 1, thread 1
Hello from block 1, thread 2
Hello from block 1, thread 3
Hello from block 1, thread 4
Hello from block 1, thread 5
....
Hello from block 8, thread 3
Hello from block 8, thread 4
Hello from block 8, thread 5
Hello from block 8, thread 6
Hello from block 8, thread 7
Hello from block 8, thread 8
Hello from block 8, thread 9
```

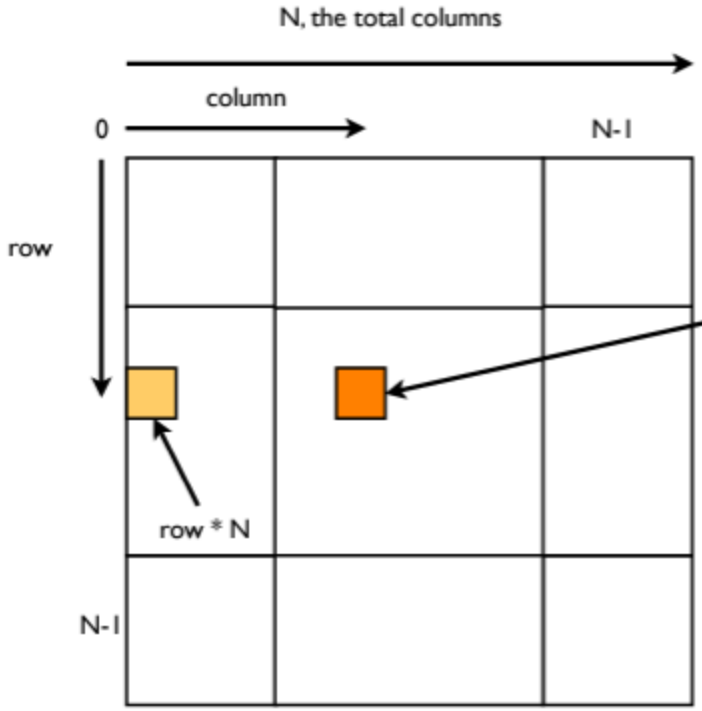
Flatten Matrices into Linear Memory

- Generally memory allocated dynamically on device (GPU) and we cannot not use two-dimensional indices (e.g. A[row][column]) to access matrices.

- We will need to know how the matrix is laid out in memory and then compute the distance from the beginning of the matrix.

- C uses **row-major** order --- rows are stored one after the other in memory, i.e. row 0 then row 1 etc.





a[row][column] = a[offset]

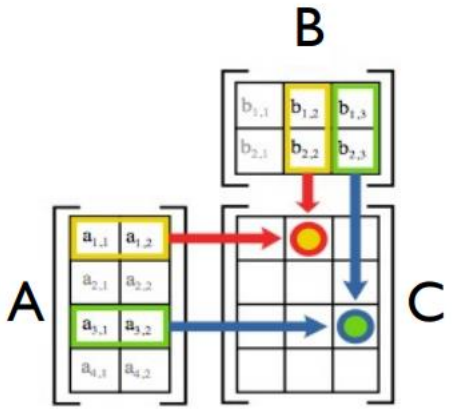
offset = column + row * N

In CUDA:

```
int col = blockIdx.x*blockDim.x+threadIdx.x;
int row = blockIdx.y*blockDim.y+threadIdx.y;
int index = col + row * N;
A[index] = …
```

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

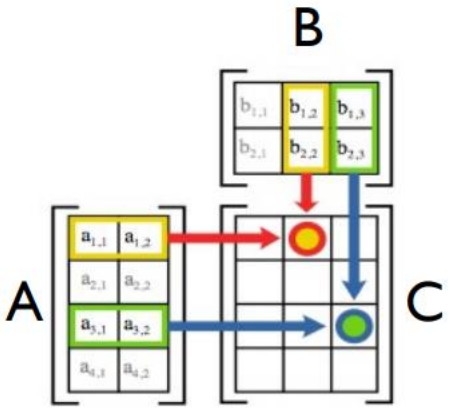
$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

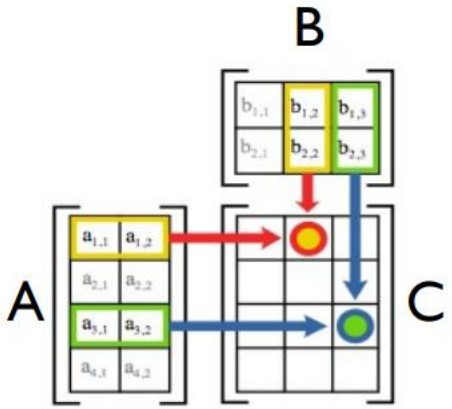
$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

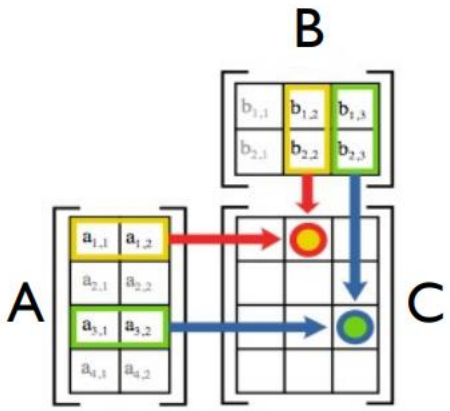
$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Multiplication Review

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$



- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.

$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Matrix Mul

```
%%cu
#include <stdio.h>
__global__ void matrixFill ( int *x )
{
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
int col = by*blockDim.y + ty;
int row = bx*blockDim.x + tx;
int dim =blockDim.x*gridDim.x;
int i = row*dim + col;
x[i] = i;
}
```

```
int main ( int argc, char* argv[] )
{
const int xb = 2; /* gridDim.x */
const int yb = 2; /* gridDim.y */
const int zb = 1; /* gridDim.z */
const int xt = 2; /* blockDim.x */
const int yt = 2; /* blockDim.y */
const int zt = 1; /* blockDim.z */
const int n = xb*yb*zb*xt*yt*zt;
printf("allocating array of length %d...\n",n);
int *xhost = (int*)calloc(n,sizeof(int));
for(int i=0; i<n; i++)
xhost[i] = -1.0;
```

```
int *xdevice;
size_t sx = n*sizeof(int);
cudaMalloc((void**)&xdevice,sx);
cudaMemcpy(xdevice,xhost,sx,cudaMemcpyHostToDevice);
dim3 dimGrid(xb,yb,zb);
dim3 dimBlock(xt,yt,zt);
matrixFill<<<dimGrid,dimBlock>>>(xdevice);
cudaMemcpy(xhost,xdevice,sx,cudaMemcpyDeviceToHost);
cudaFree(xdevice);
```

```
int *p = xhost;
for(int i1=0; i1 < xb; i1++)
for(int i2=0; i2 < yb; i2++)
for(int i3=0; i3 < zb; i3++)
for(int i4=0; i4 < xt; i4++)
for(int i5=0; i5 < yt; i5++)
for(int i6=0; i6 < zt; i6++)
printf("x[%d][%d][%d][%d][%d][%d] = %d\n",i1,i2,i3,i4,i5,i6,*
(p++));
return 0;
}
```

Lab Report

- SAMPLE – Moodle

FAT Lab

- OpenMP Program (30 Mark Out of 50)
  - Aim and Program (20 Mark)
  - Output (10 Mark)
- Vtune Profiler (10 Mark Out of 50)
- CUDA Programming Quiz (10 Mark Out of 50)