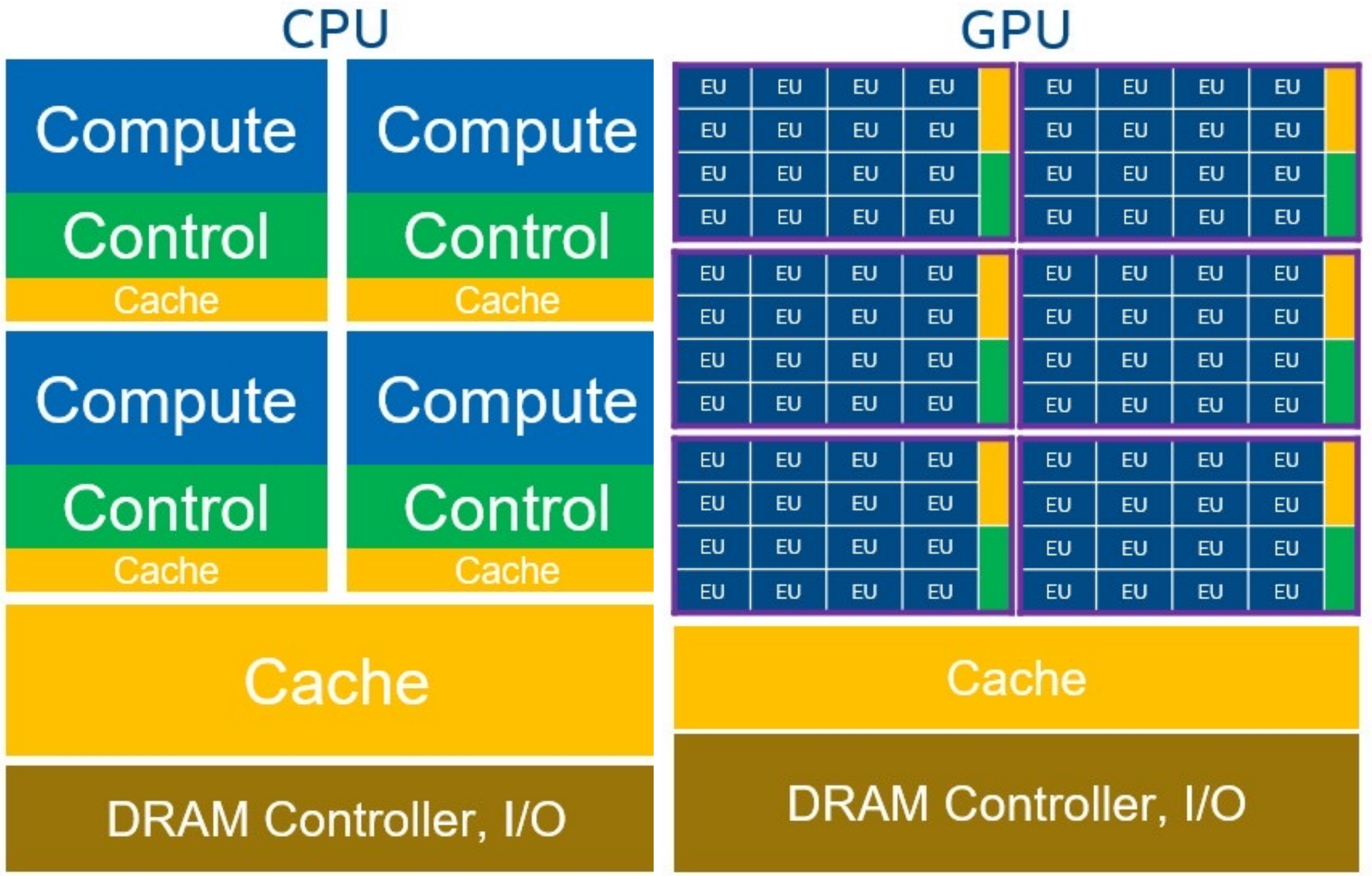
Introduction to CUDA

Ramakrishnan Kannan

Shruti Shivakumar

Motivated out of OLCF training seminar

CPU vs GPU



CPU

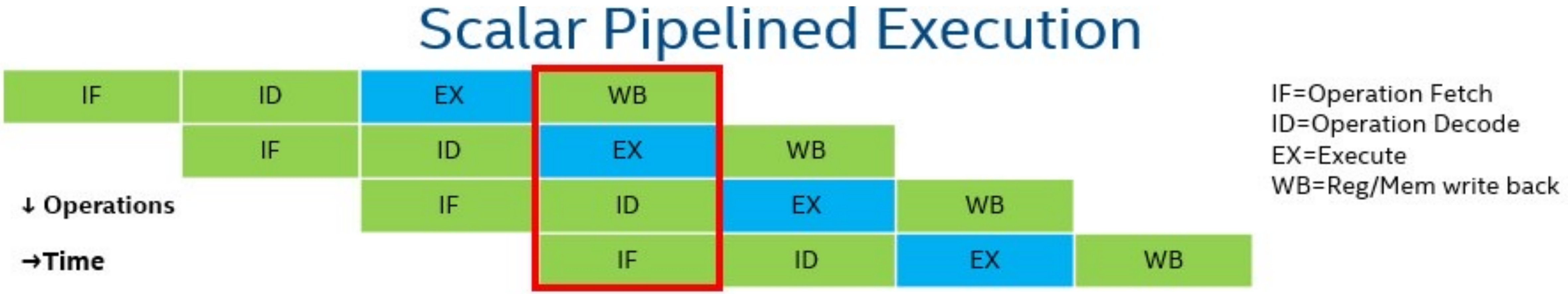
- can execute instructions, divided into stages, at the rate of up to one instruction per clock cycle (IPC) when there are no dependencies.
- Scalar Piplined Execution
- designed to process serial instructions efficiently.
- find instruction-level parallelism and execute multiple out-of-order instructions per clock cycle.

GPU

- SIMT == SIMD + Multithreading
- Is optimized for aggregate throughput across all cores, deemphasizing individual thread latency and performance.
- Efficiently processes vector data
- Dedicates more silicon space to compute and less to cache and control.

https://www.intel.com/content/www/us/en/developer/articles/technical/comparing-cpus-gpus-and-fpgas-for-oneapi.html#gs.npufd2

CPU Vs GPU Demo



CPU Advantages

- Out-of-order superscalar execution
- Sophisticated control to extract tremendous instruction level-parallelism
- Accurate branch prediction
- Automatic parallelism on sequential code
- Large number of supported instructions
- Lower latency when compared to offload acceleration
- Sequential code execution results in ease-of-development

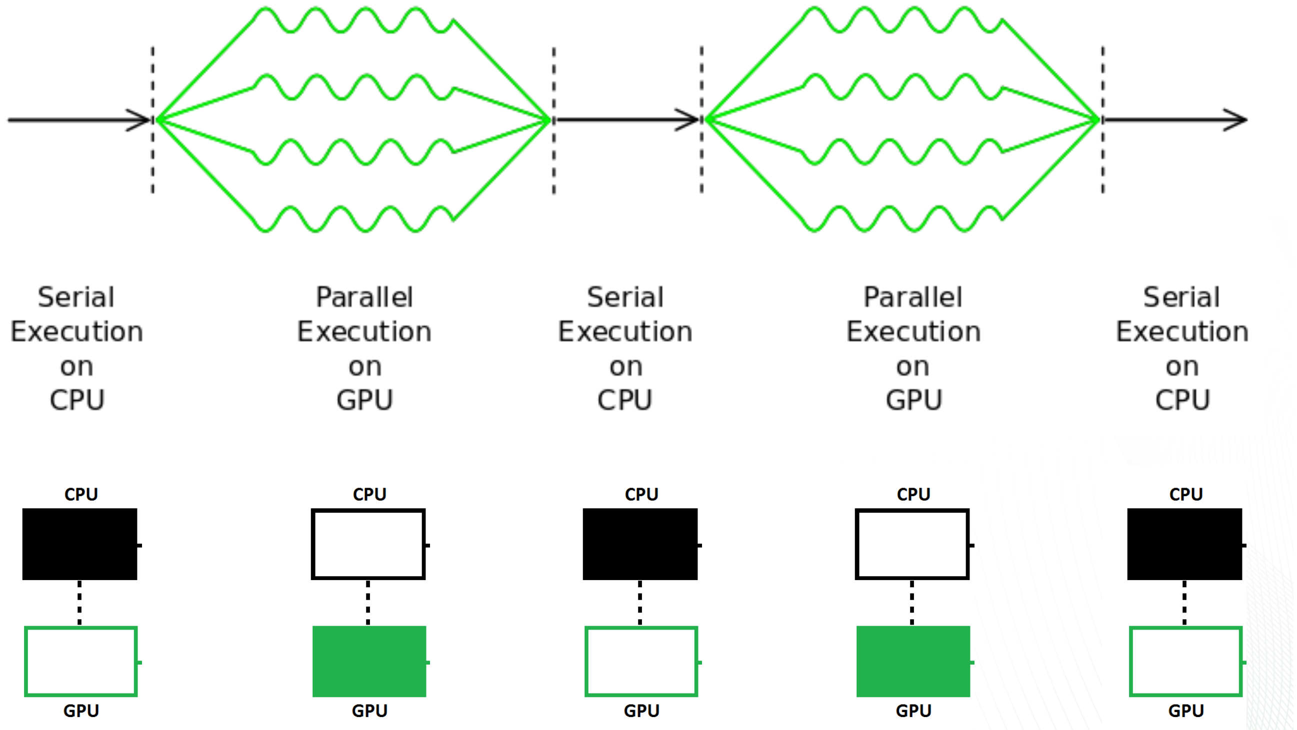
GPU Advantages:

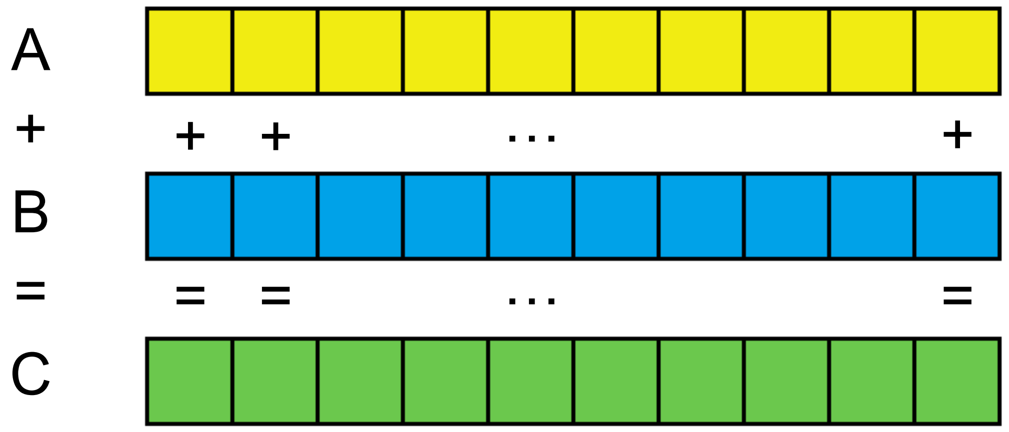
- Massively parallel, up to thousands of small and efficient SIMD cores/EUs
- Efficient execution of data-parallel code
- High dynamic random-access memory (DRAM) bandwidth

https://www.youtube.com/watch?v=-P28LKWTzrI

CUDA Programming Model

- Heterogenous Programming
  - program separated into serial regions (run on CPU) & parallel regions (run on GPU)
- Data Parallelism - Parallel regions consist of many calculations that can be executed independently

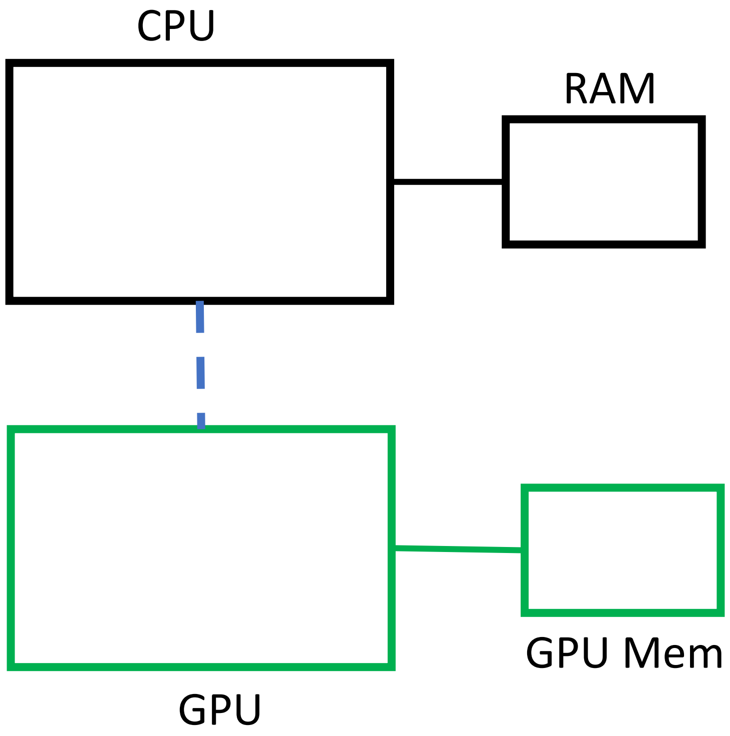




At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions (to C programming language)

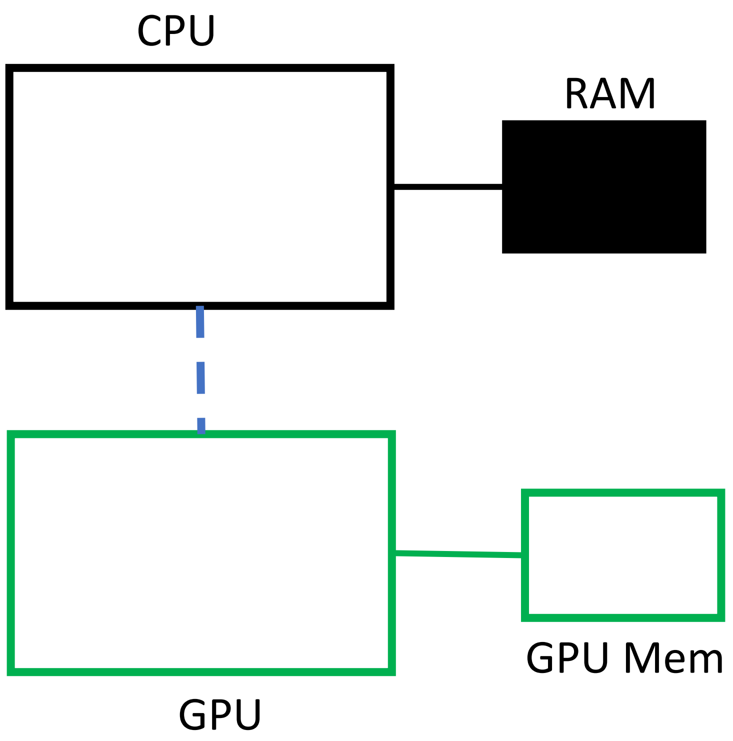
CUDA Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Host Memory
  // Free Device Memory
}
```



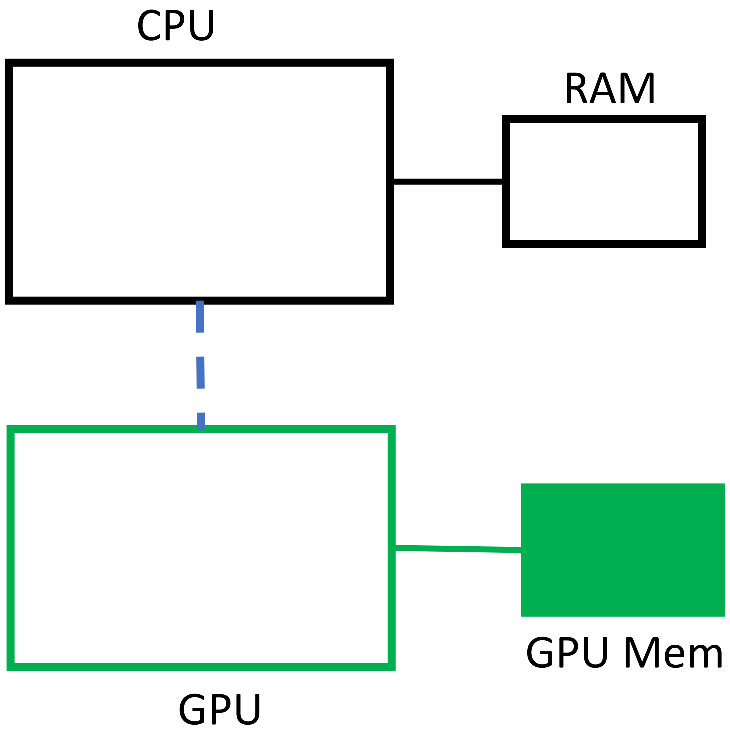
Step1

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Host Memory
  // Free Device Memory
}
```



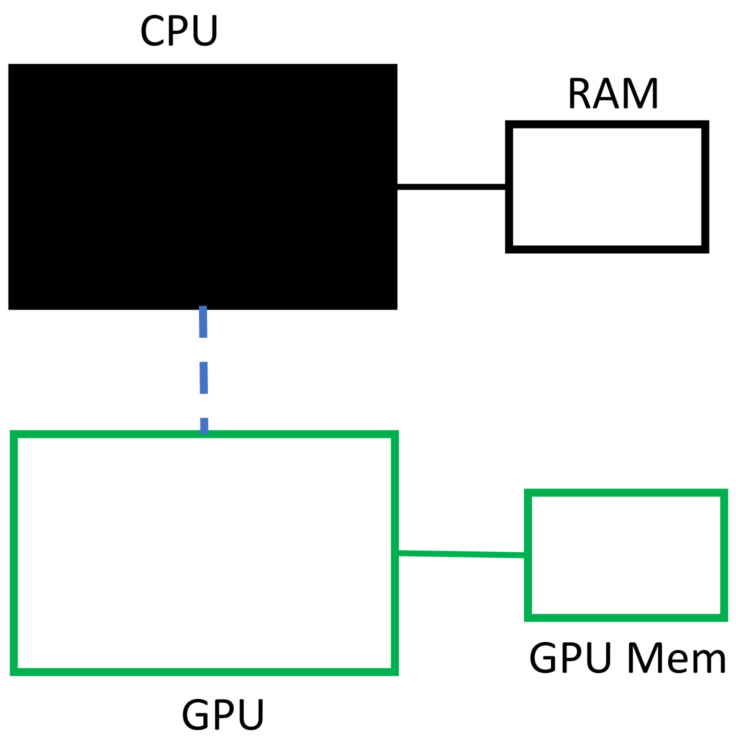
Step2

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Host Memory
  // Free Device Memory
}
```



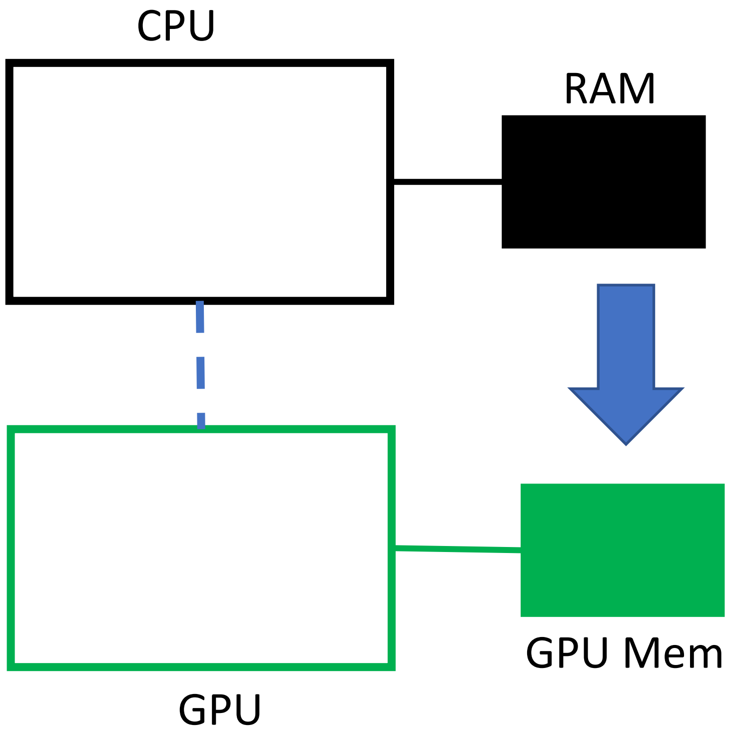
Step3

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Host Memory
  // Free Device Memory
}
```



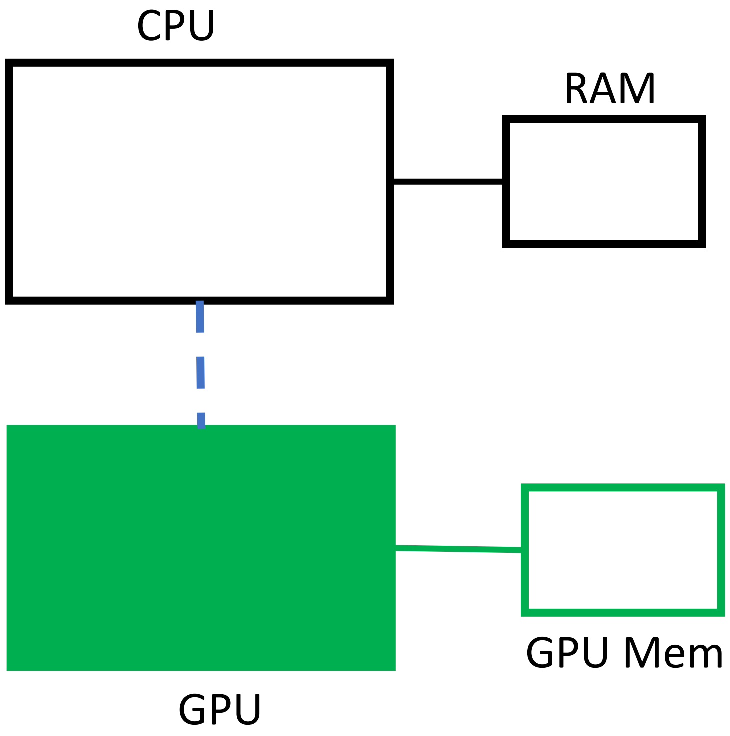
Step4

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Host Memory
  // Free Device Memory
}
```



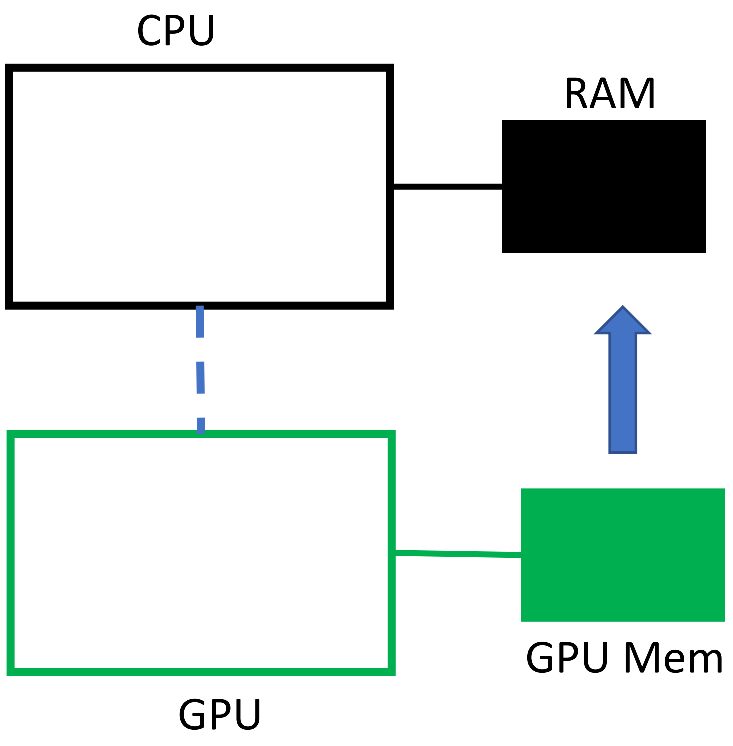
Step5

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Host Memory
  // Free Device Memory
}
```



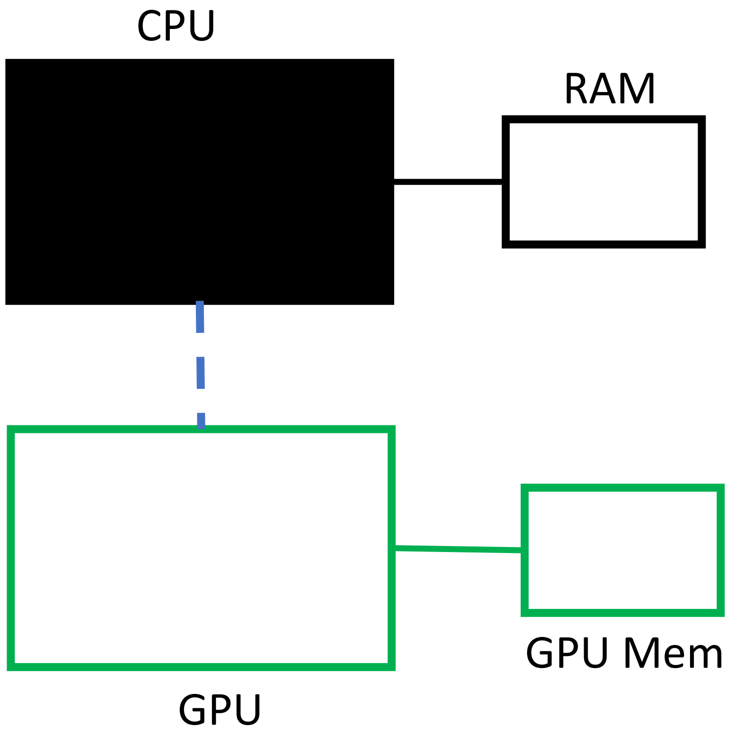
Step6

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Host Memory
  // Free Device Memory
}
```



Step7

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Host Memory
  // Free Device Memory
}
```



Real Driver Code

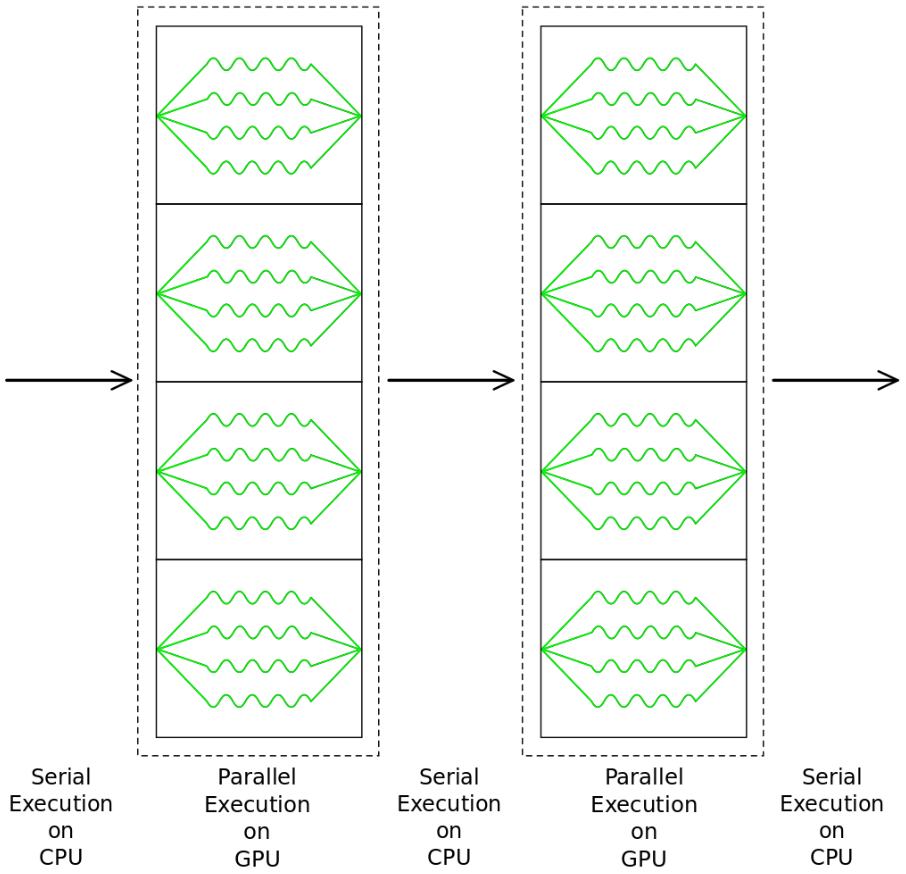
```
#define N 512
int main(void)
{
// host copies of a, b, c
int *a, *b, *c;
// device copies of a, b, c
int *d_a, *d_b, *d_c;
int size = N * sizeof(int);
// Alloc space for device copies
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
// Alloc space for host copies of
// a, b, c and setup input values
a = (int *)malloc(size);
random_ints(a, N);
b = (int *)malloc(size);
random_ints(b, N);
c = (int *)malloc(size);
// Copy inputs to device
cudaMemcpy(d_a, a, size,
cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size,
cudaMemcpyHostToDevice);
// Launch add() kernel on GPU with N
blocks
vector_addition<<<N,1>>>(d_a, d_b, d_c);
// Copy result back to host
cudaMemcpy(c, d_c, size,
cudaMemcpyDeviceToHost);
// Cleanup
free(a);
free(b);
free(c);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0;
}
```

CUDA Kernel

- When kernel is launched, a grid of threads are generated
- SIMD Code – Same code is executed by all threads
- Serial – CPU

```
for (int i=0; i<N; i++) {
  C[i] = A[i] + B[i];
}
```

- GPU – Parallel Code
  - C[i] = A[i] + B[i];



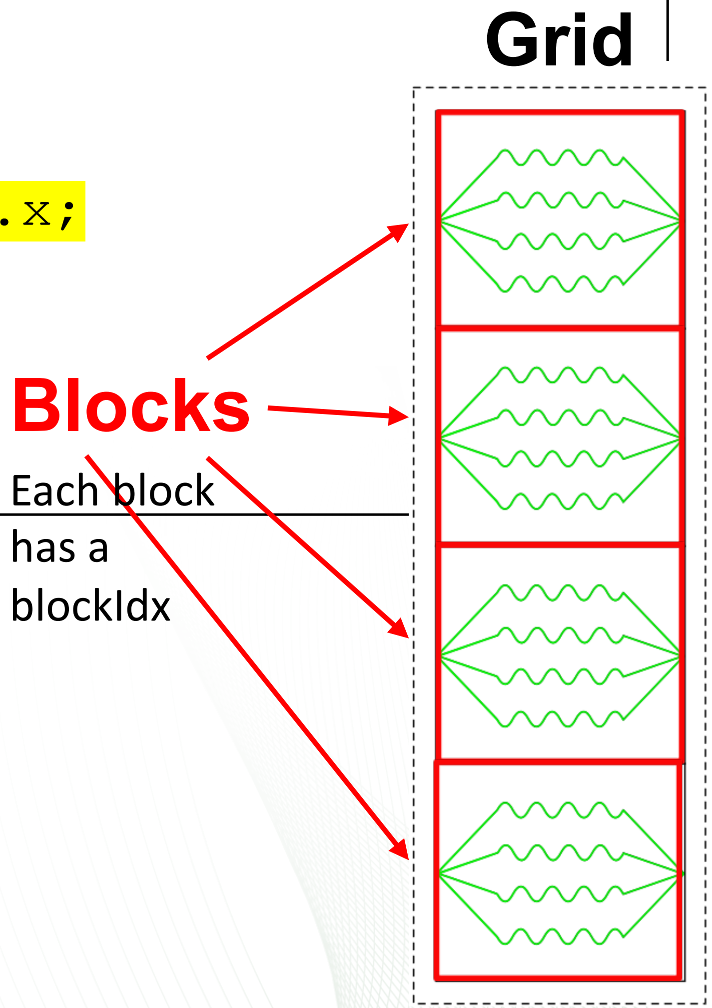
CUDA Kernel

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

- __global__
  - Indicates the function is a CUDA kernel function
  - called by the host and executed on the device.
- void
  - Kernel does not return anything.
- Kernel function arguments - int *a, int *b, int *c
  - a, b, c are pointers to device memory

CUDA Kernel - Blocks

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

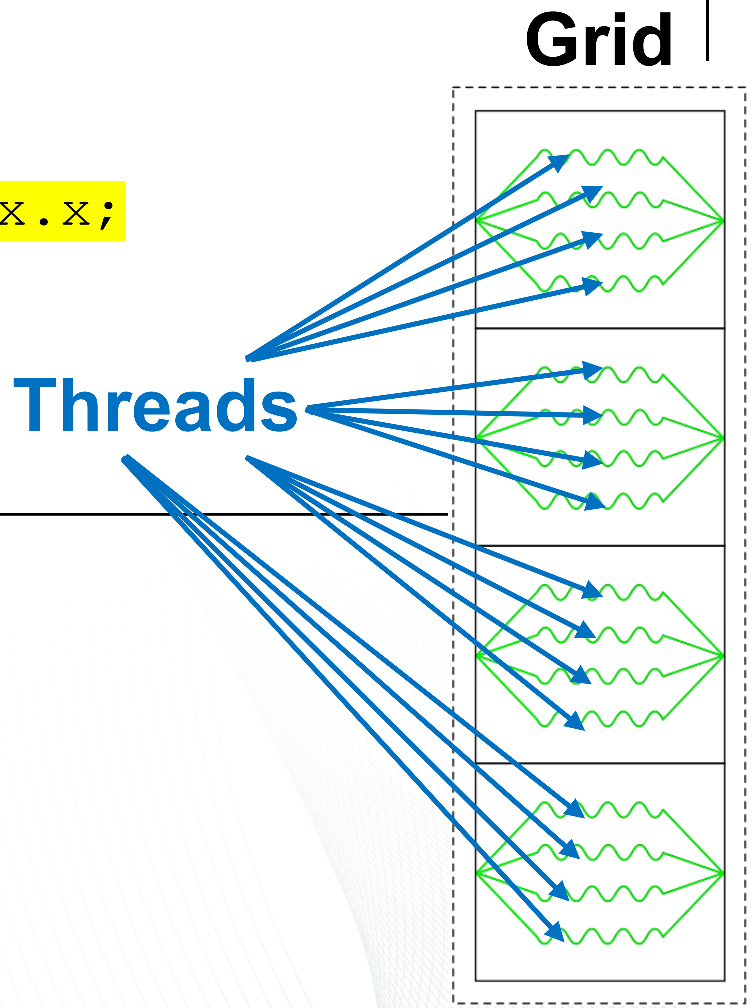



```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

This defines a unique thread id among all threads in a grid

CUDA Kernel - Threads

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

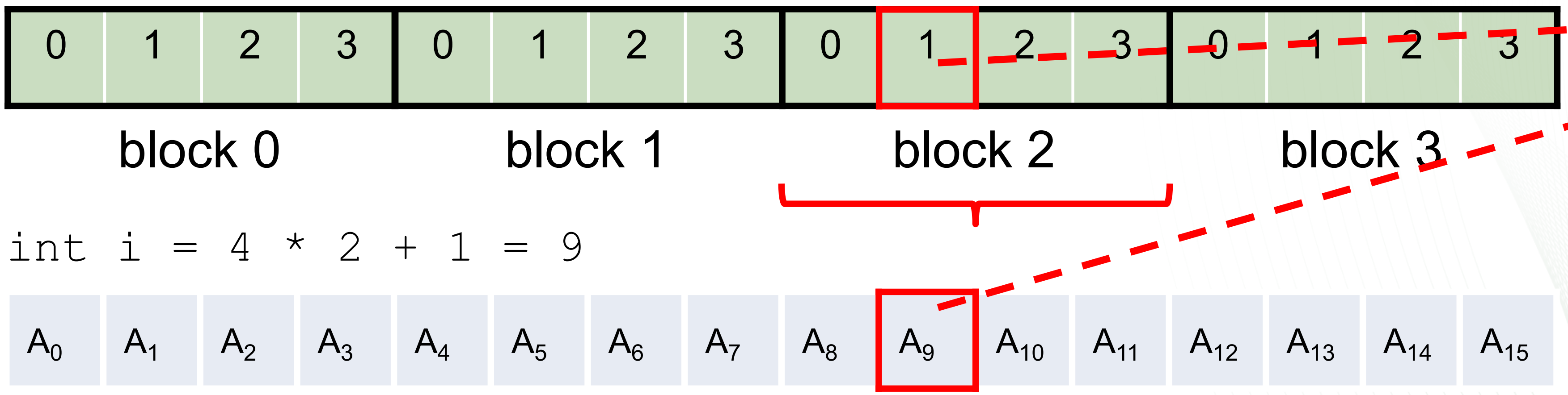


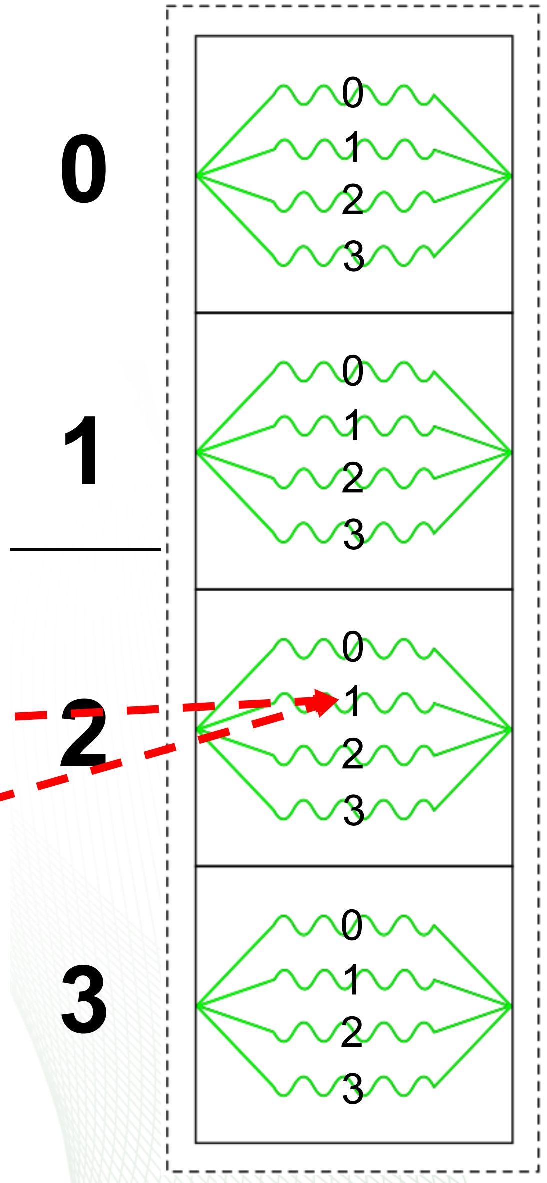
`int i = blockDim.x * blockIdx.x + threadIdx.x;`

This defines a unique thread id among all threads in a grid

Putting it all together

```
__global__ void vector_addition(int *a, int *b, int *c)
{
                (4)            (0–3)      (0–3)
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i<N) c[i] = a[i] + b[i];
}
```





Launching Kernel

- In general

```
kernel<<< blk_in_grid, thr_per_blk >>>(arg1, arg2, ...);
```

- Our specific problem

```
thr_per_blk = 128;
blk_in_grid = ceil( float(N) / thr_per_blk );
vec_add<<< blk_in_grid, thr_per_blk >>>(d_a, d_b, d_c);
```

Saxpy CUDA example

- Linear combination of two float arrays
- y = ax + y, where x and y are arrays of same length, and a is a scalar.

$$\begin{array}{c} a \cdot \\ a \cdot \\ \\ a \cdot \end{array} \begin{bmatrix} \boxed{x_1} \\ \boxed{x_2} \\ \vdots \\ \boxed{x_d} \end{bmatrix} + \begin{bmatrix} \boxed{y_1} \\ \boxed{y_2} \\ \vdots \\ \boxed{y_d} \end{bmatrix} = \begin{bmatrix} \boxed{a \cdot x_1 + y_1} \\ \boxed{a \cdot x_2 + y_2} \\ \vdots \\ \boxed{a \cdot x_d + y_d} \end{bmatrix}$$

```
for (i = 0; i < n; i++)
{
        y[i] = a*x[i] + y[i];
}
```

Sequential Code

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

CUDA Error Checking

- API Calls
  - ```
    cudaError_t err = cudaMalloc(&d_A, 8e9*bytes);
    if(err != cudaSuccess) printf("Error: %s\n",
    cudaGetErrorString(err));
    ```
- Kernels (check for synchronous and asynchronous errors)

```
add_vectors<<<blk_in_grid, thr_per_blk>>>(d_A, d_B, d_C, N);
// Kernel does not return an error, so get manually cudaError_t
errSync = cudaGetLastError();
if(errSync != cudaSuccess) printf("Error: %s\n",
cudaGetErrorString(errSync));
// After launch, control returns to the host, so errors can occur
at seemingly // random points later in the code. Calling
cudaDeviceSynchronize catches these // errors and allows you to
check them
cudaError_t errAsync = cudaDeviceSynchronize();
if(errAsync != cudaSuccess) printf("Error: %s\n",
cudaGetErrorString(errAsync));
```
```

Multi Dimension CUDA Grids

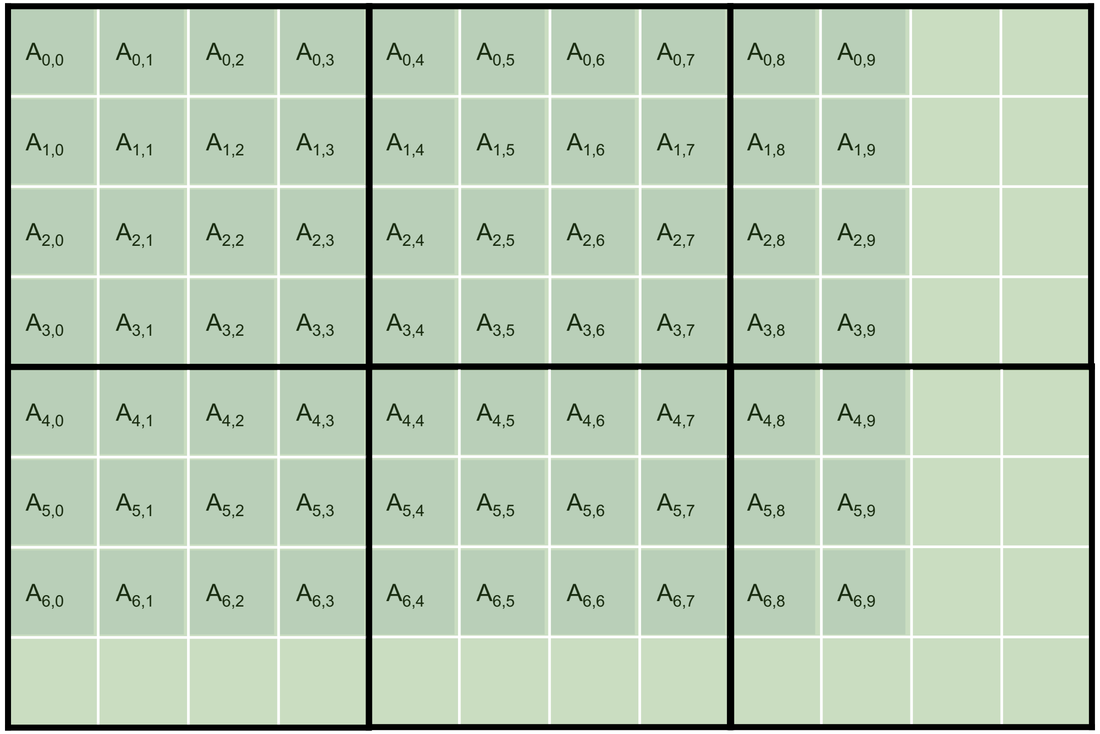
- 1D Example

```
thr_per_blk = 128
blk_in_grid = ceil( float(N) / thr_per_blk );
vec_add<<< blk_in_grid, thr_per_blk >>>(d_a, d_b, d_c);
```

- In General

```
dim3 threads_per_block( threads per block in x-dim,
threads per block in y-dim, threads per block in z-
dim);
dim3 blocks_in_grid( grid blocks in x-dim, grid blocks
in y-dim, grid blocks in z-dim );
```

CUDA Threads to 2D Array



Let M = 7 rows, N = 10 columns and a 4x4 blocks of threads...
To cover all elements in the array, we need 3 blocks in x-dim and 2 blocks in y-dim.

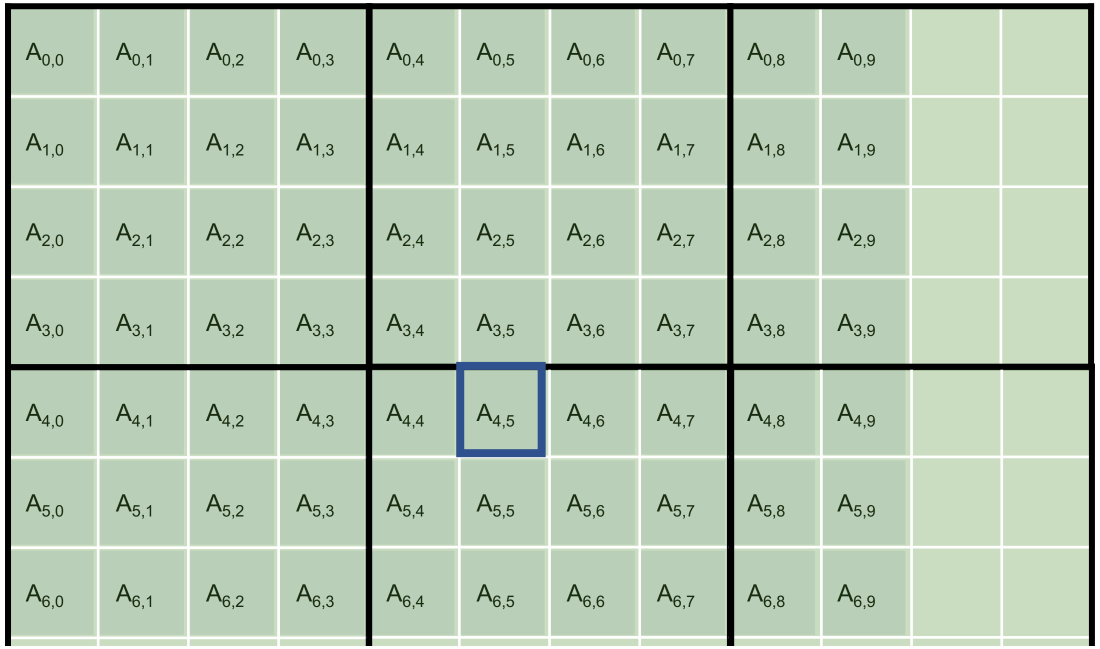
```
dim3 threads_per_block( 4, 4, 1 );
dim3 blocks_in_grid( ceil( float(N) / threads_per_block.x ), ceil( float(M) /
threads_per_block.y ) , 1 );
mat_add<<< blocks_in_grid, threads_per_block >>>(d_a, d_b, d_c);
```

Row Major ordering

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

CUDA Threads to 2D Array



Let M = 7 rows, N = 10 columns and a 4x4 blocks of threads...
To cover all elements in the array, we need 3 blocks in x-dim and 2 blocks in y-dim.

```
__global__ void add_matrices(int *a, int *b, int *c)
{
        int column = blockDim.x * blockIdx.x + threadIdx.x; // [0-11]
        int row = blockDim.y * blockIdx.y + threadIdx.y; //[0-7]
        if (row < M && column < N) //There is no row 7 and col 11.
        {
                int thread_id = row * N + column; //[0-69]
                c[thread_id] = a[thread_id] + b[thread_id];
        }
}
```

Row = 4
Column = 5
Thread_id = 4 * 10 + 5 = 45

Device Queries

- cudaDeviceProp -- C struct with many member variables
- cudaError_t cudaGetDeviceProperties(cudaDeviceProp *prop, int device) -- returns info about a particular GPU
- cudaError_t cudaGetDeviceCount(int *count) -- # of GPUs

Shared Memory

- Very fast on-chip memory
- Allocated per thread block
  - Allows data sharing between threads in the same block
  - Declared with __shared__ specifier
- Limited amount
  - 49152 B per block
- Must take care to avoid race conditions. For example...
  - Say, each thread writes the value 1 to one element of an array element.
  - Then one thread sums up the elements of the array
  - Synchronize with __syncthreads()
  - Acts as a barrier until all threads reach this point

Dot Product

$$c = \vec{a}.\vec{b} = \sum_{i=1}^{n} a_i \times b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

$$\begin{bmatrix} 1 & 3 & -5 \end{bmatrix} \begin{bmatrix} 4 \\ -2 \\ -1 \end{bmatrix} = 3$$

CUDA Dot Product – Single Block

```
#define N 1024
```

NOTE: We are only using 1 thread block here!

```
int threads_per_block = 1024;
int blocks_in_grid    = ceil( float(N) / threads_per_block);
```

```
__global__ void dot_prod(int *a, int *b, int *res)
{
    __shared__ int products[N];
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    products[id] = a[id] * b[id];

    __syncthreads();

    if(id == 0)
    {
        int sum_of_products = 0;
        for(int i=0; i<N; i++)
        {
            sum_of_products = sum_of_products + products[i];
        }

        *res = sum_of_products;
    }
```

Declare array of shared memory, shared within a grid block

Each thread calculates one element-wise product

Ensure all threads have reached this point before sum

Thread 0 sums all elements of the array and writes value to *res.

*Thanks OLCF

For additional material

- OLCF Training – 5 part training -- Robert Crovella (Nvidia)

Thank You