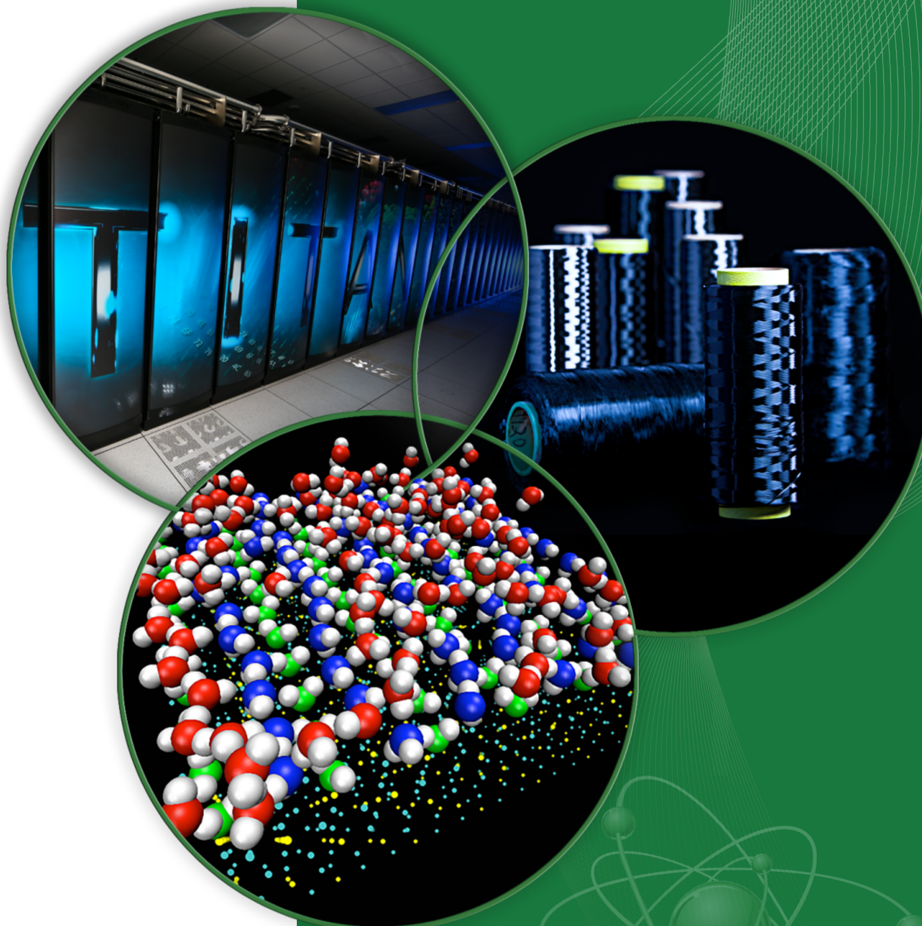
# Introduction to CUDA C/C++


Tom Papatheodore

HPC User Support Specialist/Programmer


June 19, 2017






OAK RIDGE National Laboratory
OAK RIDGE LEADERSHIP COMPUTING FACILITY

# Outline

- **Programming model for heterogeneous architectures**
- **Structure of a basic CUDA program**
  – Data transfers & kernel launches
  – Thread hierarchy
  – Run vector addition program
- **CUDA error checking**
- **Multi-D CUDA grids**
  – Run matrix addition program
  – Create matrix-vector multiply kernel
- **CUDA device queries**
- **Shared Memory**
  – Run dot product program
  – Create matrix-vector multiply kernel (with shared memory)

# Logging in to Chester

- `ssh username@home.ccs.ornl.gov`
- `ssh username@chester.ccs.ornl.gov`
- `cd $MEMBERWORK/trn001`

# Let's run something…

- `git clone https://github.com/tpapathe/intro_cuda.git`
- `module load cudatoolkit`
- `cd intro_cuda/vector_addition/`
- `nvcc vector_addition.cu -o run`
- `qsub submit.pbs`

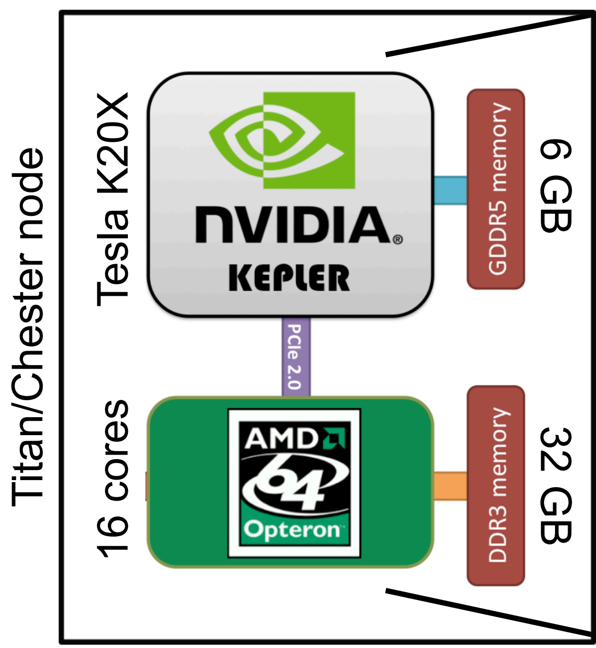
- `qstat -u username`
- `Check output file vec_add.o${JOB_ID}`
  - If you see `__SUCCESS__`, you have successfully run on GPU
  - If not, try again and/or ask for help

# CUDA Programming Model

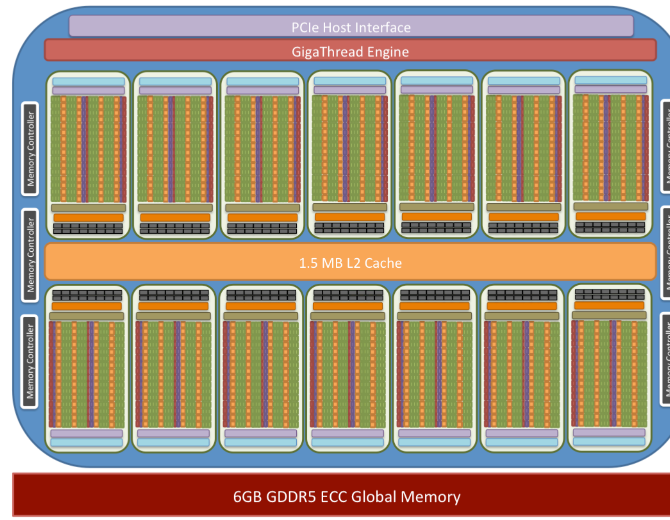
## Heterogeneous Architecture

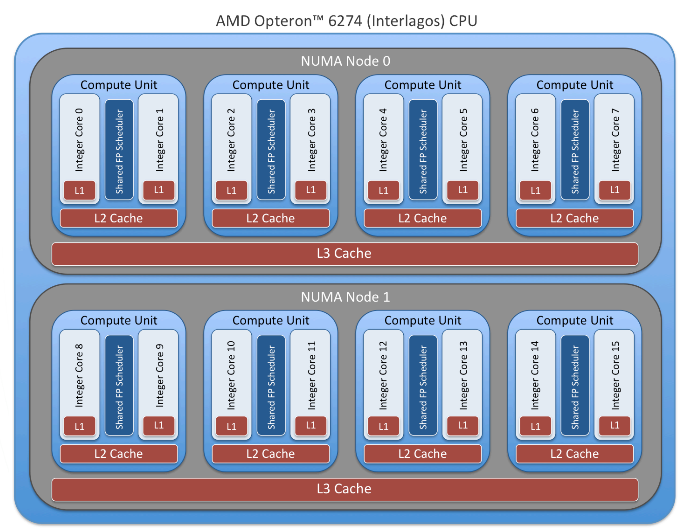
### GPU (device, accelerator)

- Thousands of compute cores




Each with its own memory




### CPU (host)

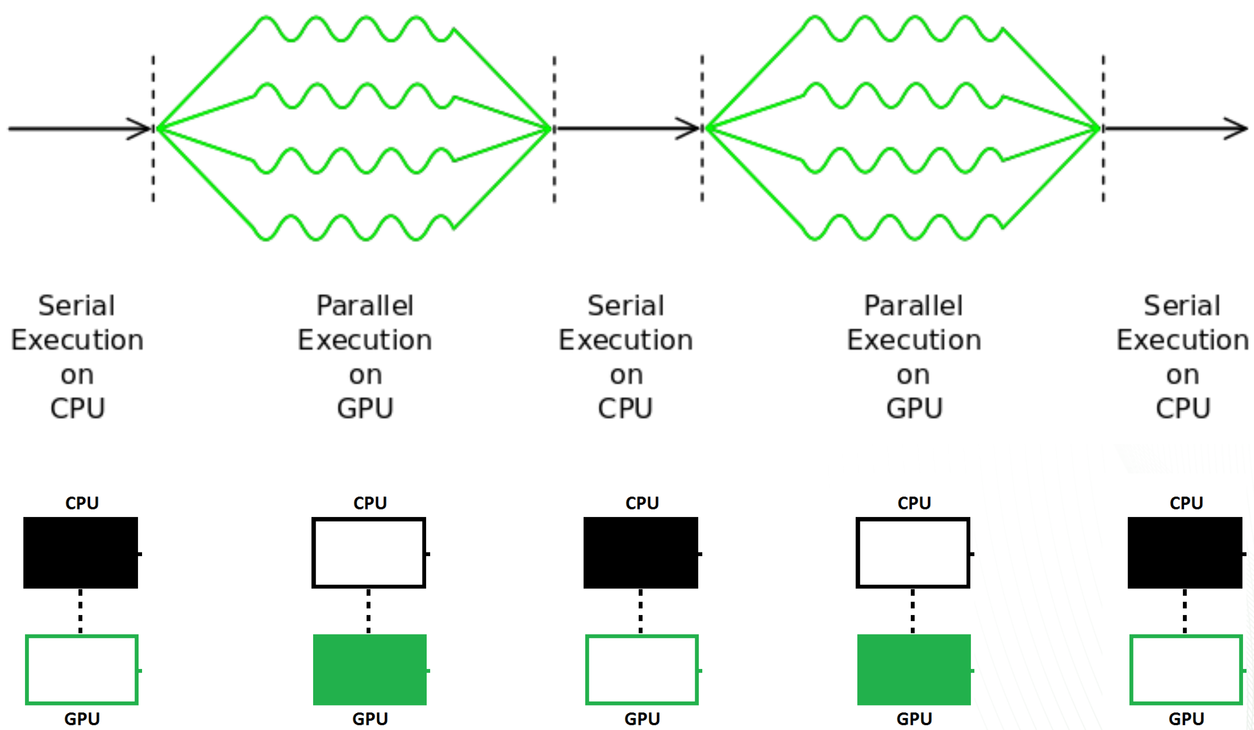
- Several compute cores



OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

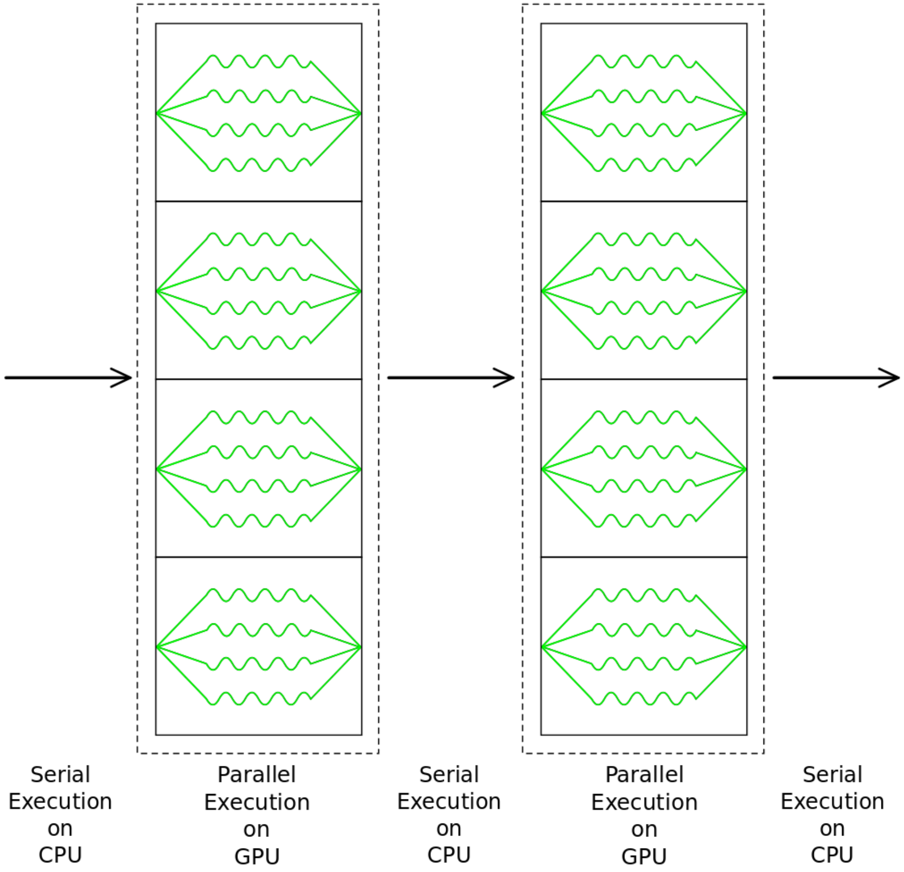
# CUDA Programming Model

- Heterogeneous Programming
  - program separated into serial regions (run on CPU) & parallel regions (run on GPU)



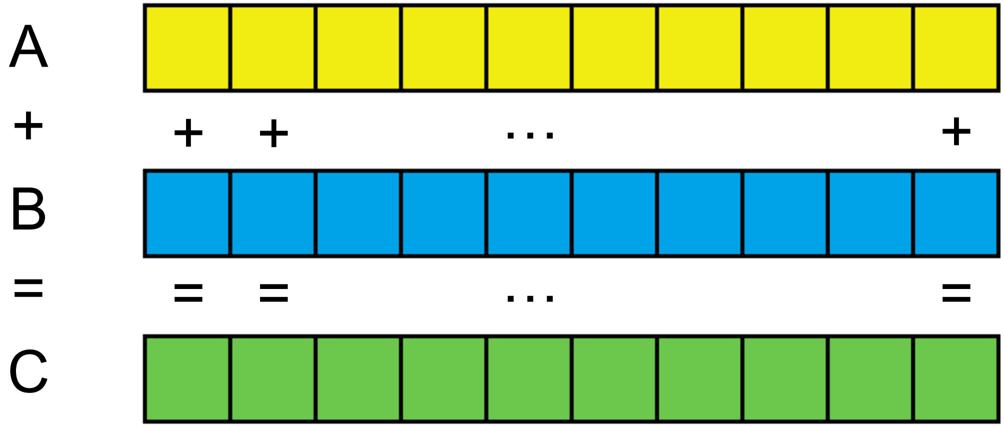
# CUDA Programming Model

- Heterogeneous Programming
  - program separated into serial regions (run on CPU) & parallel regions (run on GPU)



# CUDA Programming Model

- Parallel regions consist of many calculations that can be executed independently
  - Data Parallelism (e.g. vector addition)



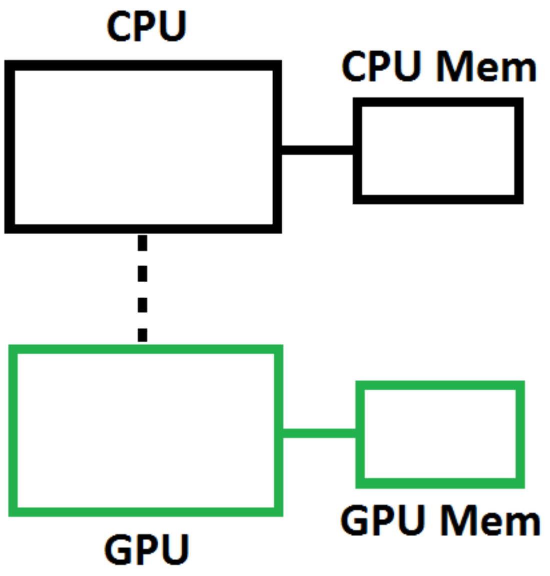
# CUDA Programming Model

From the CUDA Programming Guide:

*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)*

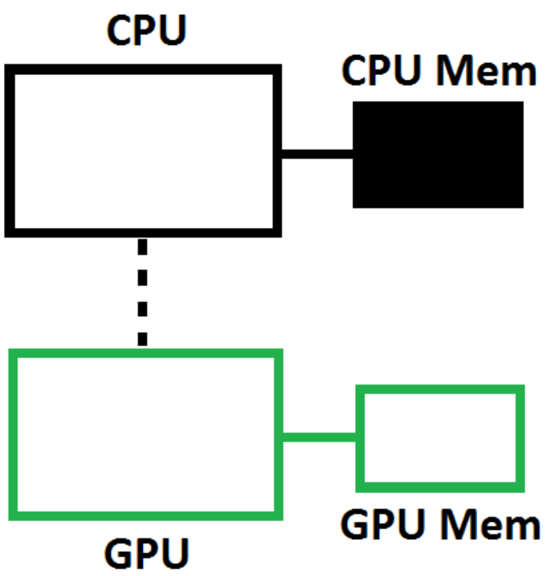
# A Basic CUDA Program Outline

```
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
}
```



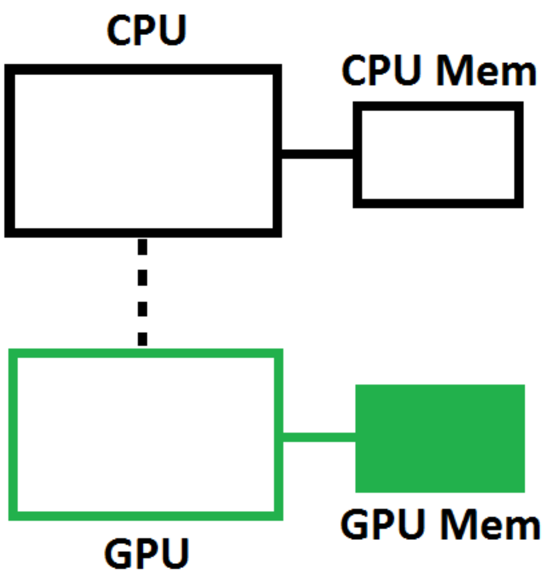
# A Basic CUDA Program Outline

```
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
}
```



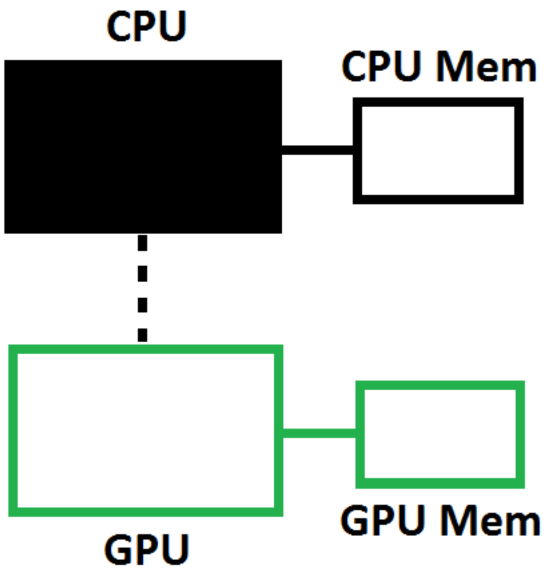
# A Basic CUDA Program Outline

```
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
}
```



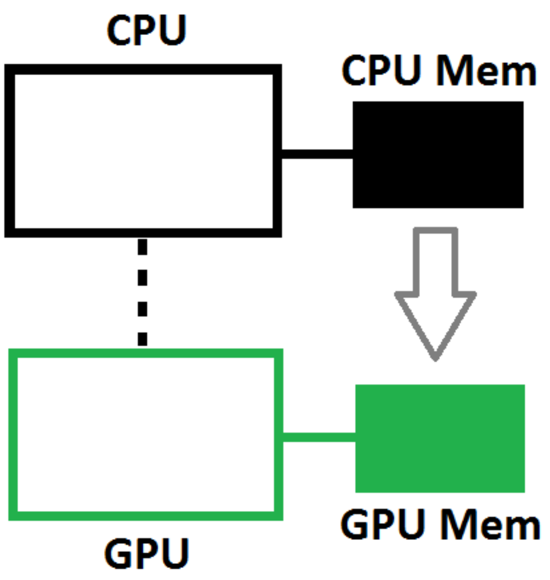
# A Basic CUDA Program Outline

```
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
}
```



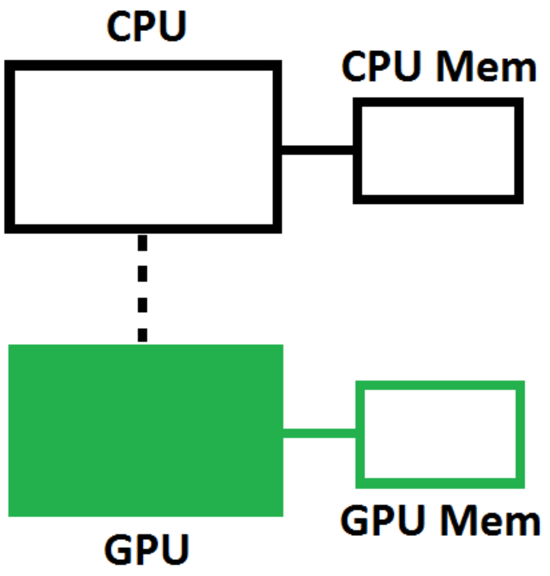
# A Basic CUDA Program Outline

```
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
}
```



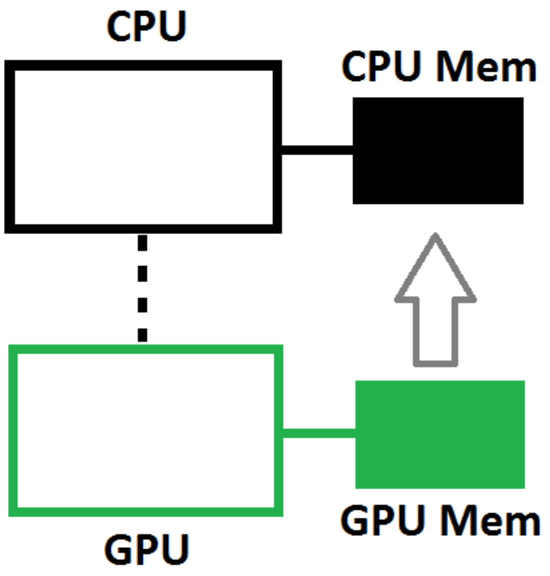
# A Basic CUDA Program Outline

```
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
}
```



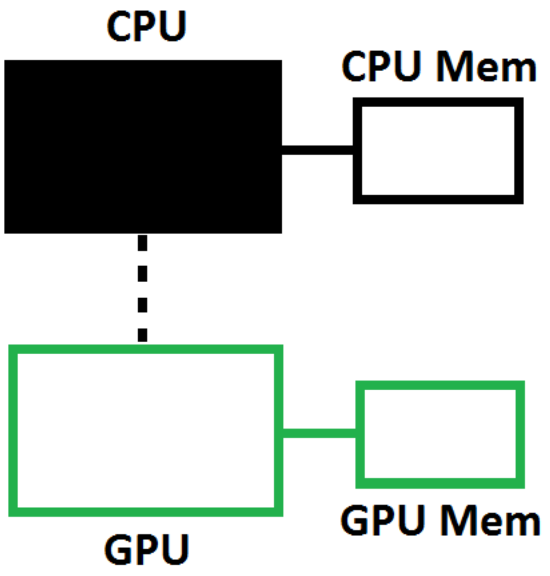
# A Basic CUDA Program Outline

```
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
}
```



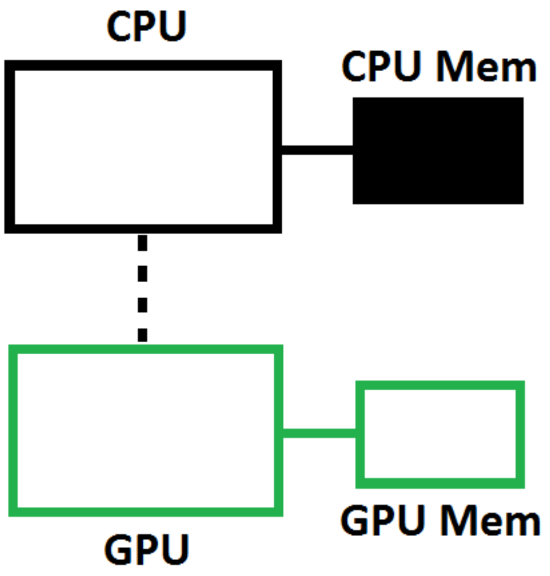
# A Basic CUDA Program Outline

```
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
}
```



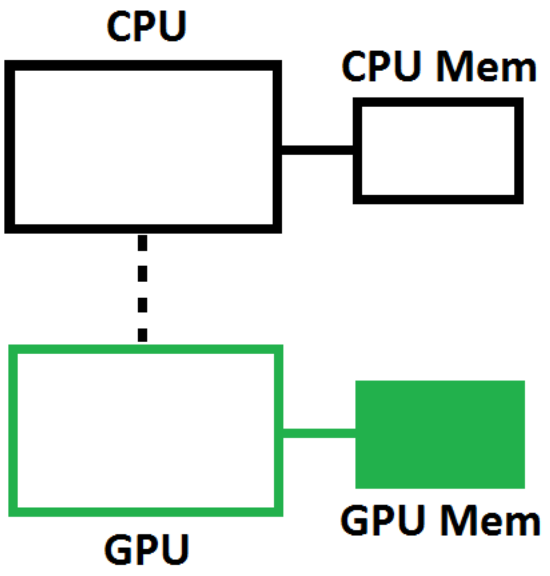
# A Basic CUDA Program Outline

```
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
}
```



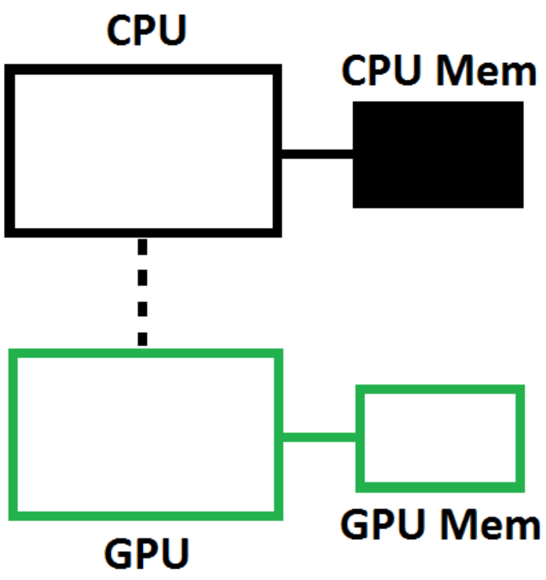
# A Basic CUDA Program Outline

```
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
}
```



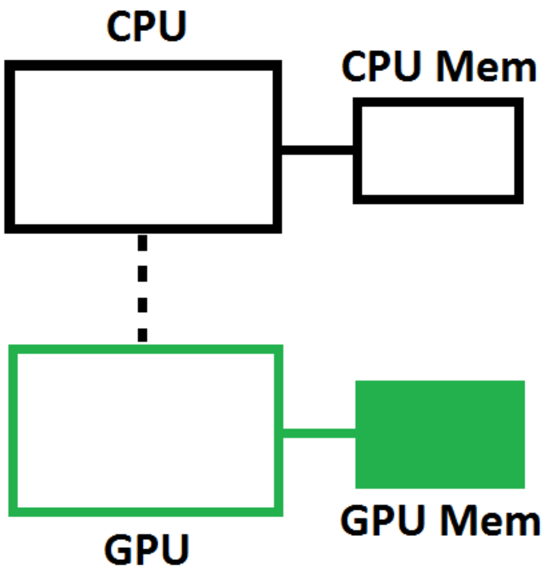
# A Basic CUDA Program Outline

```
int main(){
  // Allocate memory for array on host
  size_t bytes = N*sizeof(int);
  int *A = (int*)malloc(bytes);
  int *B = (int*)malloc(bytes);
  int *C = (int*)malloc(bytes);
  . . .
}
```



# A Basic CUDA Program Outline

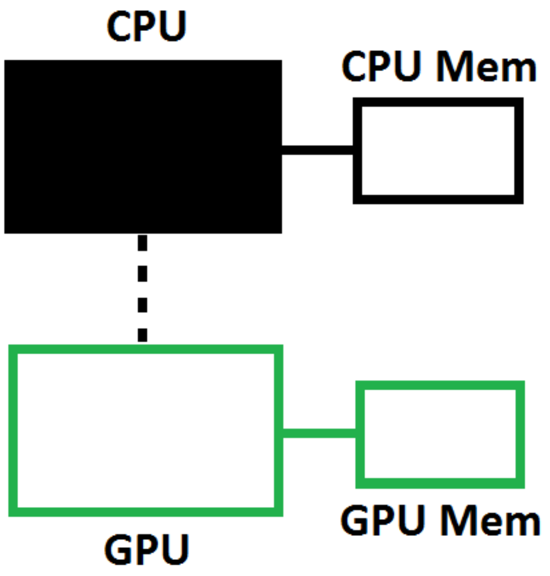
```
int main(){
   . . .
   // Allocate memory for array on device
   int *d_A, *d_B, *d_C;
   cudaMalloc(&d_A, bytes);
   cudaMalloc(&d_B, bytes);
   cudaMalloc(&d_C, bytes);
   . . .
}
```




```
cudaError_t cudaMalloc( void** devPtr, size_t size )
```

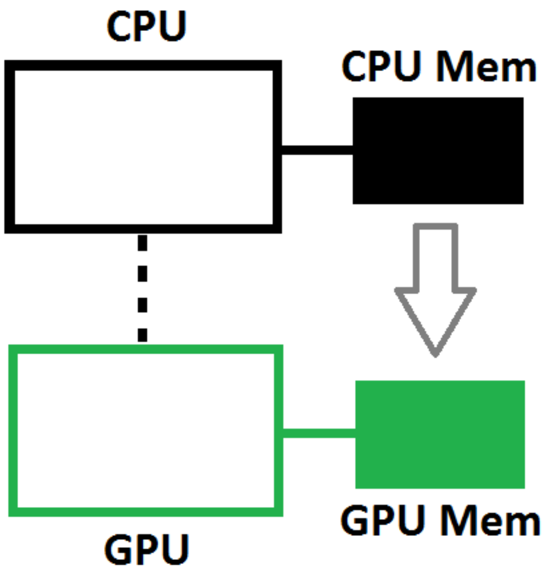
# A Basic CUDA Program Outline

```
int main(){
  . . .
  // Fill array on host
  for(int i=0; i<N; i++)
  {
    A[i] = 1;
    B[i] = 2;
    C[i] = 0;
  }
  . . .
}
```



# A Basic CUDA Program Outline

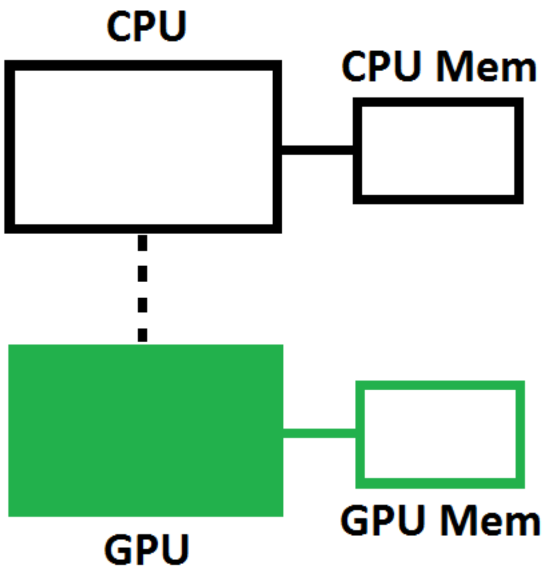
```
int main(){
  . . .
  // Copy data from host array to device array
  cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);
  . . .
}
```




```
cudaError_t cudaMemcpy( void* dst, const void* src, size_t count,
                        cudaMemcpyKind kind )
```

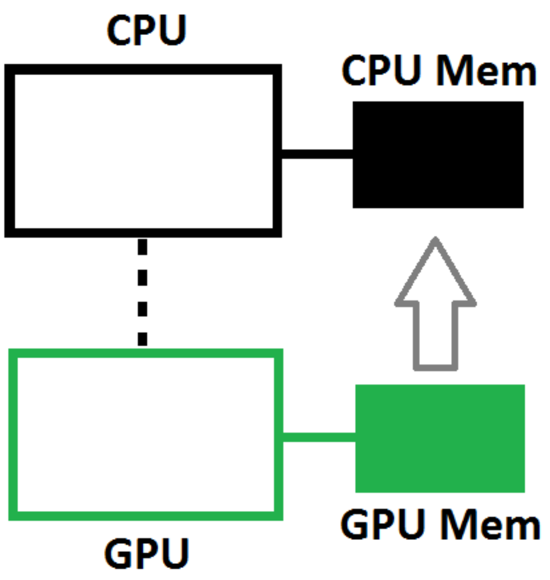
# A Basic CUDA Program Outline

```
int main(){
  . . .
  // Do something on device (e.g. vector addition)
    // We'll come back to this soon
  . . .
}
```



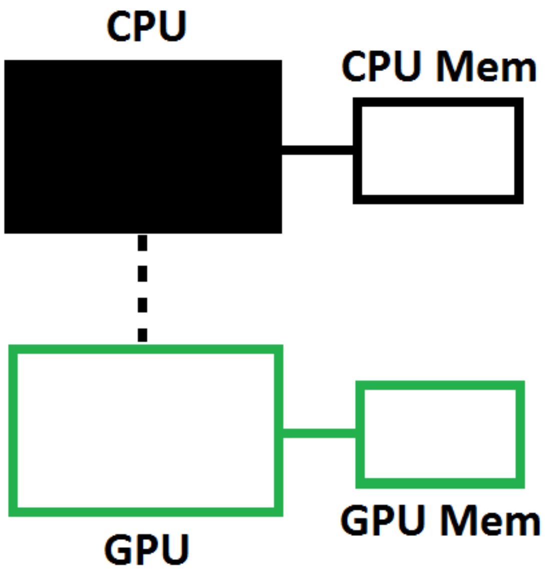
# A Basic CUDA Program Outline

```
int main(){

  . . .

  // Copy data from device array to host array

  cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);

  . . .

}
```



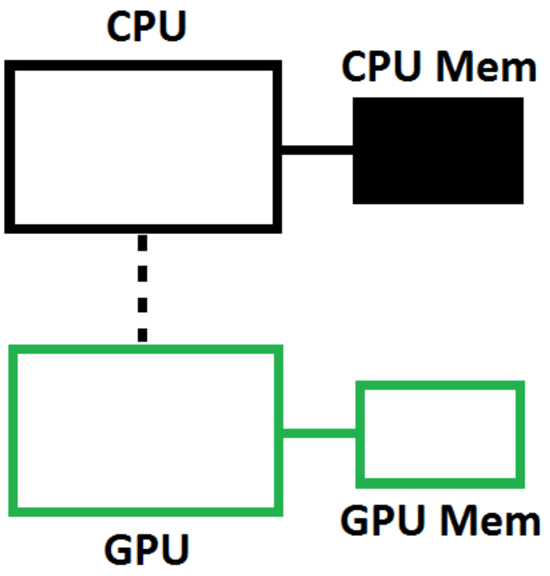
# A Basic CUDA Program Outline

```
int main(){
  . . .
  // Check data for correctness
  for (int i=0; i<N; i++)
  {
    if(C[i] != 3)
    {
      // Error - value of C[i] is not correct!
    }
  }
  . . .
}
```



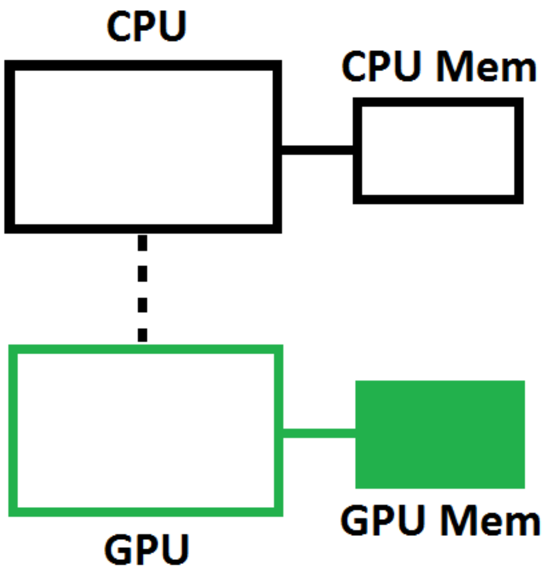
# A Basic CUDA Program Outline

```
int main(){
  . . .
  // Free Host Memory
  free(A);
  free(B);
  free(C);
  . . .
}
```



# A Basic CUDA Program Outline

```
int main(){
  . . .
  // Free Device Memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```




```
cudaError_t cudaFree( void* devPtr )
```

# CUDA Kernels

## Ok, so what about the kernel?

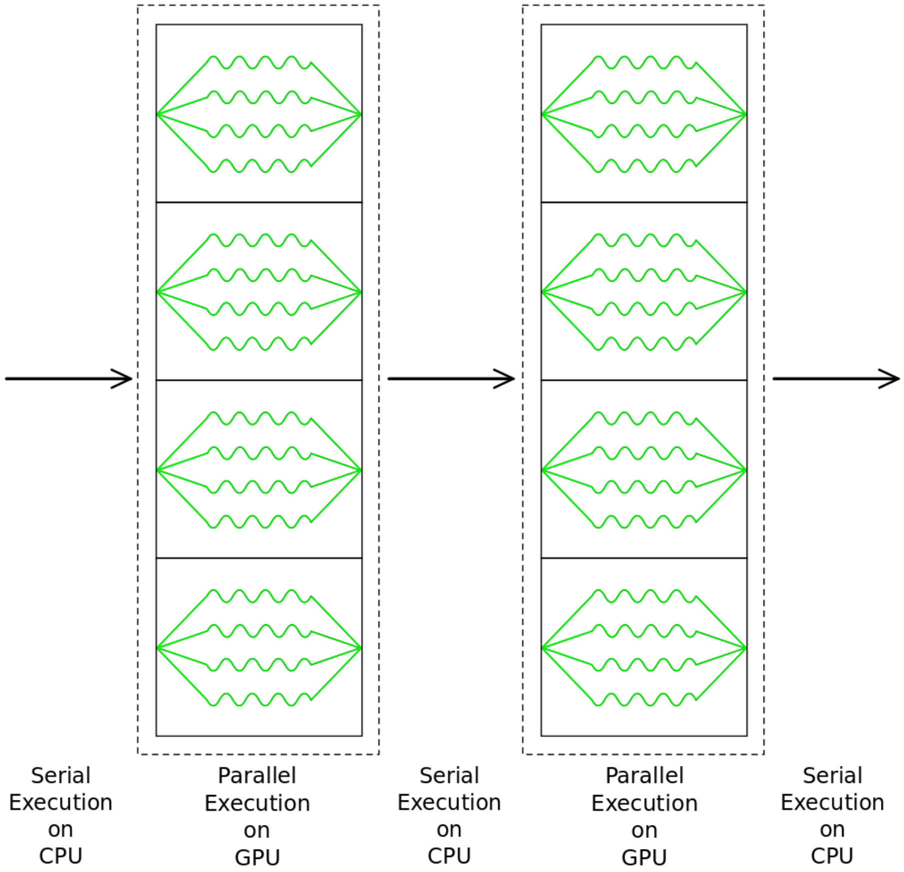
- How is it different from a normal function?
  - When kernel is launched, a grid of threads are generated
  - Same code is executed by all threads
    - Single-Program Multiple-Data (SPMD)

<u>Serial – CPU</u>

```
for (int i=0; i<N; i++){

  C[i] = A[i] + B[i];

}
```

<u>Parallel – GPU</u>

```
C[i] = A[i] + B[i];
```



# CUDA Kernels

Ok, so what about the kernel? What does it look like?

```
__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];
}
```

# CUDA Kernels

Ok, so what about the kernel? What does it look like?

```
__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.

# CUDA Kernels

Ok, so what about the kernel? What does it look like?

```
__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];
}
```

`__void__`

Kernel does not return anything.

# CUDA Kernels

Ok, so what about the kernel?  What does it look like?

```
__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 *a, int *b, int *c`

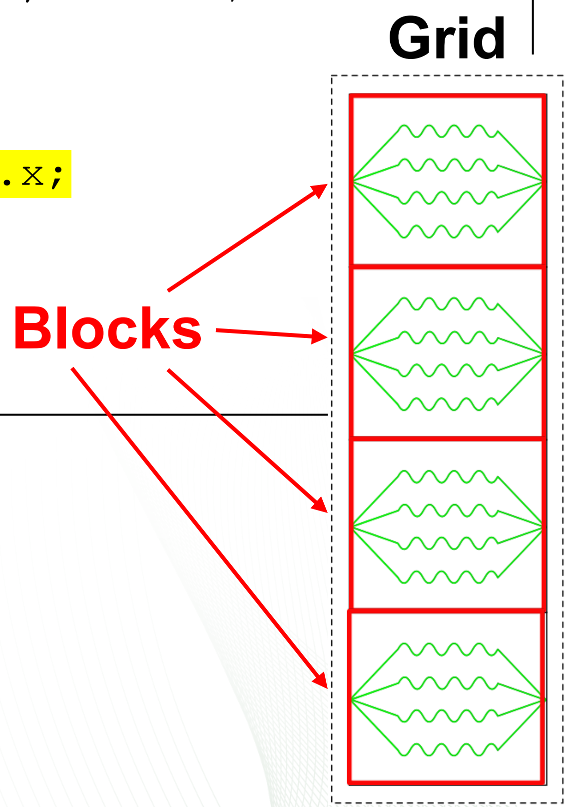
Kernel function arguments

- a, b, c are pointers to device memory

# CUDA Kernels

Ok, so what about the kernel? What does it look like?

```
__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];
}
```



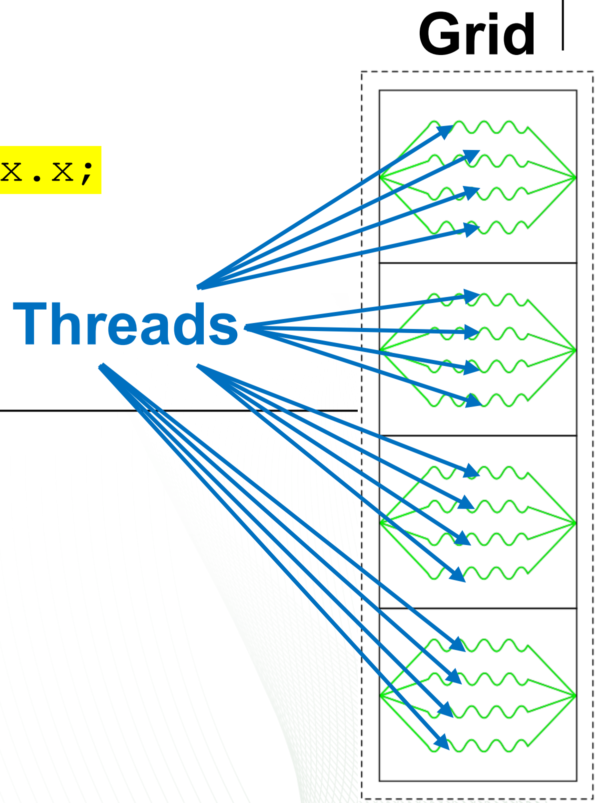

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

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

# CUDA Kernels

Ok, so what about the kernel?  What does it look like?

```
__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];
}
```



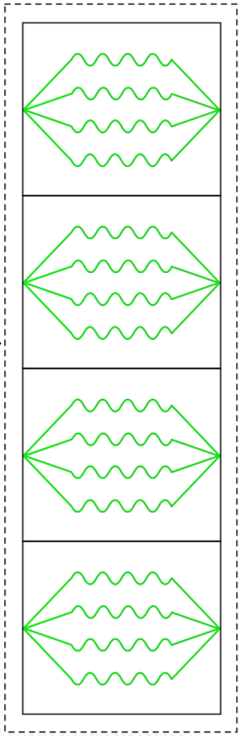

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

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

# CUDA Kernels

Ok, so what about the kernel? What does it look like?

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



`blockDim`

Gives the number of threads within each block (in the x-dimension in 1D case)

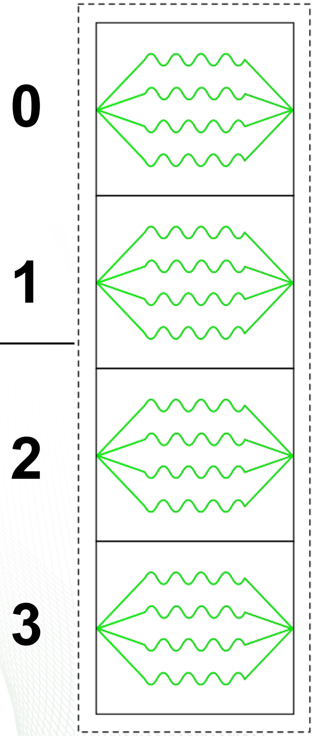
- E.g., 4 threads per block

# CUDA Kernels

Ok, so what about the kernel?  What does it look like?

(4) (0-3)

```
__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];
}
```



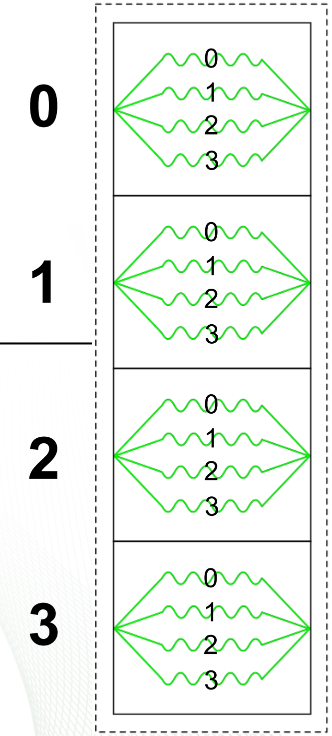

`blockIdx`

Specifies which block the thread belongs to (within the grid of blocks)

# CUDA Kernels

Ok, so what about the kernel?  What does it look like?

```
__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];
}
```



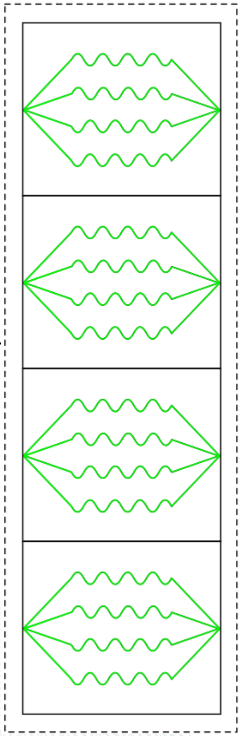

`threadIdx`

Specifies a local thread id within a thread block

# CUDA Kernels

Ok, so what about the kernel?  What does it look like?

```
__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];
}
```



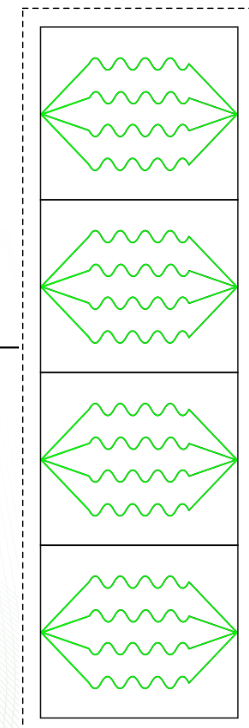
```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

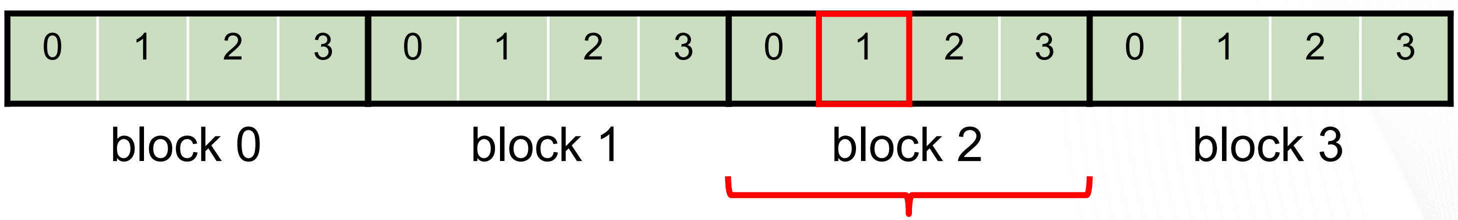
This defines a unique thread id among all threads in a grid

# CUDA Kernels

Ok, so what about the kernel? What does it look like?

```
__global__ void vector_addition(int *a, int *b, int *c)
{
                     (4)               (2)               (1)
   int i = blockDim.x * blockIdx.x + threadIdx.x;

   if (i<N) c[i] = a[i] + b[i];
}
```



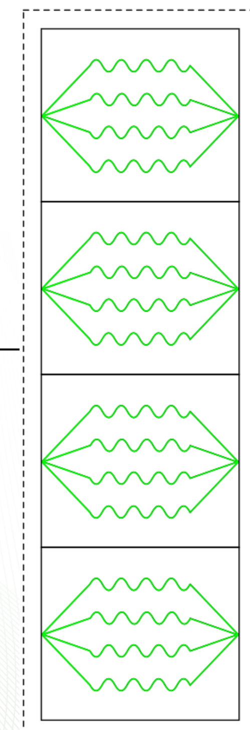



```
int i = 4 * 2 + 1 = 9
```

# CUDA Kernels

Ok, so what about the kernel?  What does it look like?

```
__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];
}
```



```
if (i<N)
```

Number of threads in the grid might be larger than number of elements in array.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ | $A_8$ | $A_9$ | $A_{10}$ | $A_{11}$ | $A_{12}$ | $A_{13}$ | $A_{14}$ | |

# CUDA Kernels

Ok, so what about the kernel?  What does it look like?

```
__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`

Local variables are private to each thread.

The loop was replaced by a grid of threads.

# CUDA Kernels

## Ok, so what about the kernel?

- How is it called (launched)?

In general

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

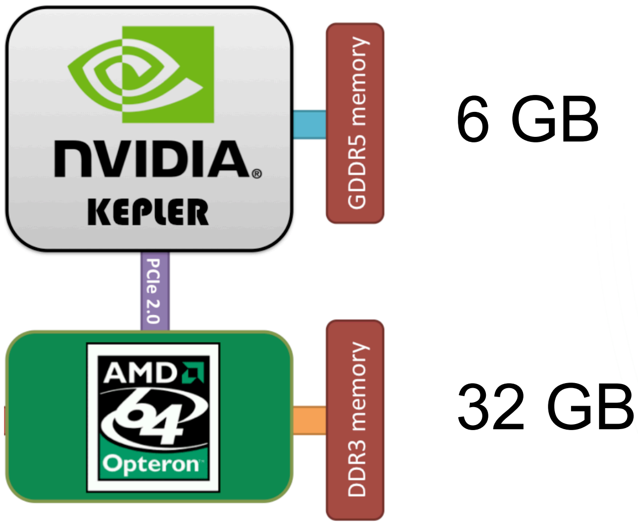
Our specific problem

```
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);
```

# Now let's run the vector_addition program again.

Run the vector_addition program `(nvcc vector_addition.cu -o run)`

- Change execution configuration parameters (i.e. change thr_per_blk)
  - What happens if you make thr_per_blk too large?
- Change thr_per_blk back to a value <= 1024 and change the size of d_A
  - `e.g., cudaMalloc(&d_A, 8e9*bytes);`



# 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-D CUDA Grids

# Multi-D CUDA Grids

In previous 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 );
```

**dim3** is c struct with member variables x, y, z.

# Multi-D CUDA Grids

In previous 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);
```

So we could have used

```
dim3 threads_per_block( 128, 1, 1 );
```

**dim3** is c struct with member variables x, y, z.

```
dim3 blocks_in_grid( ceil( float(N) / threads_per_block.x ), 1, 1 );

vec_add<<< blocks_in_grid, threads_per_block >>>(d_a, d_b, d_c);
```

# Map CUDA threads to 2D array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $A_{0,0}$ | $A_{0,1}$ | $A_{0,2}$ | $A_{0,3}$ | $A_{0,4}$ | $A_{0,5}$ | $A_{0,6}$ | $A_{0,7}$ | $A_{0,8}$ | $A_{0,9}$ | | |
| $A_{1,0}$ | $A_{1,1}$ | $A_{1,2}$ | $A_{1,3}$ | $A_{1,4}$ | $A_{1,5}$ | $A_{1,6}$ | $A_{1,7}$ | $A_{1,8}$ | $A_{1,9}$ | | |
| $A_{2,0}$ | $A_{2,1}$ | $A_{2,2}$ | $A_{2,3}$ | $A_{2,4}$ | $A_{2,5}$ | $A_{2,6}$ | $A_{2,7}$ | $A_{2,8}$ | $A_{2,9}$ | | |
| $A_{3,0}$ | $A_{3,1}$ | $A_{3,2}$ | $A_{3,3}$ | $A_{3,4}$ | $A_{3,5}$ | $A_{3,6}$ | $A_{3,7}$ | $A_{3,8}$ | $A_{3,9}$ | | |
| $A_{4,0}$ | $A_{4,1}$ | $A_{4,2}$ | $A_{4,3}$ | $A_{4,4}$ | $A_{4,5}$ | $A_{4,6}$ | $A_{4,7}$ | $A_{4,8}$ | $A_{4,9}$ | | |
| $A_{5,0}$ | $A_{5,1}$ | $A_{5,2}$ | $A_{5,3}$ | $A_{5,4}$ | $A_{5,5}$ | $A_{5,6}$ | $A_{5,7}$ | $A_{5,8}$ | $A_{5,9}$ | | |
| $A_{6,0}$ | $A_{6,1}$ | $A_{6,2}$ | $A_{6,3}$ | $A_{6,4}$ | $A_{6,5}$ | $A_{6,6}$ | $A_{6,7}$ | $A_{6,8}$ | $A_{6,9}$ | | |
| | | | | | | | | | | | |

M = 7 rows
N = 10 columns

Assume a 4x4 blocks of threads…

Then 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 Order

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

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| … | | | | | | | | | | | | | | | | | | | | | … |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | |

# Row-Major Order

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

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

# Row-Major Order

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

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

# Row-Major Order

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

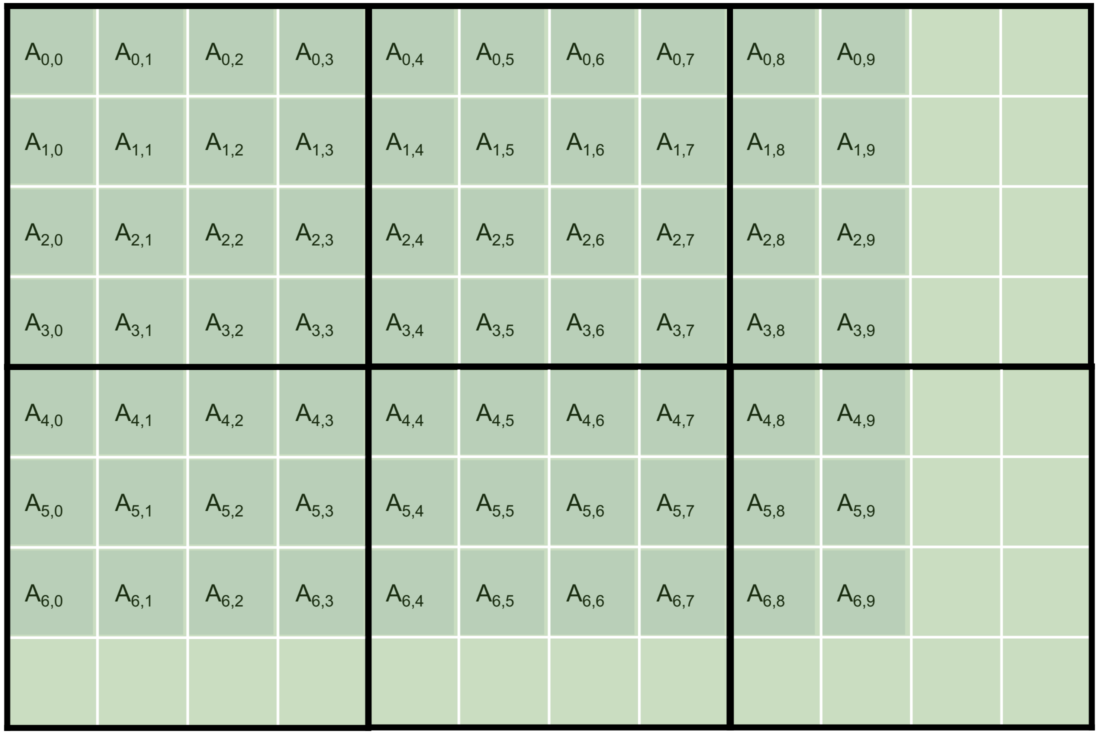
| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| … | $A_{0,0}$ | $A_{0,1}$ | $A_{0,2}$ | $A_{0,3}$ | $A_{0,4}$ | $A_{1,0}$ | $A_{1,1}$ | $A_{1,2}$ | $A_{1,3}$ | $A_{1,4}$ | $A_{2,0}$ | $A_{2,1}$ | $A_{2,2}$ | $A_{2,3}$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

# Row-Major Order

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

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

# Map CUDA threads to 2D array

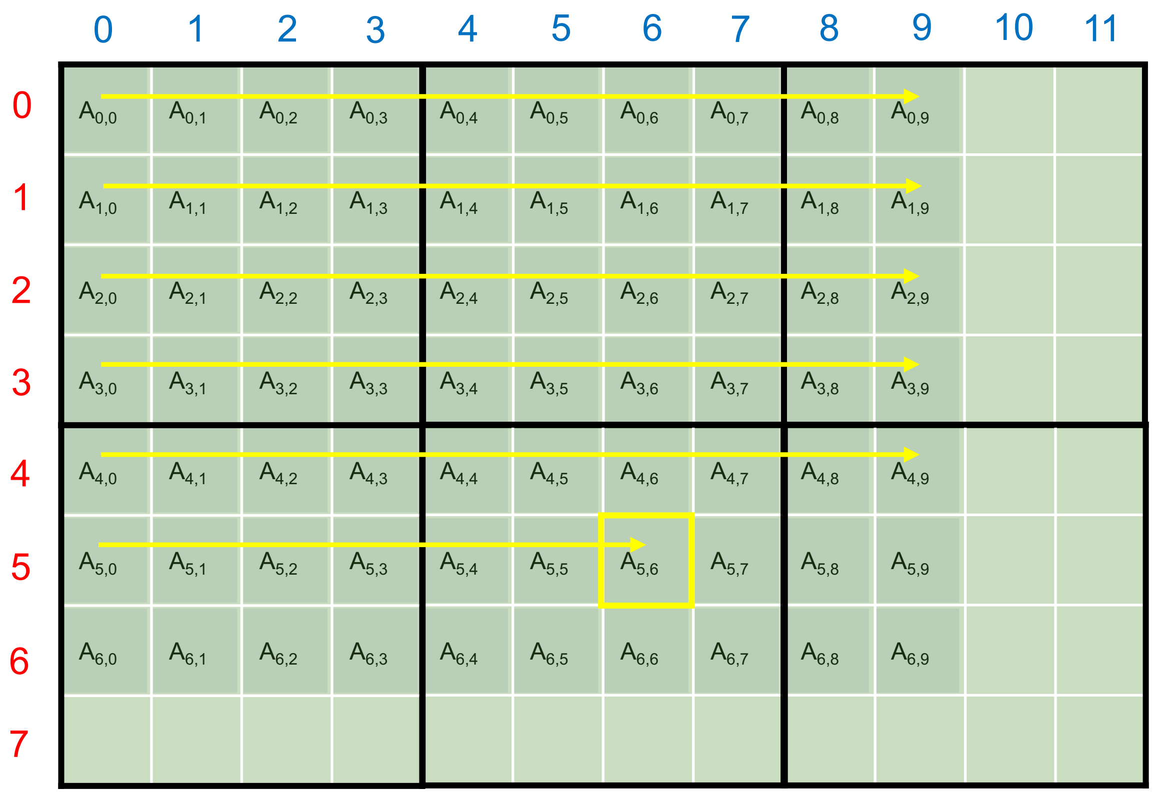



M = 7 rows
N = 10 columns

Assume 4x4 blocks of threads…

Then 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;
        int row    = blockDim.y * blockIdx.y + threadIdx.y;
        if (row < M && column < N){
                int thread_id = row * N + column;
                c[thread_id] = a[thread_id] + b[thread_id];
        }
}
```




M = 7 rows
N = 10 columns

Assume 4x4 blocks of threads…

Then 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){
                int thread_id = row * N + column;                (0 – 69)
                c[thread_id] = a[thread_id] + b[thread_id];
        }
}
```

Ex: What element of the array does the highlighted thread correspond to?

**thread_id = row * N + column**

**= 5 * 10 + 6 = 56**

# Now let's run the matrix_addition program.

Run the matrix_addition program

- Change execution configuration parameters
  - `threads_per_block( 16, 16, 1 );`

- NOTE: you cannot exceed 1024 threads per block (in total)
  - `threads_per_block( 16, 16, 1 );` 256 ✔
  - `threads_per_block( 32, 32, 1 );` 1024 ✔
  - `threads_per_block( 64, 64, 1 );` 4096 ✗

# 2D CUDA Ex. – matrix vector multiply

- Navigate into matVec_multiply_template/ directory
  - Edit the file: `matVec_multiply.cu`

- Write CUDA kernel for this program
  - HINT: Each thread will basically calculate the dot product of one row of the matrix **A** and the vector **x** (i.e. calculate one element of resulting vector).

# 2D CUDA Ex. – matrix vector multiply

$$
\begin{bmatrix}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{0,4} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4}
\end{bmatrix}
\begin{bmatrix}
x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4
\end{bmatrix}
=
\begin{bmatrix}
A_{0,0} * x_0 + A_{0,1} * x_1 + A_{0,2} * x_2 + A_{0,3} * x_3 + A_{0,4} * x_4 \\
\\
\\
\\
\end{bmatrix}
$$

| | | |
|---|---|---|
| ( 4 x 5 ) | ( 5 x 1 ) | ( 4 x 1 ) |
| ( M x N ) | ( N x 1 ) | ( M x 1 ) |

# 2D CUDA Ex. – matrix vector multiply

$$
\begin{bmatrix}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{0,4} \\
\boxed{A_{1,0}} & \boxed{A_{1,1}} & \boxed{A_{1,2}} & \boxed{A_{1,3}} & \boxed{A_{1,4}} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4}
\end{bmatrix}
\boxed{\begin{bmatrix}
x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4
\end{bmatrix}}
=
\begin{bmatrix}
A_{0,0} * x_0 + A_{0,1} * x_1 + A_{0,2} * x_2 + A_{0,3} * x_3 + A_{0,4} * x_4 \\
\boxed{A_{1,0} * x_0 + A_{1,1} * x_1 + A_{1,2} * x_2 + A_{1,3} * x_3 + A_{1,4} * x_4} \\
\\
\\
\end{bmatrix}
$$

( 4 x 5 ) ( 5 x 1 ) ( 4 x 1 )

( M x N ) ( N x 1 ) ( M x 1 )

# 2D CUDA Ex. – matrix vector multiply

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{0,4} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ \boxed{A_{2,0} \quad A_{2,1} \quad A_{2,2} \quad A_{2,3} \quad A_{2,4}} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \end{bmatrix} \boxed{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}} = \begin{bmatrix} A_{0,0} * x_0 + A_{0,1} * x_1 + A_{0,2} * x_2 + A_{0,3} * x_3 + A_{0,4} * x_4 \\ A_{1,0} * x_0 + A_{1,1} * x_1 + A_{1,2} * x_2 + A_{1,3} * x_3 + A_{1,4} * x_4 \\ \boxed{A_{2,0} * x_0 + A_{2,1} * x_1 + A_{2,2} * x_2 + A_{2,3} * x_3 + A_{2,4} * x_4} \\ \; \end{bmatrix}$$

( 4 x 5 ) ( 5 x 1 ) ( 4 x 1 )

( M x N ) ( N x 1 ) ( M x 1 )

# 2D CUDA Ex. – matrix vector multiply

$$
\begin{bmatrix}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{0,4} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\
\boxed{A_{3,0} \; A_{3,1} \; A_{3,2} \; A_{3,3} \; A_{3,4}} & & & &
\end{bmatrix}
\boxed{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}
=
\begin{bmatrix}
A_{0,0} * x_0 + A_{0,1} * x_1 + A_{0,2} * x_2 + A_{0,3} * x_3 + A_{0,4} * x_4 \\
A_{1,0} * x_0 + A_{1,1} * x_1 + A_{1,2} * x_2 + A_{1,3} * x_3 + A_{1,4} * x_4 \\
A_{2,0} * x_0 + A_{2,1} * x_1 + A_{2,2} * x_2 + A_{2,3} * x_3 + A_{2,4} * x_4 \\
\boxed{A_{3,0} * x_0 + A_{3,1} * x_1 + A_{3,2} * x_2 + A_{3,3} * x_3 + A_{3,4} * x_4}
\end{bmatrix}
$$

( 4 x 5 ) ( 5 x 1 ) ( 4 x 1 )

( M x N ) ( N x 1 ) ( M x 1 )

# 2D CUDA Ex. – matrix vector multiply

- Navigate into matVec_multiply_template/ directory
  - Edit the file: `matVec_multiply.cu`

- Write CUDA kernel for this program
  - HINT: Each thread will basically calculate the dot product of one row of the matrix **A** and the vector **x** (i.e. calculate one element of resulting vector).

- Assign the execution configuration parameters
  - `threads_per_block( ?, ?, ? );`
  - `blocks_in_grid( ?, ?, ? );`
  - NOTE: This should be a <u>1D</u> grid of threads

- When you finish, try changing values of M and N and adjust threads_per_block accordingly

```
dim3 threads_per_block( 1, 128, 1 );
dim3 blocks_in_grid( 1, ceil( float(M) / threads_per_block.y ), 1 );
```

```
__global__ void multiply_mat_vec(int *a, int *b, int *c)
{
    int row = blockDim.y * blockIdx.y + threadIdx.y;

    if(row < M)
    {
        for(int i=0; i<N; i++)
        {
            y[row] = y[row] + a[row*N + i] * x[i];
        }
    }
}
```

**Each thread computes one element of resulting vector (i.e. sum of N element-wise products)**

**Move through the 2D array with 1D indexing**

( M x N )

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

( N x 1 )

| |
|---|
| $x_0$ |
| $x_1$ |
| $x_2$ |
| $x_3$ |
| $x_4$ |

=

( M x 1 )

| | |
|---|---|
| $A_{0,0} * x_0 + A_{0,1} * x_1 + A_{0,2} * x_2 + A_{0,3} * x_3 + A_{0,4} * x_4$ | 0 |
| $A_{1,0} * x_0 + A_{1,1} * x_1 + A_{1,2} * x_2 + A_{1,3} * x_3 + A_{1,4} * x_4$ | 1 |
| $A_{2,0} * x_0 + A_{2,1} * x_1 + A_{2,2} * x_2 + A_{2,3} * x_3 + A_{2,4} * x_4$ | 2 |
| $A_{3,0} * x_0 + A_{3,1} * x_1 + A_{3,2} * x_2 + A_{3,3} * x_3 + A_{3,4} * x_4$ | 3 |

# Device Queries

# Device Queries

- **`cudaDeviceProp`**
  - `C struct with many member variables`
- **`cudaError_t cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`**
  - `CUDA API: returns info about the device`
- **`cudaError_t cudaGetDeviceCount(int *count)`**
  - `CUDA API: returns the number of CUDA-capable devices`

## Let's look at an example problem

- `intro_cuda/device_query/device_query.cu`

# Device Queries

- Add output for memory clock frequency and memory bus width
  - Google "**cudaDeviceProp**", find the member variables and add print statements

- Add user-defined output for theoretical memory bandwidth (GB/s)
  - BW (GB/s) = (memory clock rate) * (memory bus width)

GDDR5 - double data rate

"ticks per second"

"bits per tick"

$$\text{BW (GB/s)} = (\overbrace{(2.6\times10^{6}\ \text{kHz})}^{\text{Mem clock rate (kHz) (returned from query)}} * \underbrace{(1\times10^{3}\ \text{Hz / kHz})}_{\text{Convert from kHz to Hz}}) * (\overbrace{(384\ \text{bits})}^{\text{Lanes in bus memory (returned from query)}} * 2) * \underbrace{(\text{B / 8 bits})}_{\text{Convert from bits to B}}) * \underbrace{(\text{GB / }1\times10^{9}\ \text{B})}_{\text{Convert from B to GB}}$$

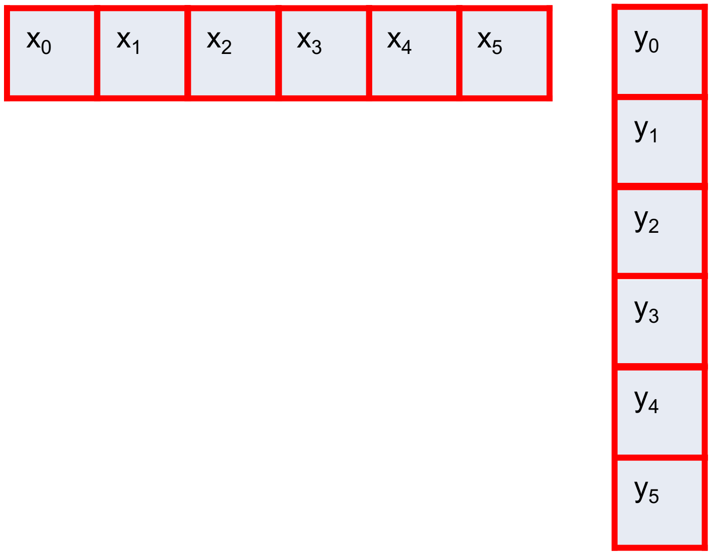
BW = ((memory clock rate in kHz)*1e3) * ((memory bus width in bits)*2) * (1/8) * (1/1e9) )

# Shared Memory

# 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

# Ex: `intro_cuda/dot_product/dot_product.cu`




$$= x_0 * y_0 + x_1 * y_1 + x_2 * y_2 + x_3 * y_3 + x_4 * y_4 + x_5 * y_5$$

# Ex: intro_cuda/dot_product/dot_product.cu

```
#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.

# Multi-Block dot_product Program

## The intro_cuda/dot_product/ program only works for a single block

- Can only perform dot product on arrays of at most 1024 elements (due to 1024 threads per block limit)
- Shared memory is only shared among threads within the same block

## In order to perform dot product on larger arrays, we need to use multiple blocks

- Each block computes element-wise products and sum on at most 1024 elements
- Then, the results from each block can be summed together for final result

## But when thread 0 from each block tries to update the (global) res variable, thread 0 from another block might also be writing to it

- Data race condition!
- Solution: Atomic Functions

# Atomic Functions

From the CUDA Programming Guide:

*An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.*

```
int atomicAdd(int *address, int val)
```

- Reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address.
- The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads.
  - No other thread can access this address until the operation is complete.

# Multi-Block dot_product Program

The program dot_product_multiBlock_template/ is currently identical to dot_product/ program

- Edit the template version so it can use multiple blocks (for larger arrays)

To do so

- Edit the kernel so that each blocks computes only THREADS_PER_BLOCK elements of the dot product (i.e. only a portion of the sum of products)
- Sum results from each block into global res variable using atomicAdd()

HINTS

- Each block's (local) thread 0 should be computing a portion of the dot product.
  - i.e. threadIdx.x instead of the global thread id
- In `__shared__ int products[num_elements]`, `num_elements` must be known at compile time

```
#define N 4096
#define THREADS_PER_BLOCK 511
```

Increase size of array
Set value of threads per block globally

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

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

    __syncthreads();

    if(threadIdx.x == 0)
    {
        int sum_of_products = 0;
        for(int i=0; i<THREADS_PER_BLOCK; i++)
        {
            sum_of_products = sum_of_products + products[i];
        }

        atomicAdd(res, sum_of_products);
    }
}
```

Since shared memory is only shared among threads in same block, only compute portion of dot product

Use thread 0 within each block to sum its block's portion of the dot product

Sum results from individual blocks using atomicAdd() to avoid race condition

# Where to go from here

CUDA C Programming Guide

- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

PARALLEL FORALL Blog (search through archives)

- https://devblogs.nvidia.com/parallelforall/

Programming Massively Parallel Processors: A Hands-on Approach

- David B. Kirk, Wen-mei Hwu (3rd Edition)

CUDA by Example: An Introduction to General-Purpose GPU Programming

- https://developer.nvidia.com/cuda-example

cuBLAS – Dense Linear Algebra on GPUs

- https://developer.nvidia.com/cublas

Things we didn't cover

- Timing CUDA codes, NVIDIA visual profiler, many others