

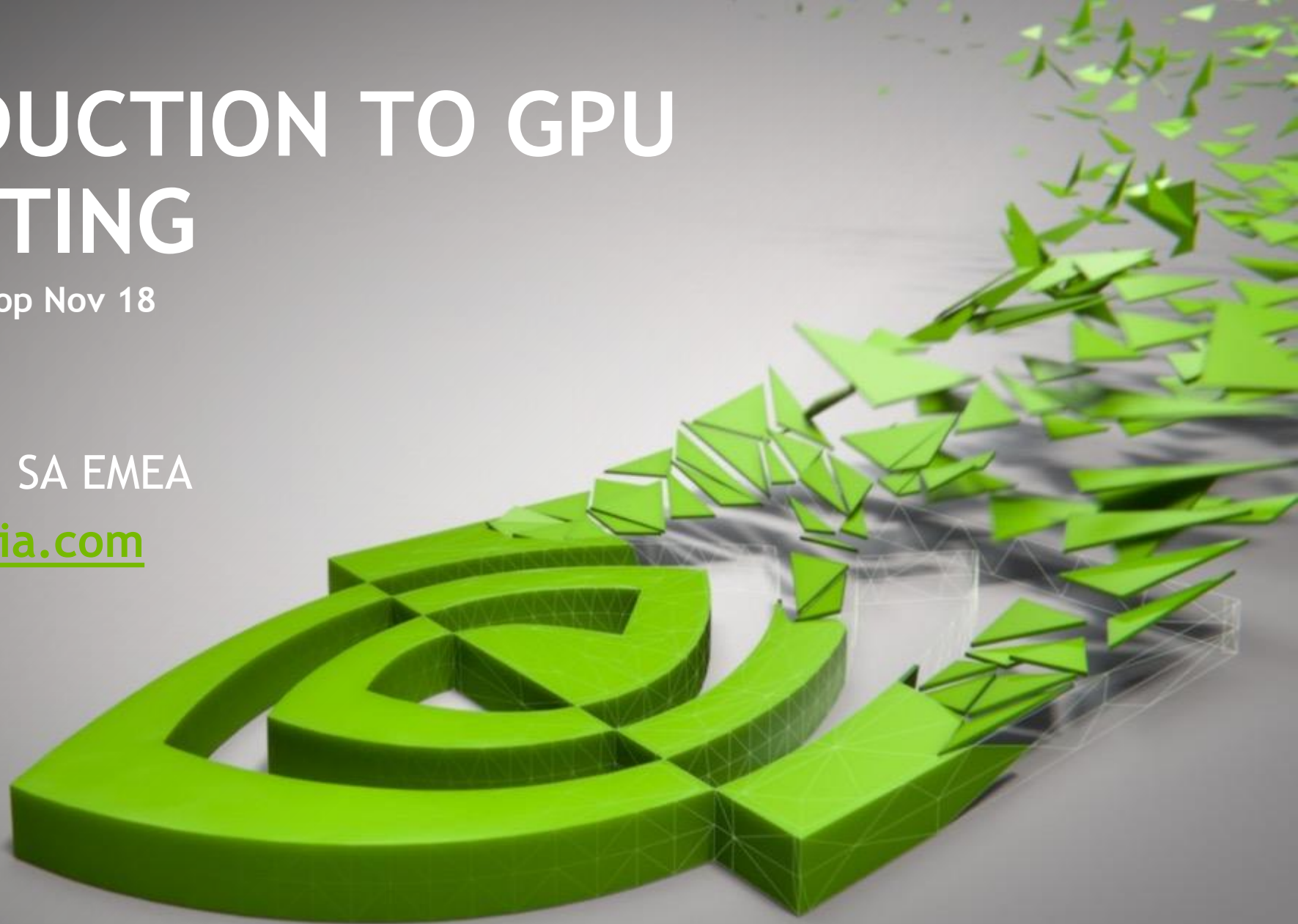
# INTRODUCTION TO GPU COMPUTING

Comtegra Workshop Nov 18



Gunter Roeth, SA EMEA

[gunterr@nvidia.com](mailto:gunterr@nvidia.com)



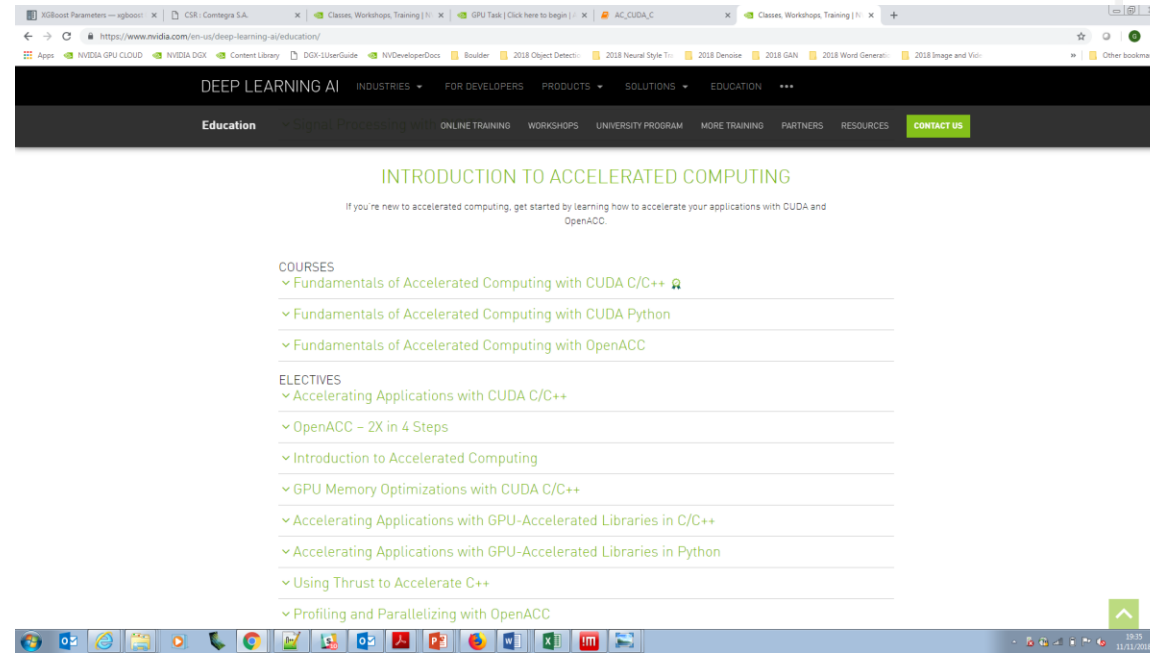
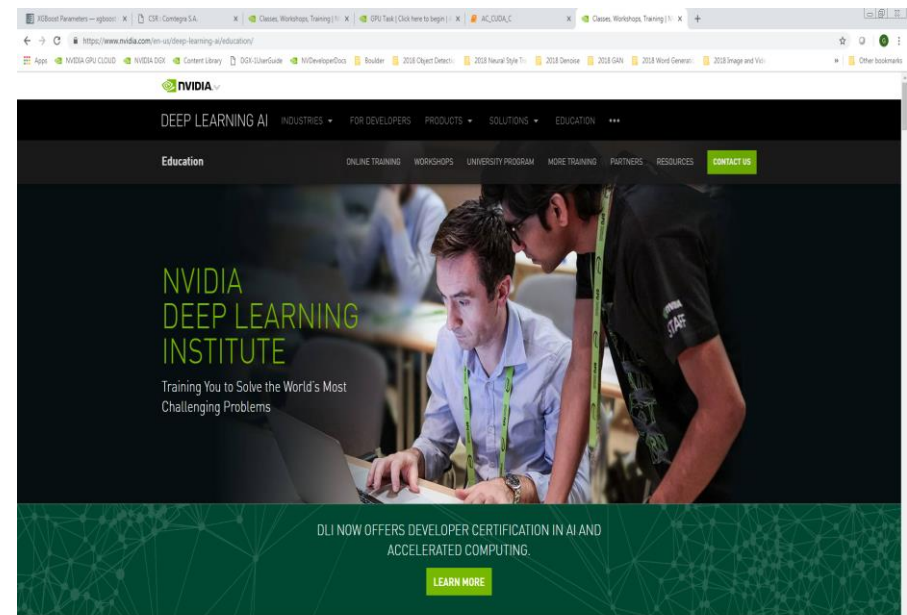
# NAVIGATING TO COURSES

1. Navigate to:  
<https://www.nvidia.com/en-us/deep-learning-ai/education/>

2. Google search for  
nvidia dli

3. Scroll down  
Training Online ELECTIVES

Use NV Developer login or new  
account.



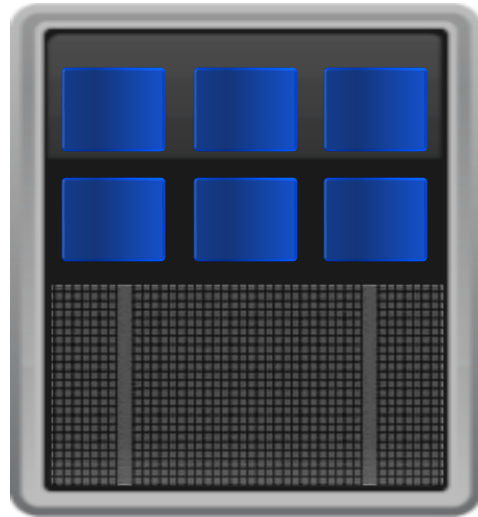
# GPU CPU DIFFERENCES

# Accelerated Computing CPU/GPU differences

*10x Performance & 5x Energy Efficiency for HPC*

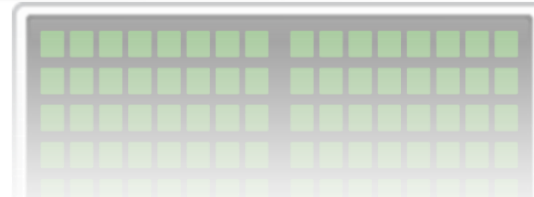
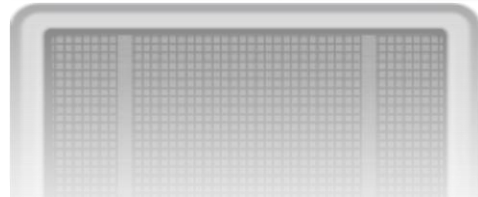
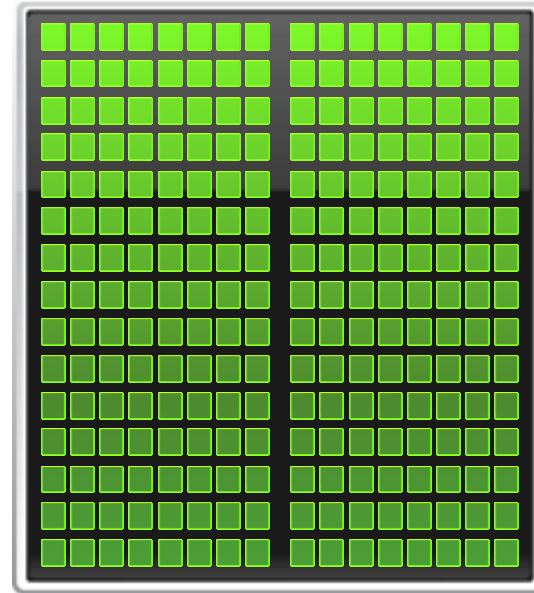
**CPU**

Optimized for  
Serial Tasks



**GPU Accelerator**

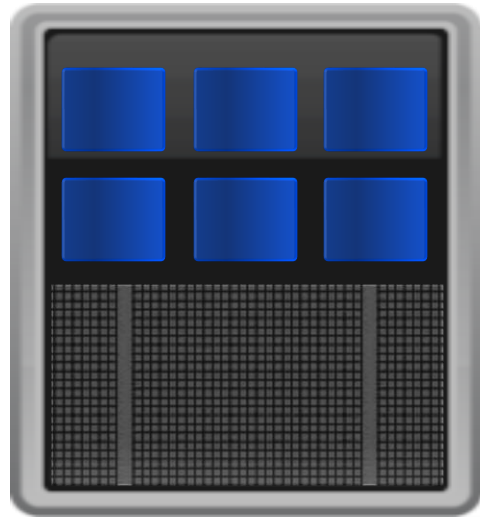
Optimized for  
Parallel Tasks



# Accelerated Computing

*10x Performance & 5x Energy Efficiency for HPC*

**CPU**  
Optimized for  
Serial Tasks

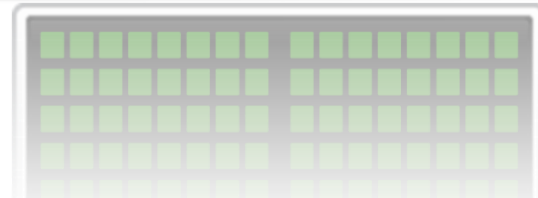
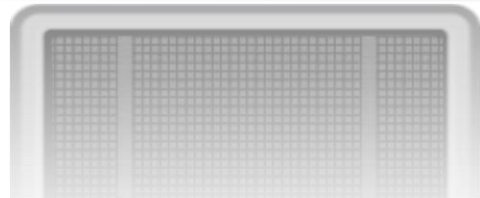


## CPU Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

## CPU Weaknesses

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt



# Accelerated Computing

*10x Performance & 5x Energy Efficiency for HPC*

## GPU Strengths

- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

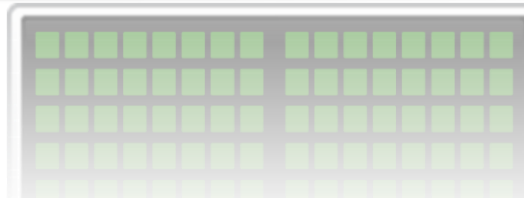
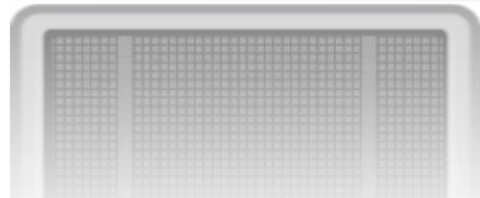
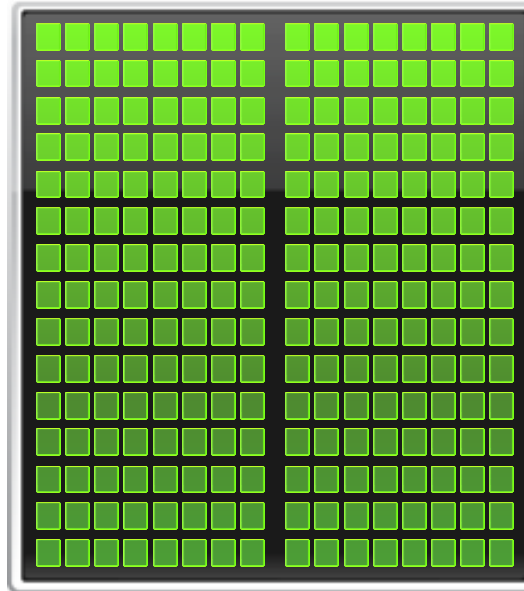
## GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance

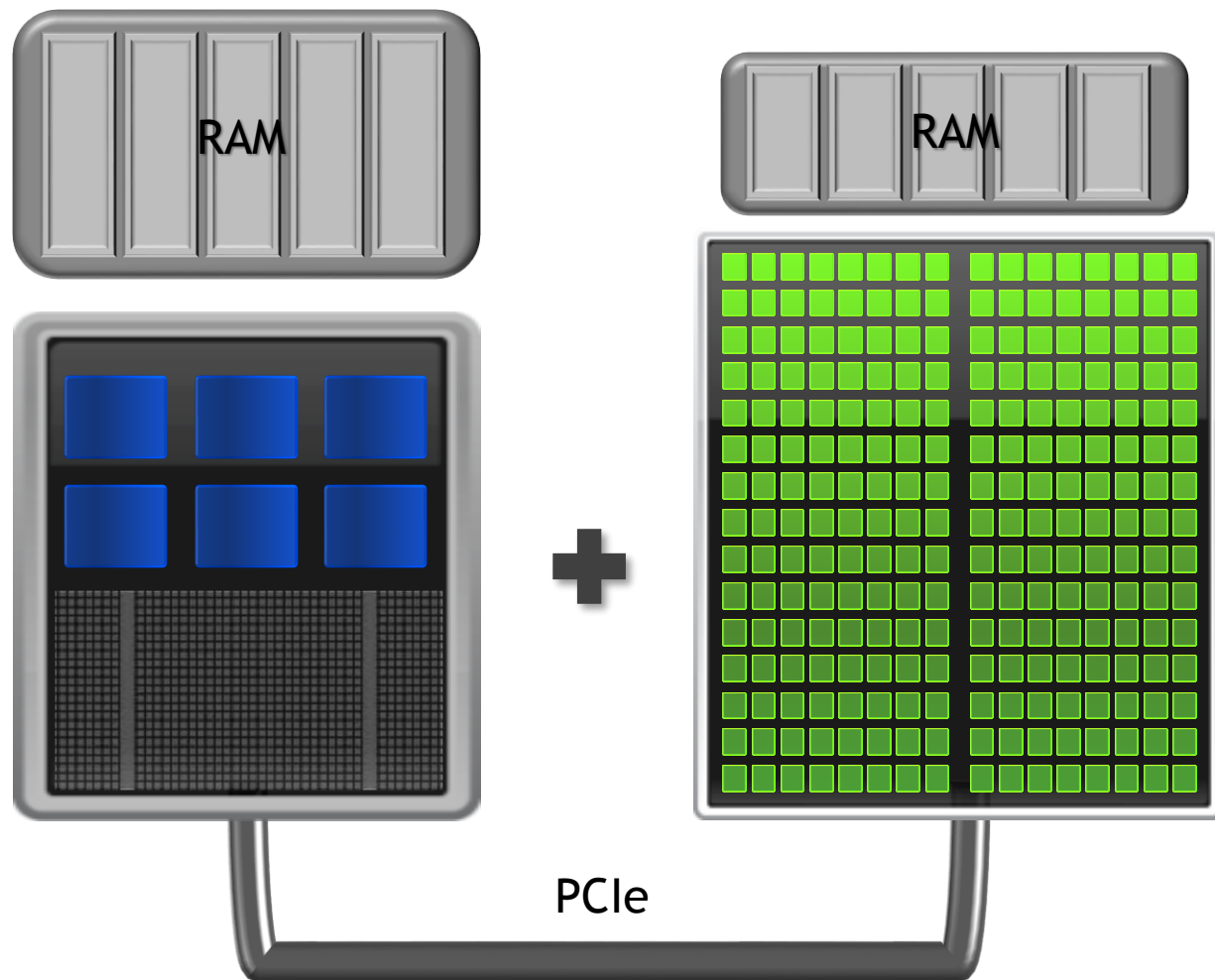


## GPU Accelerator

Optimized for  
Parallel Tasks



# Accelerator Nodes



CPU and GPU have distinct memories

- CPU generally larger and slower
- GPU generally smaller and faster

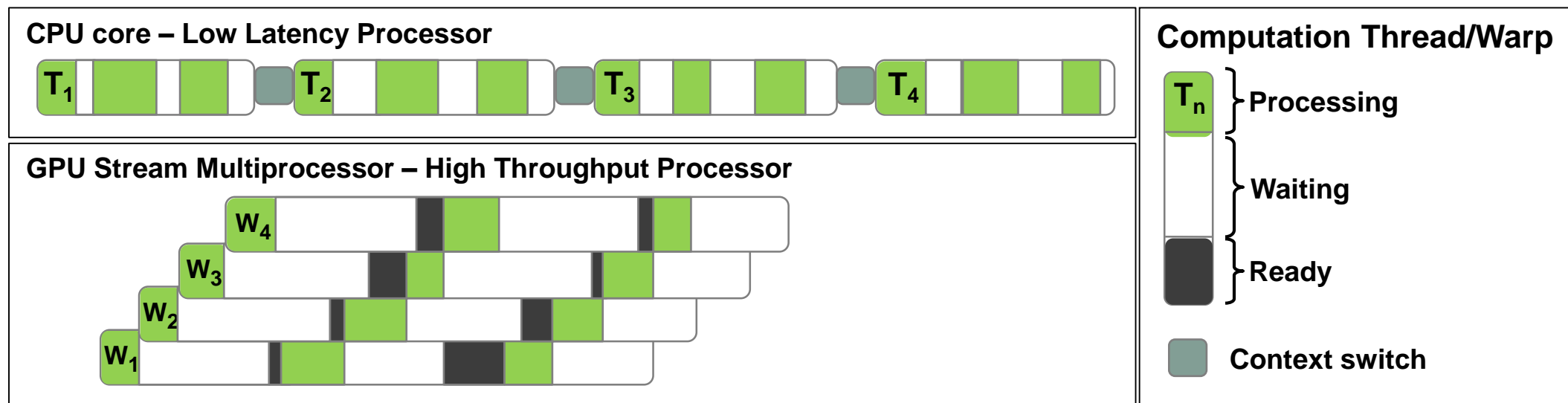
CPU and GPU communicate via PCIe

- Data must be copied between these memories over PCIe
- PCIe Bandwidth is much lower than either memories

# LOW LATENCY OR HIGH THROUGHPUT?

**CPU** architecture must **minimize latency** within each thread

**GPU** architecture **hides latency** with computation from other thread warps





# Some Terminology



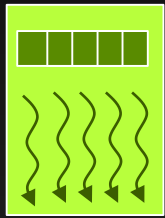
- **Kernel** - A function which runs on the GPU
  - A kernel is launched on a **grid** of **thread blocks**.
  - The grid and block size are called the **launch configuration**.
- **Global Memory** - GPU's on-board DRAM
- **Shared Memory** - On-chip fast memory local to a thread block

# SIMT Execution Model

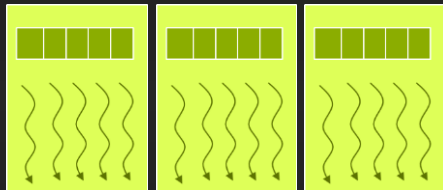
## Software



Thread



Thread Block

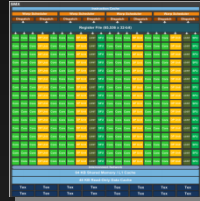


Grid

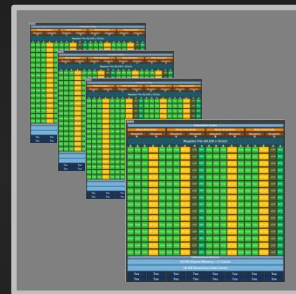
## Hardware



CUDA  
Core



Multiprocessor



Device

Threads are executed by CUDA Cores

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A **kernel** is launched as a grid of thread blocks



## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# HELLO WORLD!

# Hello World!



```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

## Output:

```
$ nvcc  
hello_world.cu  
$ a.out  
Hello World!  
$
```

# Hello World! with Device Code



```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    cudaDeviceSynchronize();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Two new syntactic elements...

# Hello World! with Device Code



```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`

# Hello World! with Device Code



```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!



## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

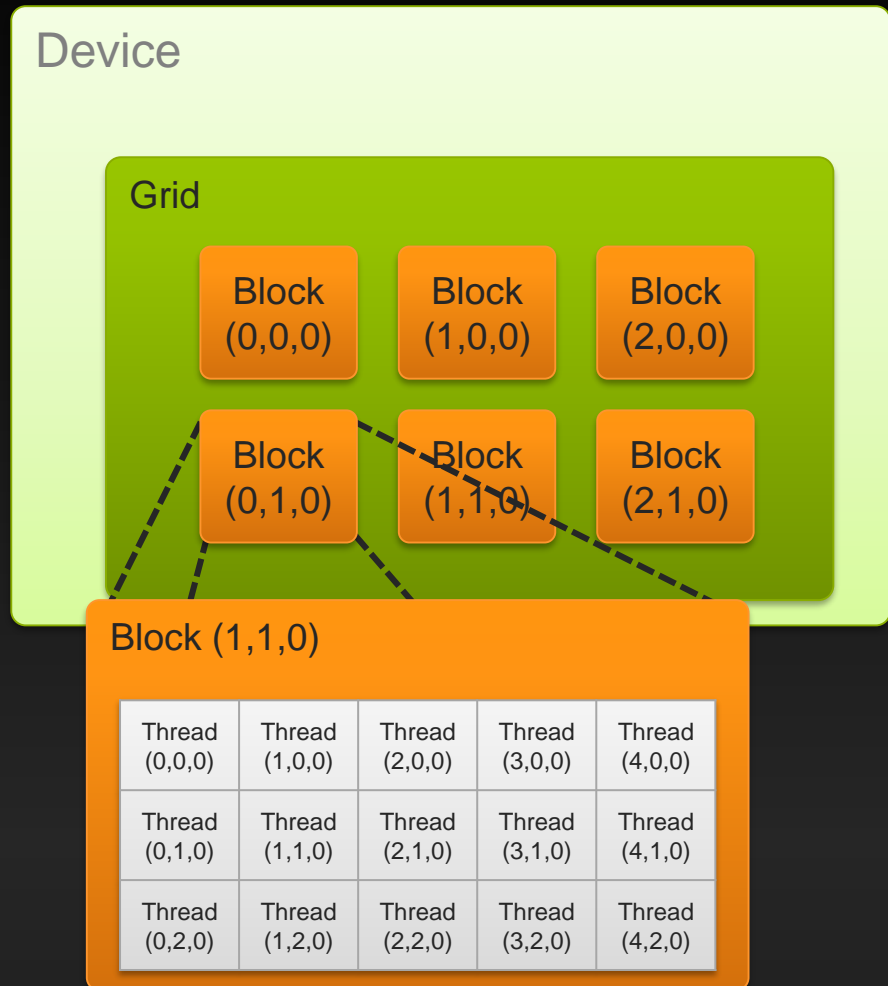
# RUNNING IN PARALLEL BLOCKS & THREADS



# IDs and Dimensions



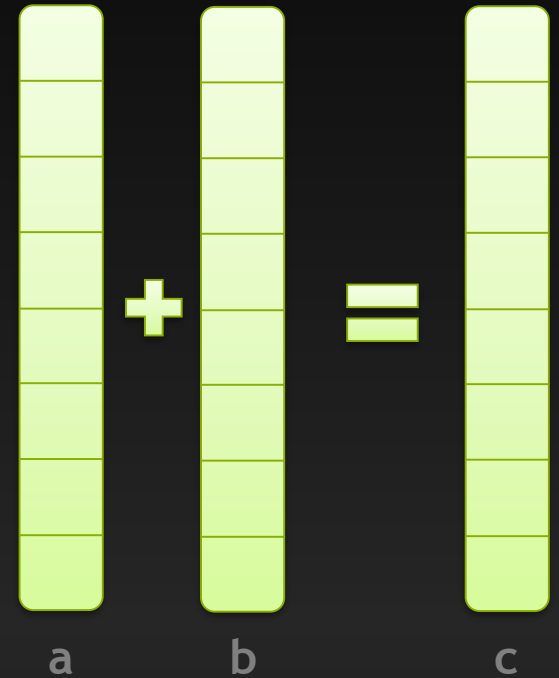
- **Built-in variables:**
  - `threadIdx.[x y z]`
    - **thread index within a thread block**
  - `blockIdx.[x y z]`
    - **block index within the grid.**
  - `blockDim.[x y z]`
    - **Number of threads in each block.**
  - `gridDim.[x y z]`
    - **Number of blocks in the grid.**



# Parallel Programming in CUDA C/C++



- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition



# Vector Add: GPU's Hello World



- GPU is *parallel computation* oriented.
  - Vector add is a very simple parallel algorithm.
- **Problem:  $C = A + B$** 
  - C, A, B are length N vectors

```
void vecAdd(int n, float * a,  
            float * b, float * c)  
{  
    for(int i=0; i<n; i++)  
    {  
        c[i] = a[i] + b[i];  
    }  
}
```

# Vector Add: GPU's Hello World



- GPU is *parallel computation* oriented.
  - Vector add is a very simple parallel algorithm.
- **Problem:  $C = A + B$** 
  - C, A, B are length N vectors

```
void vecAdd(int n, float * a,  
            float * b, float * c)  
{  
    for(int i=0; i<n; i++)  
    {  
        c[i] = a[i] + b[i];  
    }  
}
```

```
void main()  
{  
    int N = 1024;  
    float * a, *b, *c;  
    a = (float*)malloc(N*sizeof(float));  
    b = (float*)malloc(N*sizeof(float));  
    c = (float*)malloc(N*sizeof(float));  
    memset(c, 0, N*sizeof(float));  
    init_rand_f(a, N);  
    init_rand_f(b, N);  
  
    vecAdd(N, a, b, c);  
}
```

# Moving Computation to the GPU



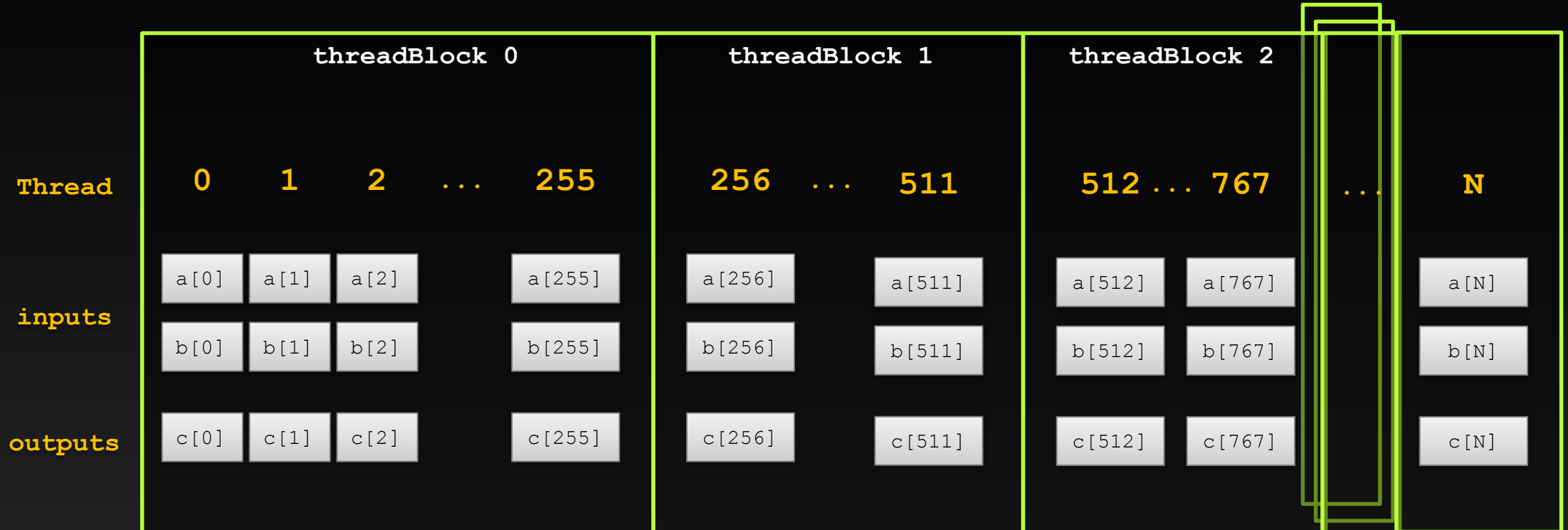
- Step 1: Identify parallelism.
  - Design problem decomposition
- Step 2: Write your GPU Kernel
- Step 3: Setup the Problem
- Step 4: Launch the Kernel
- Step 5: Copy results back from GPU

Remember: big font means important

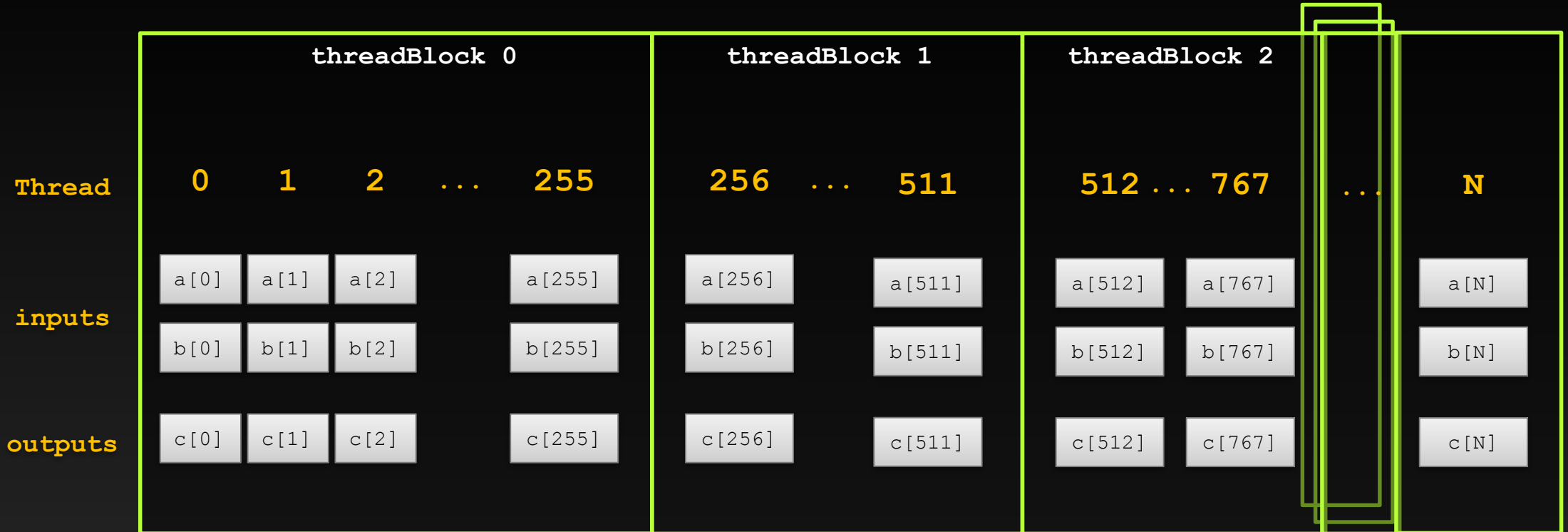
# Parallelization of VecAdd



# Parallelization of VecAdd



# Parallelization of VecAdd



$$\text{work index } i = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x};$$

index of the thread within a thread block

index of the threadblock within the grid

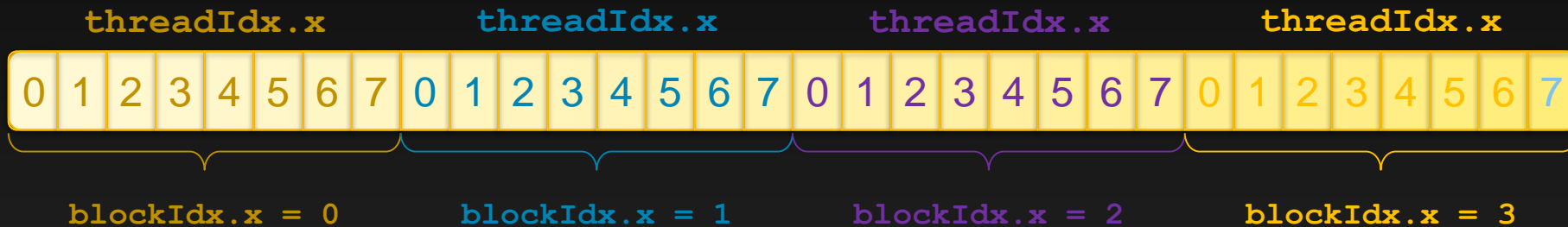
number of threads within each block



# Indexing Arrays with Blocks and Threads



- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)



- With `blockDim.x` threads per block, a unique index for each thread is given by:
  - `int index = blockIdx.x * blockDim.x + threadIdx.x`

# Vector Add: GPU's Hello World



- Step 2: Make it a GPU Kernel

Identify this function as something to be run on the GPU.

i is a different value for each thread.

```
__global__ void vecAdd(int n, float * a, float * b, float * c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < n){
        c[i] = a[i] + b[i];
    }
}
```

Protect against invalid access if too many threads are launched.

# Step 3: Setup the problem



```
void main()
{
    int N = 1024;
    float *a, *b, *c;
    float *devA, *devB, *devC;
    a = (float*)malloc(N*sizeof(float));
    b = (float*)malloc(N*sizeof(float));
    c = (float*)malloc(N*sizeof(float));
    cudaMalloc(&devA, N*sizeof(float));
    cudaMalloc(&devB, N*sizeof(float));
    cudaMalloc(&devC, N*sizeof(float));

    memset(c, 0, N*sizeof(float));
    init_rand_f(a, N);
    init_rand_f(b, N);
    cudaMemcpy(devA, a, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(devB, b, N*sizeof(float), cudaMemcpyHostToDevice);
}
```

Pointers to storage on the GPU.

Allocate memory in the GPU Global Memory.

Copy data to the GPU.

# Step 4: Launch the GPU Kernel



call function by name  
as usual

Angle Brackets: Specify  
**launch configuration**  
for the kernel.

Normal parameter passing  
syntax. Note that *devA*,  
*devB*, and *devC* are  
**device pointers**.  
They point to memory  
allocated on the GPU.

```
void main()  
{  
  ...  
  vecAdd<<<(N+127)/128, 128>>>(N, devA, devB, devC);  
  ...  
}
```

First argument is  
the number of  
**thread blocks**  
(rounding up)

Second argument is  
the shape of  
(i.e, number of threads in)  
each **thread block**

# Step 5: Copy data back.



```
void main()
{
    ...

    cudaMemcpy(c, devC, N*sizeof(float), cudaMemcpyDeviceToHost);

    ...
}
```



# Handling Arbitrary Vector Sizes

Typical problems are not even multiples of `blockDim.x`  
Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = blockDim.x * blockIdx.x + threadIdx.x;  
    if(index < n)  
        c[index] = a[index] + b[index];  
}
```

Update the kernel launch:

```
add <<<(N + M - 1) / M, M>>>(a, b, c, N);
```

**N** = problem size, **M** = block size

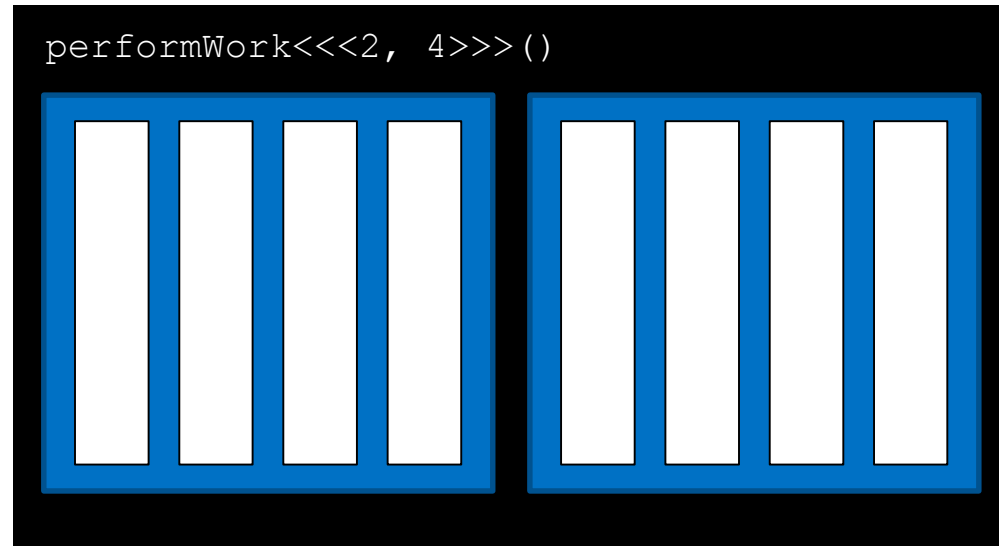
# Multi-Dimensional Thread-Blocks

- CUDA supports up to 3-dimensional grids and blocks.
  - `threadIdx.{x,y,z}`
  - `blockIdx.{x,y,z}`
  - `blockDim.{x,y,z}`
  - `gridDim.{x,y,z}`
- Easily accelerate multiple nested loops
- Launch a 2D grid using dim3 structures
  - `kernel<<<dim3(32,32),dim3(32,4)>>>()`

Order of loops is important for performance. Ideally `threadIdx.x` is consecutive in memory

```
for(int y=...) -> int y=blockIdx.y*blockDim.y+threadIdx.y;  
for(int x=...) -> int x =blockIdx.x*blockDim.x+threadIdx.x;
```

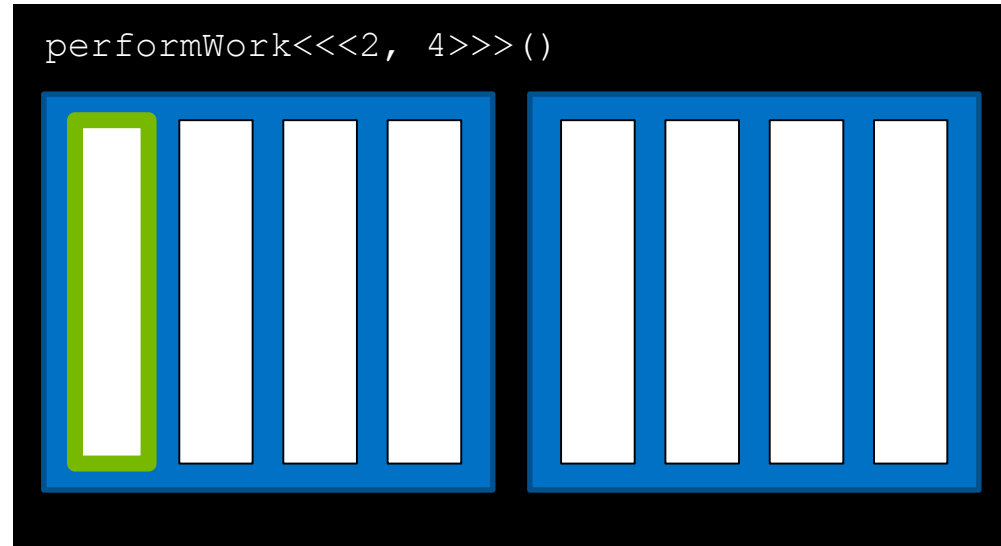
GPU





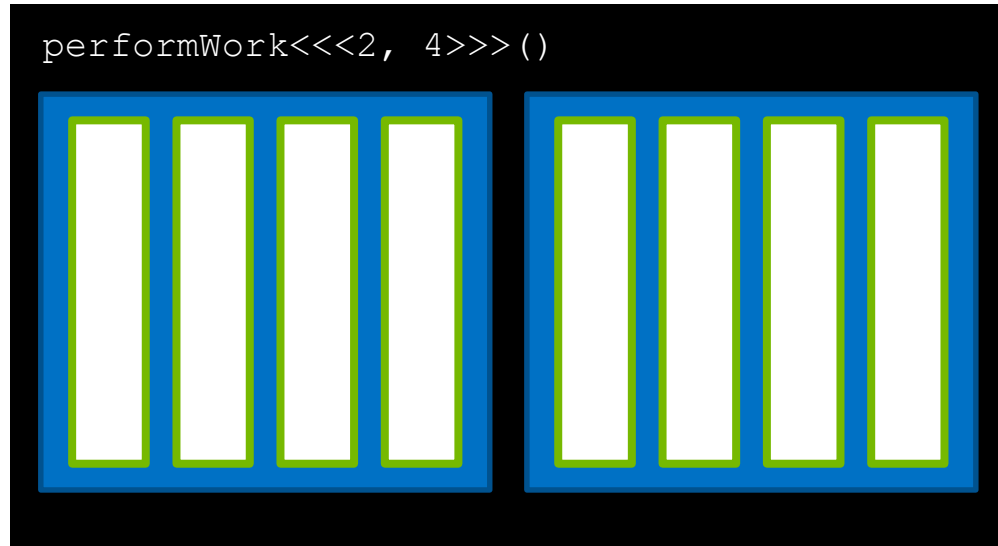
GPU work is done in a **thread**

GPU



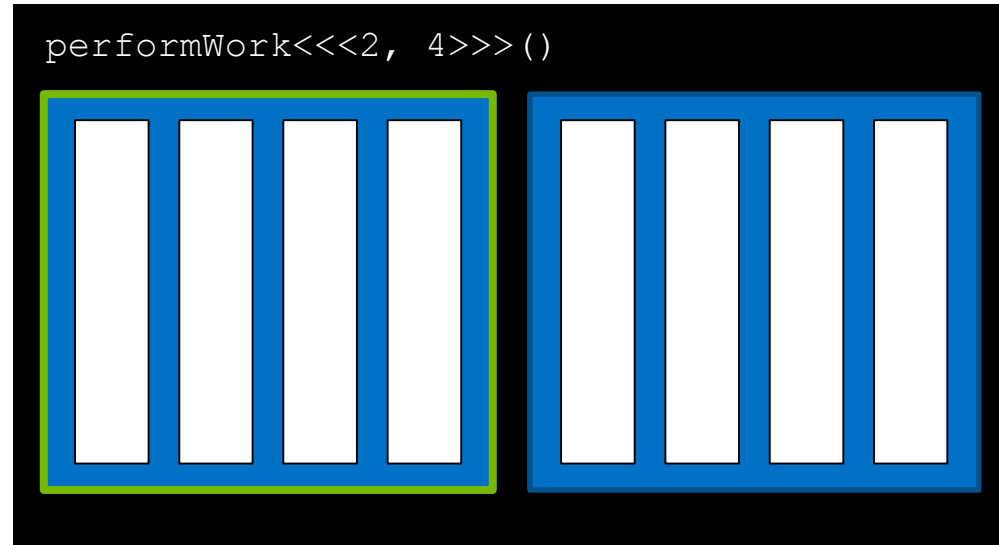
Many threads run in parallel

GPU



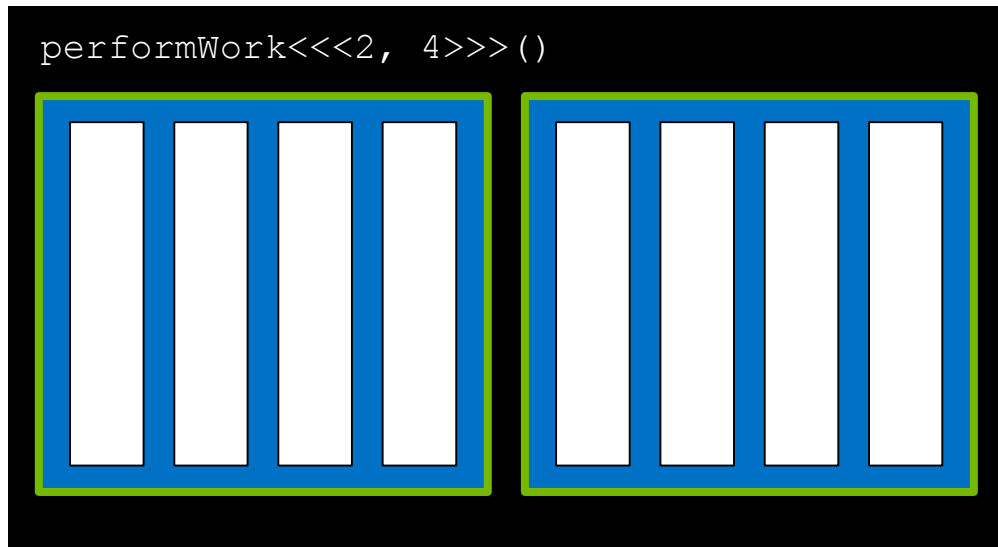
A collection of threads is a **block**

GPU



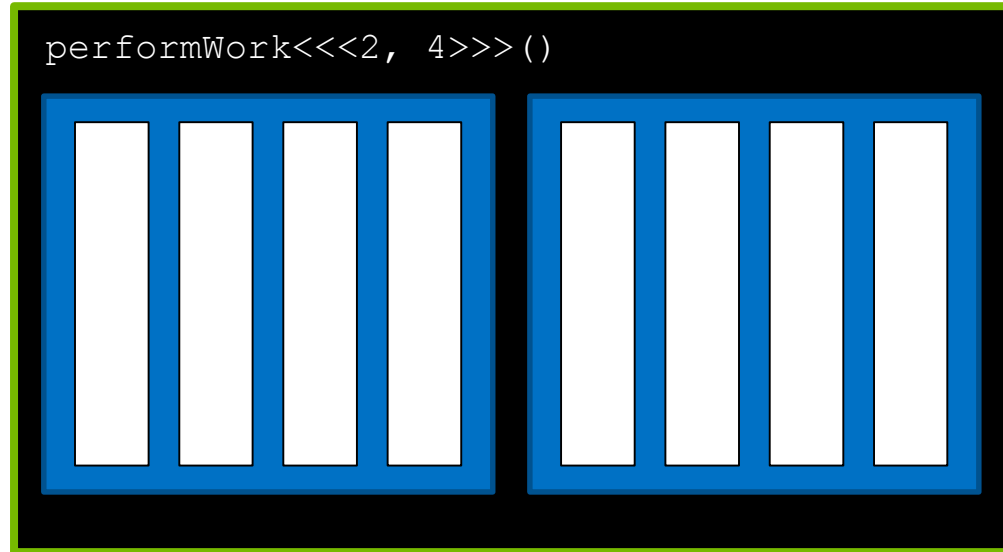
There are many blocks

GPU



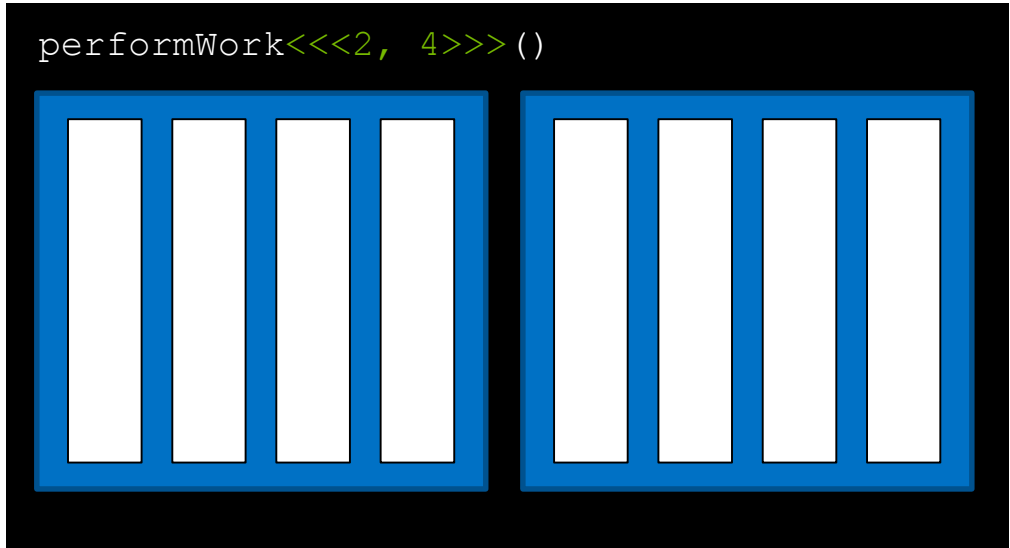
A collection of blocks associated with a given kernel launch is a **grid**

GPU



Kernels are **launched** with an **execution configuration**

GPU



## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# COOPERATING THREADS

# 1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:





# Implementing Within a Block

- Each thread processes one output element
  - `blockDim.x` elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times



# Sharing Data Between Threads

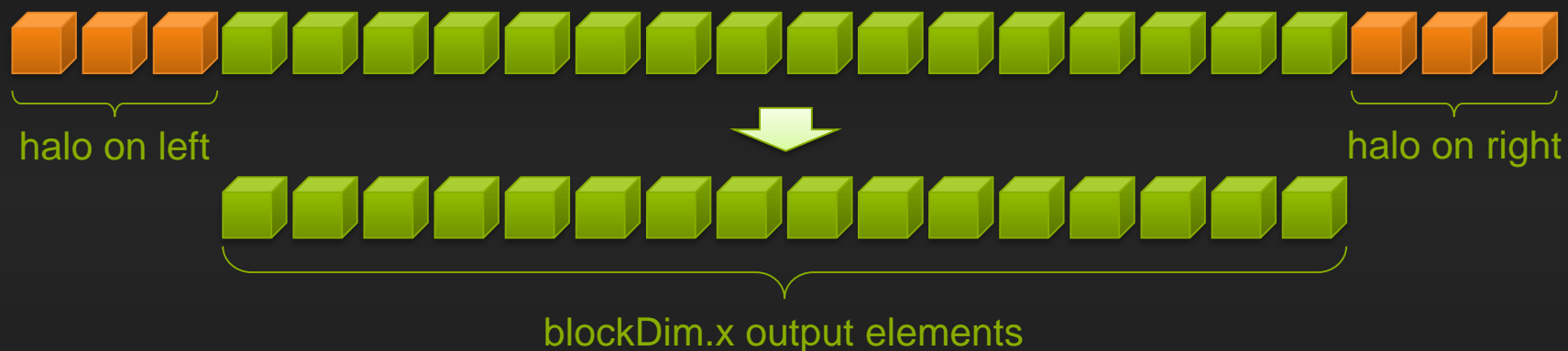


- Terminology: within a block, threads share data via **shared memory**
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

# Implementing With Shared Memory



- Cache data in shared memory
  - Read  $(\text{blockDim.x} + 2 * \text{radius})$  input elements from global memory to shared memory
  - Compute  $\text{blockDim.x}$  output elements
  - Write  $\text{blockDim.x}$  output elements to global memory
- Each block needs a **halo** of  $\text{radius}$  elements at each boundary



# \_\_syncthreads()

- `void __syncthreads ();`
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

# Stencil Kernel



```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
```

# Stencil Kernel



```
// Apply the stencil  
int result = 0;  
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)  
    result += temp[lindex + offset];  
  
// Store the result  
out[gindex] = result;  
}
```



## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# MANAGING THE DEVICE

# Coordinating Host & Device



- Kernel launches are **asynchronous**
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

`cudaMemcpy ()`

Blocks the CPU until the copy is complete

Copy begins when all preceding CUDA calls have completed

`cudaMemcpyAsync ()`

Asynchronous, does not block the CPU

`cudaDeviceSynchronize ()`

Blocks the CPU until all preceding CUDA calls have completed



# Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

# Device Management



- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
```

```
cudaSetDevice(int device)
```

```
cudaGetDevice(int *device)
```

```
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device

- A single thread can manage multiple devices

```
cudaSetDevice(i) to select current device
```

```
cudaMemcpy (...) for peer-to-peer copies†
```

<sup>†</sup> requires OS and device support



# EXECUTION CONFIGURATION

# Blocks Size Guidelines



- **Block size cannot be larger than 1024 threads**
  - $\text{Block size} = \text{blockDim.x} * \text{blockDim.y} * \text{blockDim.z}$
- **For performance**
  - Block size should be divisible by 32
  - Have at least 128 threads per block
- **Tip:**
  - start with 128 threads per block and tune up by increments of 32 if necessary

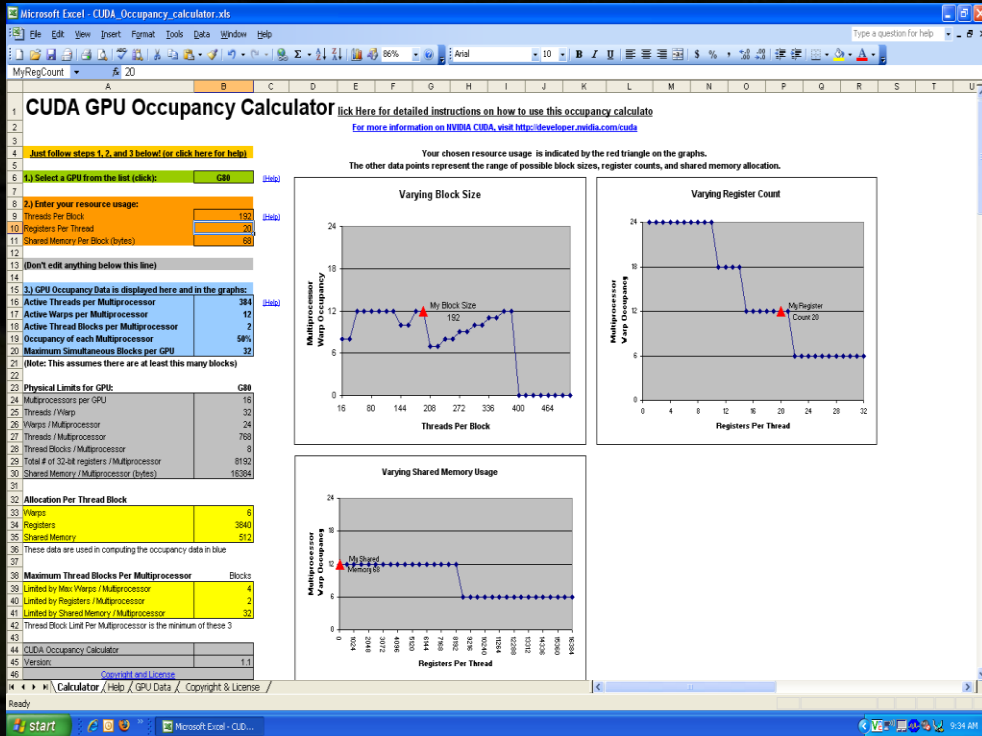
# GV100 Compute Capability 7.0



GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255*
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
# of Registers to FP32 Cores Ratio	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB

Compute Capabilities and SM limits of comparable Kepler, Maxwell, Pascal and Volta GPUs. (\*The per-thread program counter (PC) that forms part of the improved SIMT model typically requires two of the register slots per thread.)

# Occupancy Calculator



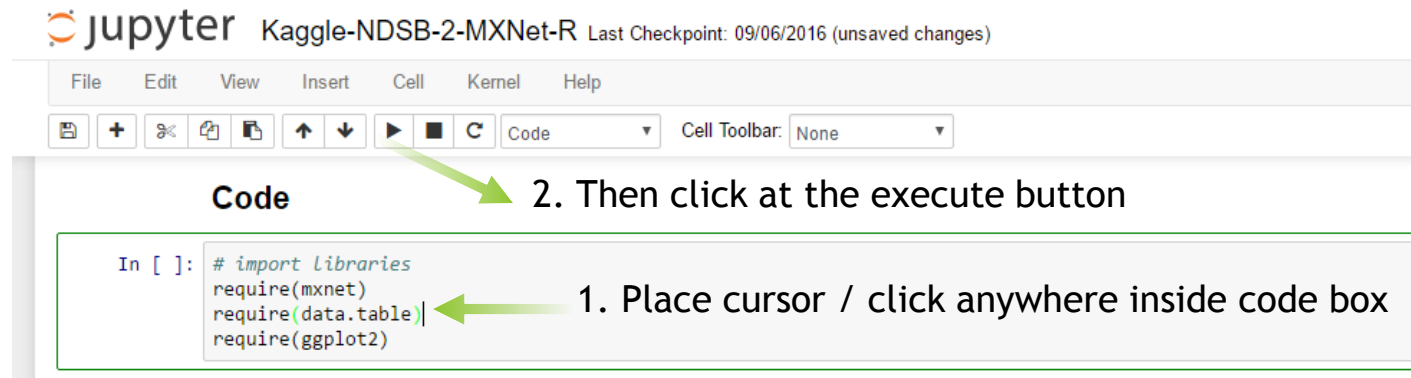
$$\text{occupancy} = \frac{\text{blocks per SM} \times \text{threads per block}}{\text{maximum threads per SM}}$$

- Occupancy calculator shows trade-offs between thread count, register use, shared memory use
- Low occupancy is bad
- Increasing occupancy doesn't always help

# JUPYTER NOTEBOOK

Key components are the grey boxes, that are executable cells. This is where you will be able to modify codes or use a command line.

The executable boxes are those that have “IN [ ]:” in front of the box. The blank space between brackets means that it hasn’t been run yet, and to run it you have to click in the box and press the play button (see below). Or you may also hit Control + Enter.



The screenshot shows the Jupyter Notebook interface. At the top, the title bar reads "jupyter Kaggle-NDSB-2-MXNet-R Last Checkpoint: 09/06/2016 (unsaved changes)". Below the title bar is a menu bar with "File", "Edit", "View", "Insert", "Cell", "Kernel", and "Help". Underneath the menu bar is a toolbar with various icons, including a play button (execute) and a square button (stop). A dropdown menu shows "Code" and "Cell Toolbar: None".


The main content area is a code cell titled "Code". It contains the following code:

```
In [ ]: # import Libraries
require(mxnet)
require(data.table)
require(ggplot2)
```

Two green arrows point to the code cell. One arrow points to the code text, with the instruction "1. Place cursor / click anywhere inside code box". The other arrow points to the play button in the toolbar, with the instruction "2. Then click at the execute button".

# JUPYTER NOTEBOOK

Once it is successfully running, you will see a number appear, such as the order as they have been executed.



```
In [1]: print "The answer should be three: " + str(1+2)
```

The answer should be three: 3

Let's execute the cell below to display information about the GPUs running on the server.

```
In [ ]: !nvidia-smi
```

## Introduction



Questions?

