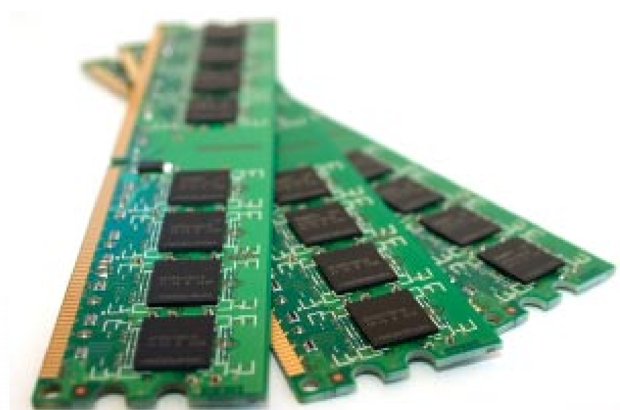
CUDA Programming

Outline

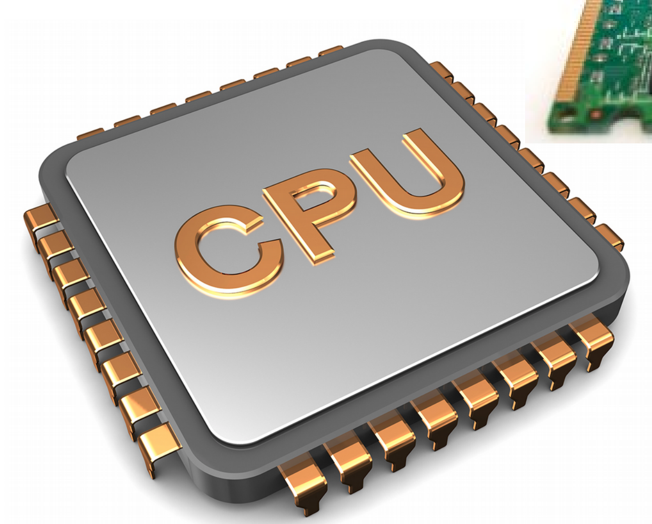
- Basic CUDA program

CUDA C - Terminology

- Host – The CPU and its memory (host memory)
- Device – The GPU and its memory (device memory)



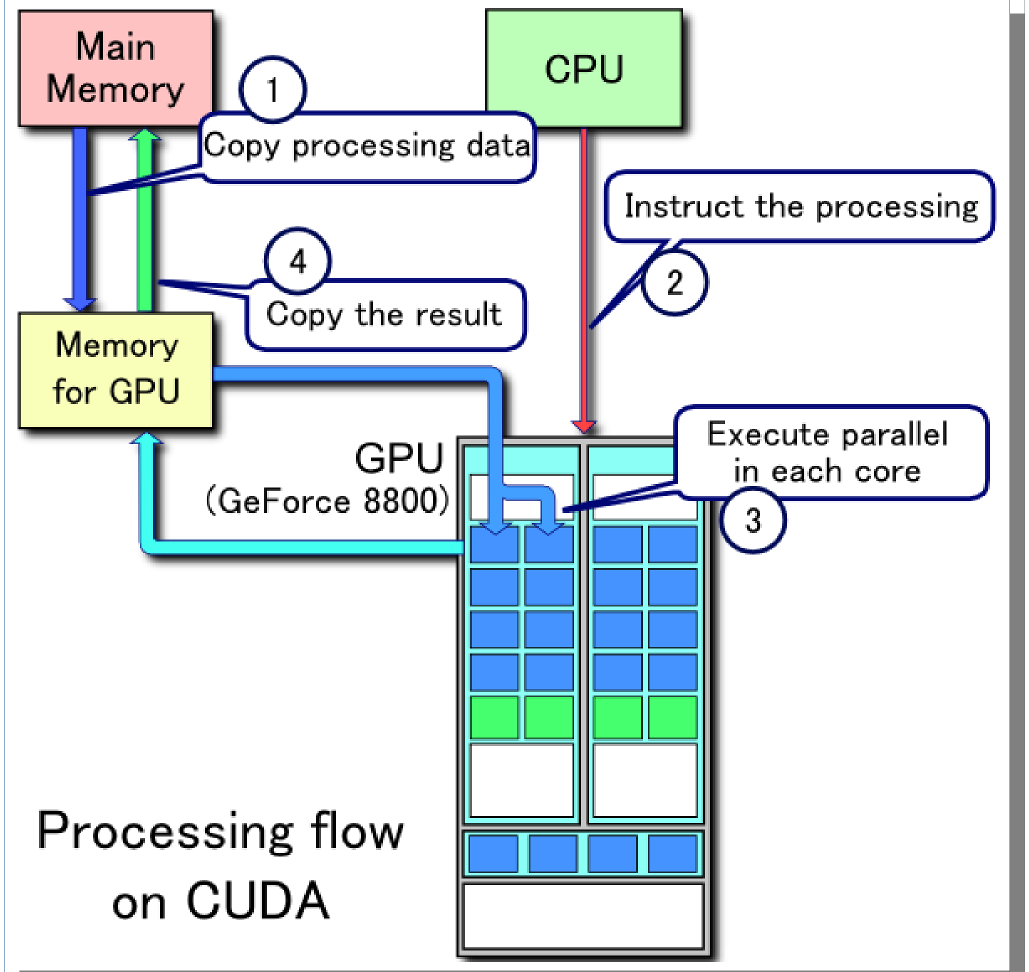
HOST



DEVICE

Compute Unified Device Architecture

- Hybrid CPU/GPU Code
- Low latency code is run on CPU
  - Result immediately available
- High latency, high throughput code is run on GPU
  - Result on bus
  - GPU has many more cores than CPU



Hello, World!

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

- The standard C program runs on the host
- NVIDIA’s compiler (nvcc) will not complain about CUDA programs with no device code
- At its simplest, CUDA C is just C!

Hello, World! with Device Code

Function runs on the device. Called from the host.

```
__global__ void kernel( void ) {
}

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

Hello, World! with Device Code

Function runs on the device. Called from the host.

```
__global__ void kernel( void ) {
}

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

- nvcc splits source file into *host* and *device* components
  - NVIDIA's compiler handles device functions. eg. **kernel()**
  - Standard host compiler handles host functions like **main()**

Hello, World! with Device Code

Function runs on the device. Called from the host.

```
__global__ void kernel( void ) {
}

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

- Triple angle brackets mark a call from host code to device code - a kernel launch
- The kernel executes on the GPU

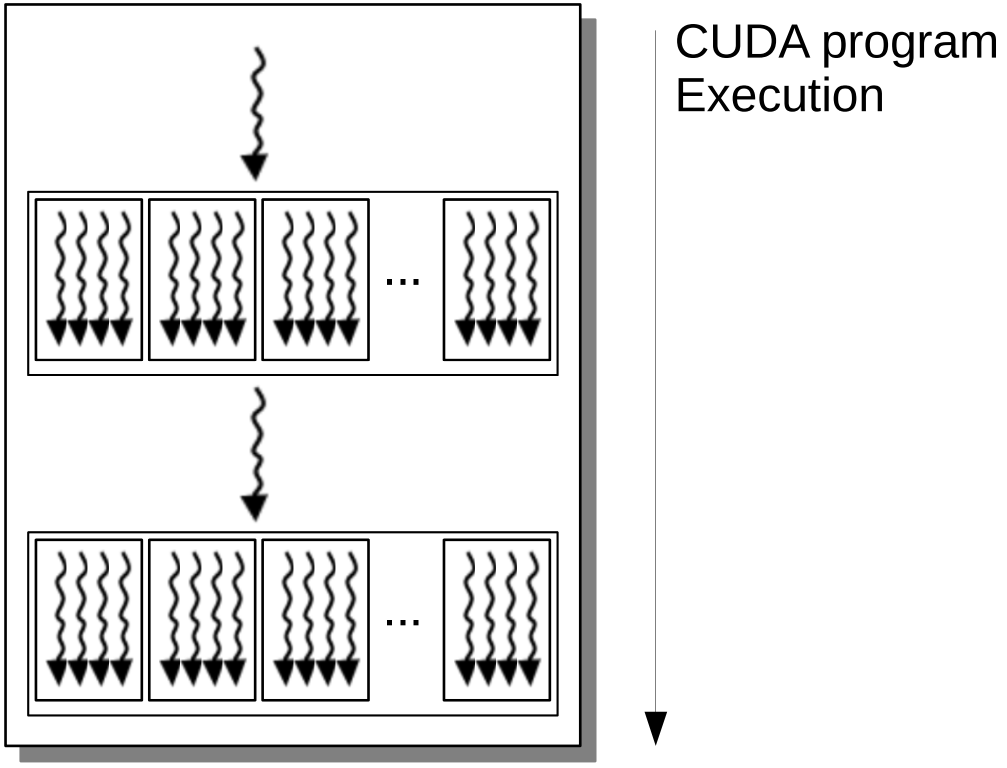
CUDA /OpenCL – Execution Model

CPU Serial Code (on Host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

CPU Serial Code (on Host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);



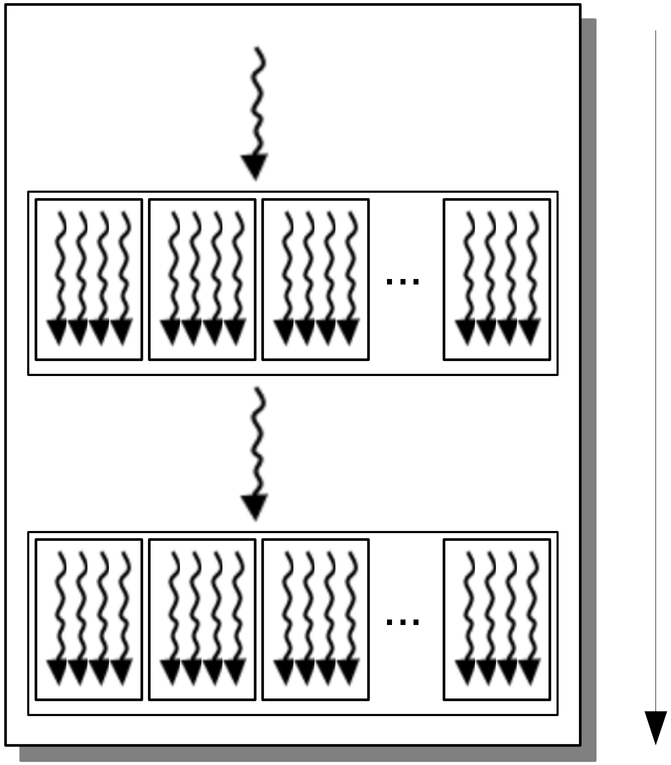
CUDA /OpenCL – Execution Model

CPU Serial Code (on Host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

CPU Serial Code (on Host)

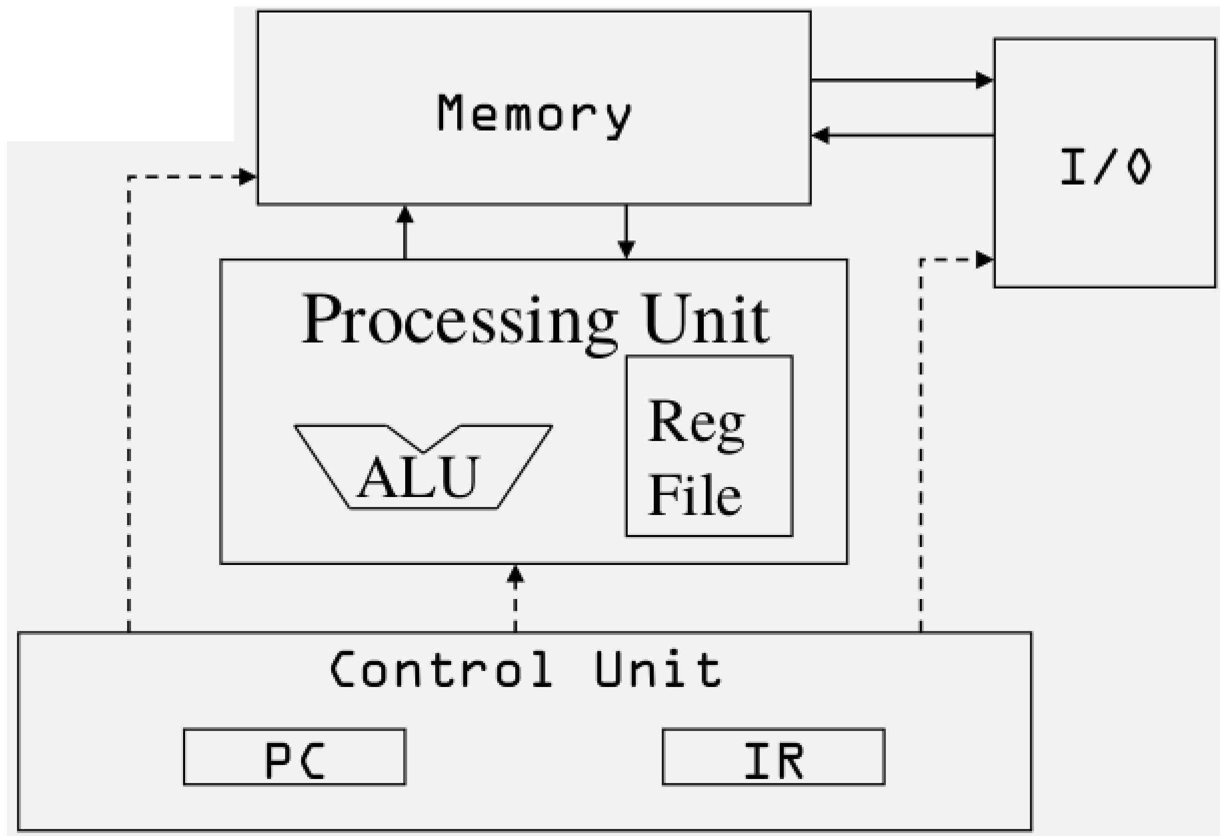
Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);



CUDA program Execution

- Heterogeneous host+device application C program
  - Serial parts in host C code
  - Parallel parts in device SPMD kernel C code

A von-Neumann Processor



- A thread is a “virtualized” or “abstracted” von-Neumann Processor

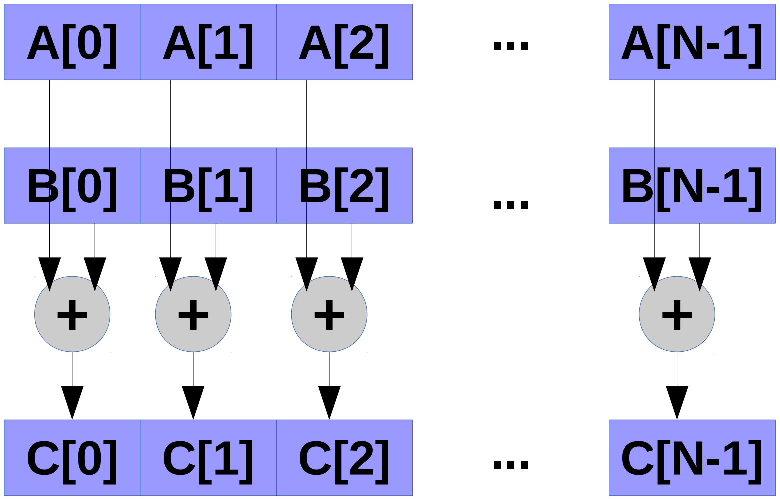
CUDA Program Example

Vector Addition Example

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

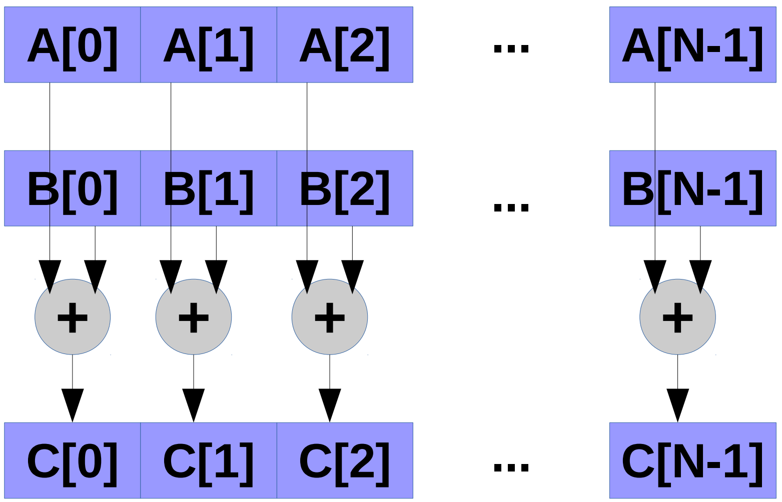
Vector Addition Example

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



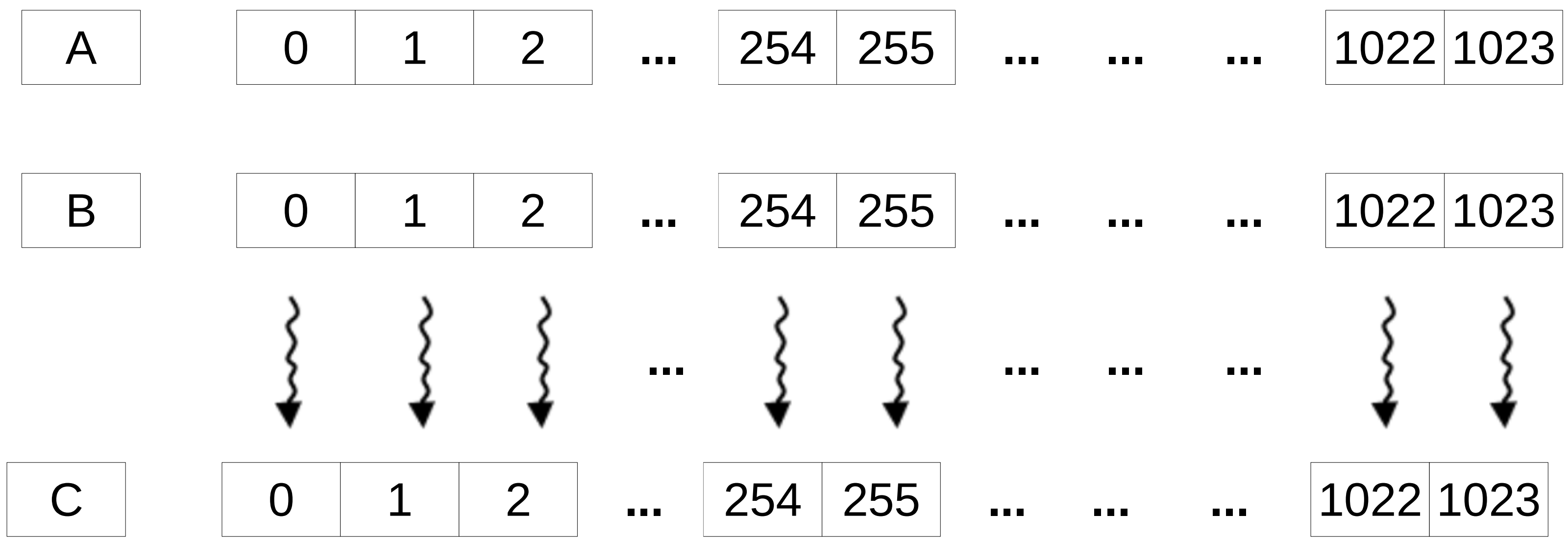
Vector Addition Example

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

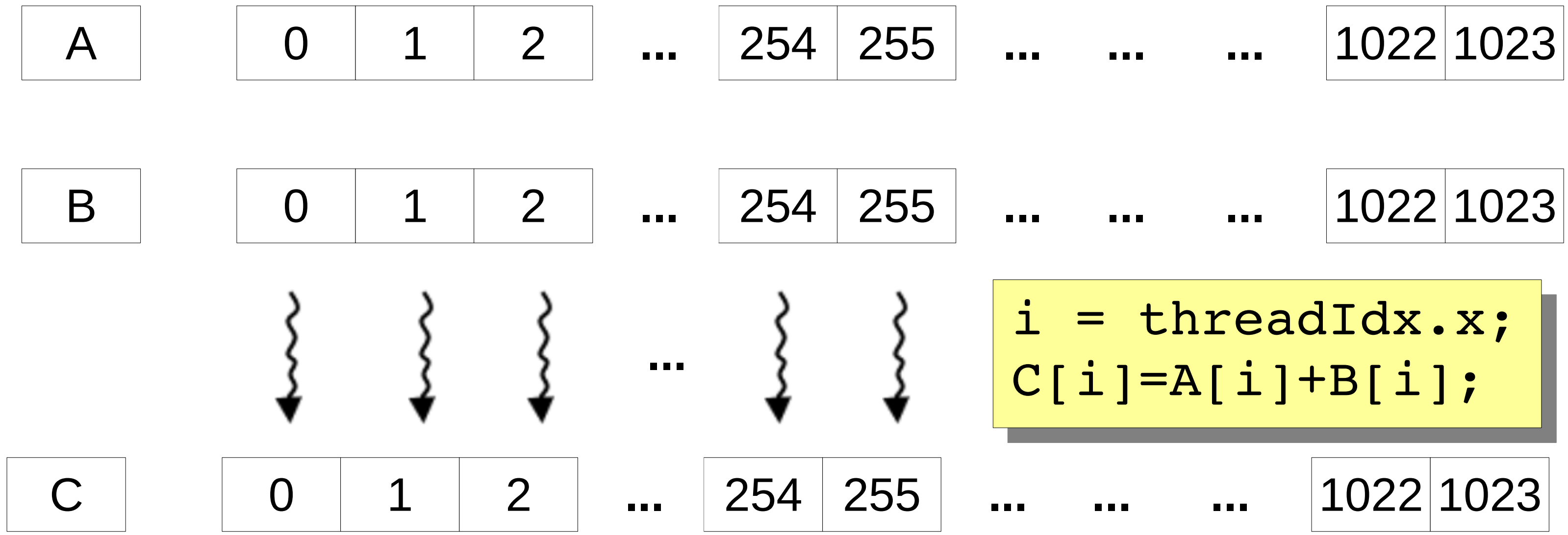


- Each thread adds one element from A[] and B[] and updates one element in C[]
  - Data level parallelism!

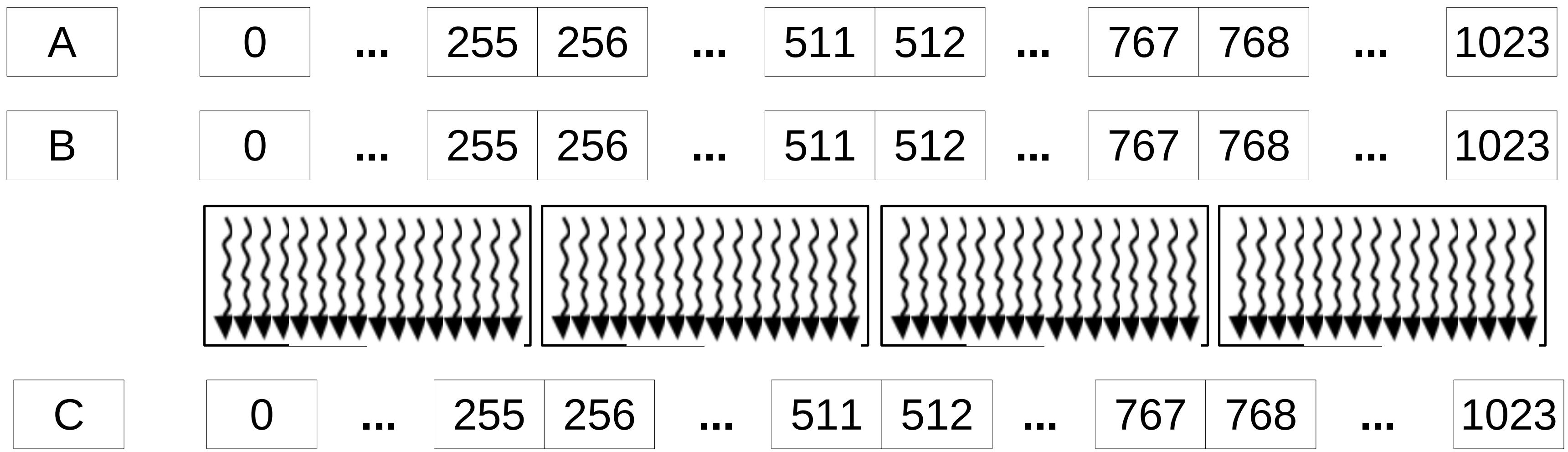
Parallel Threads



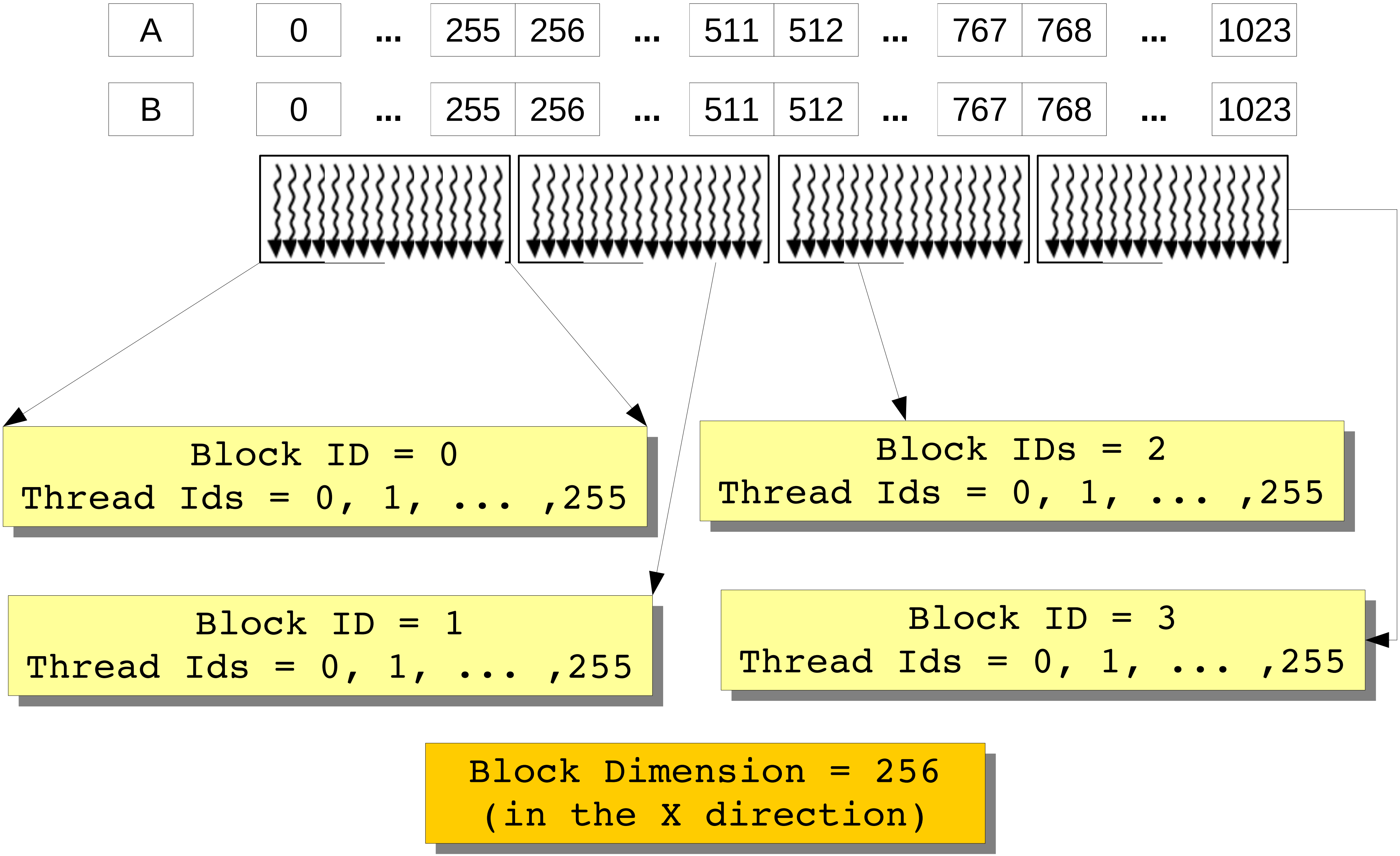
Parallel Threads



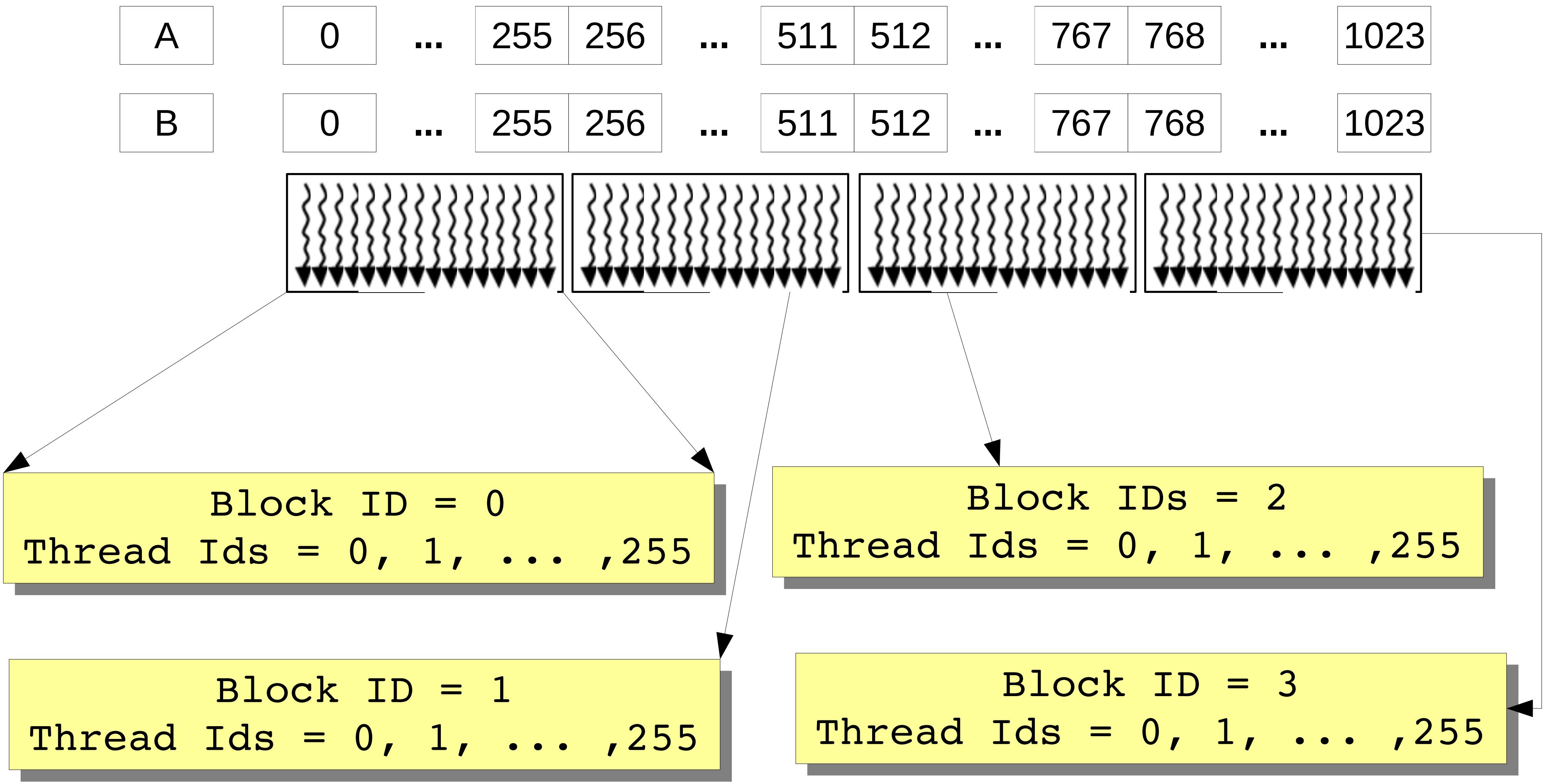
Arrays of Parallel Threads



Arrays of Parallel Threads

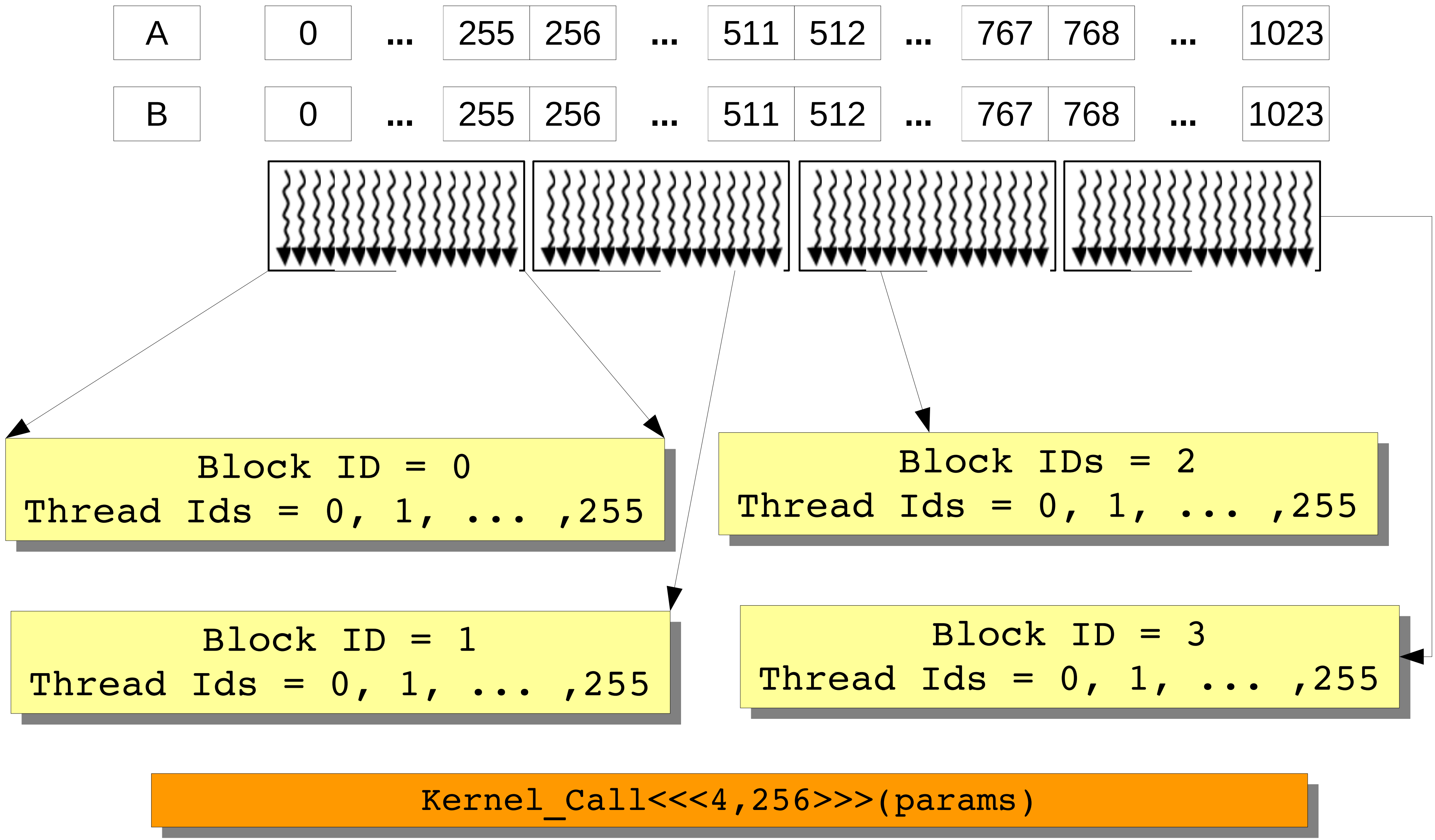


Arrays of Parallel Threads



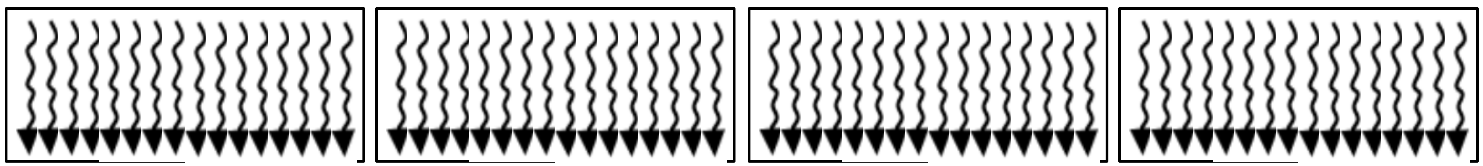

```
Kernel_Call<<<No_of_Blocks,ThreadsPerBlock>>>(params)
```

Arrays of Parallel Threads



Arrays of Parallel Threads

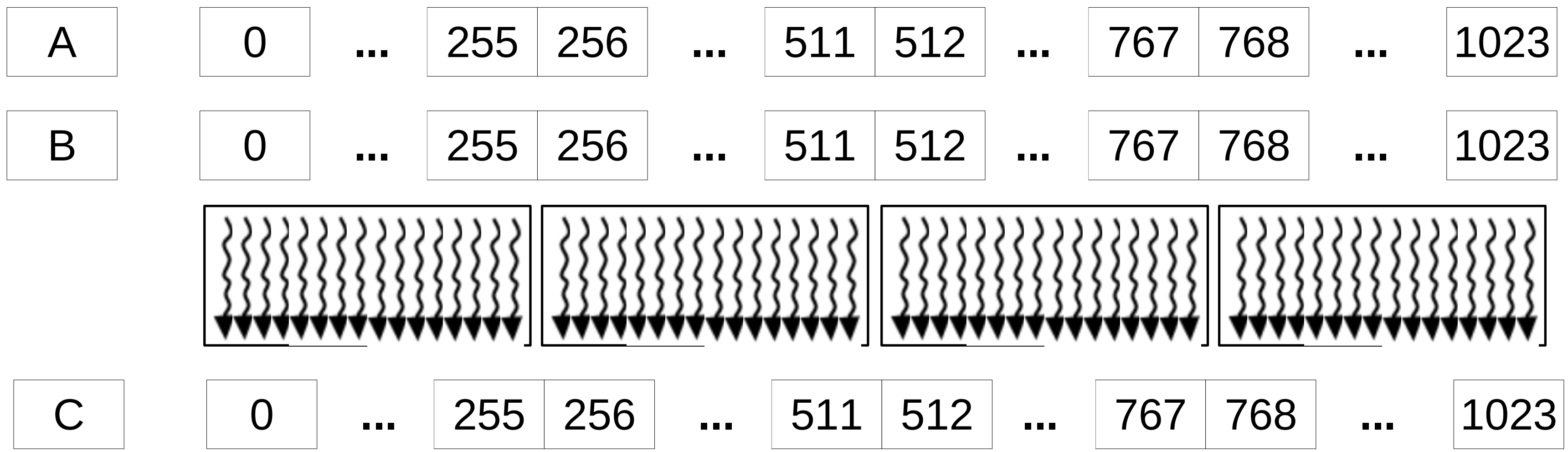
A	0	...	255	256	...	511	512	...	767	768	...	1023
B	0	...	255	256	...	511	512	...	767	768	...	1023



C	0	...	255	256	...	511	512	...	767	768	...	1023

```
i = blockIDx.x * blockDim.x + threadIdx.x;
C[i]=A[i]+B[i];
```

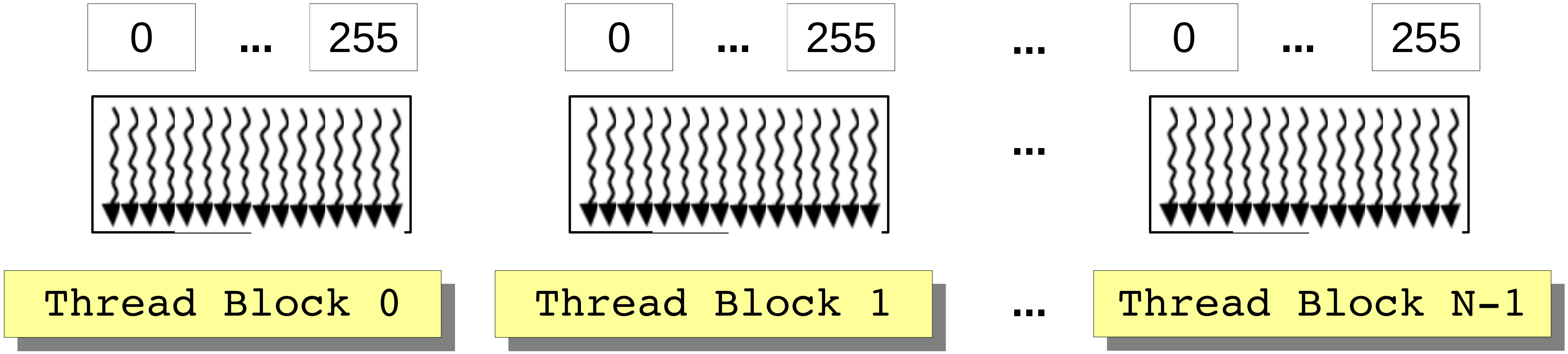
Arrays of Parallel Threads



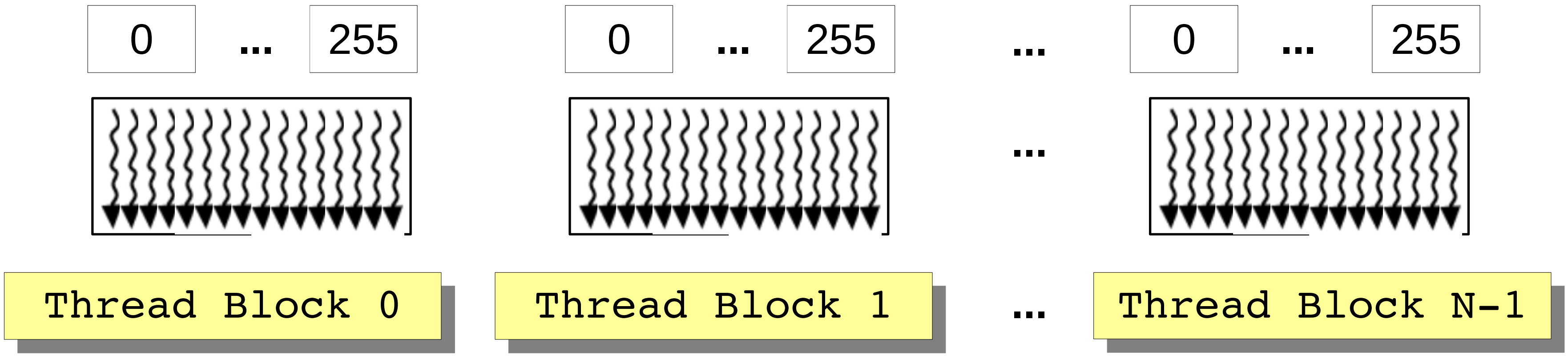

```
i = blockIDx.x * blockDim.x + threadIdx.x;
C[i]=A[i]+B[i];
```

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (SPMD)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions

Thread Blocks: Scalable Cooperation

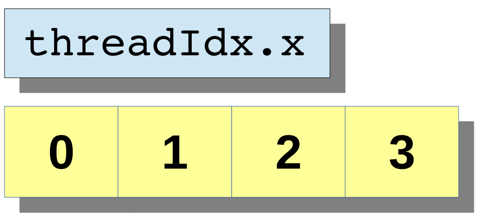


Thread Blocks: Scalable Cooperation



- Thread array is divided into multiple blocks
- Threads within a block cooperate via **shared memory**, **atomic operations** and **barrier synchronization**
- Threads in different blocks do not interact

blockIdx and threadIdx



blockIdx and threadIdx

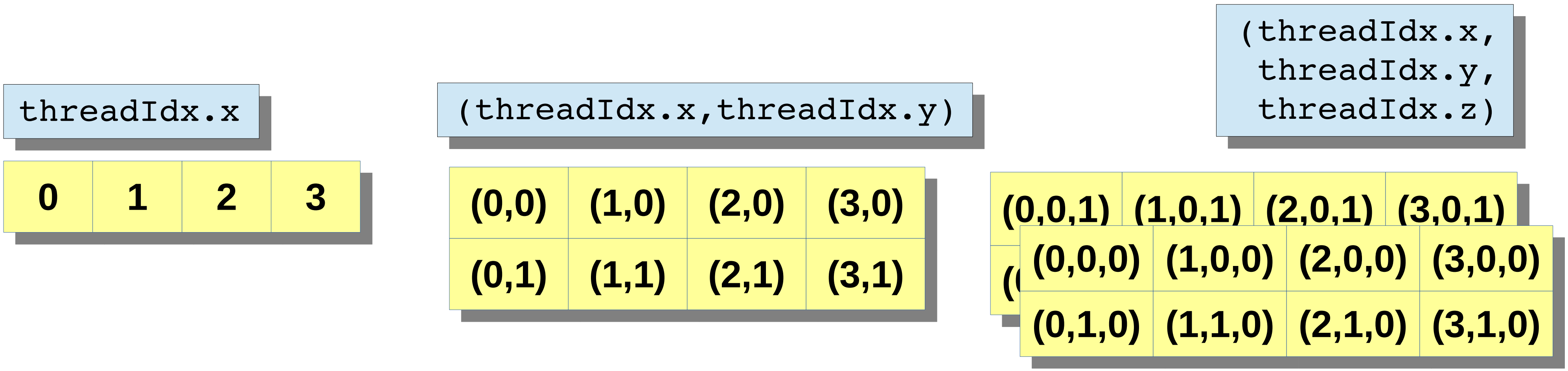
`threadIdx.x`

0	**1**	**2**	**3**

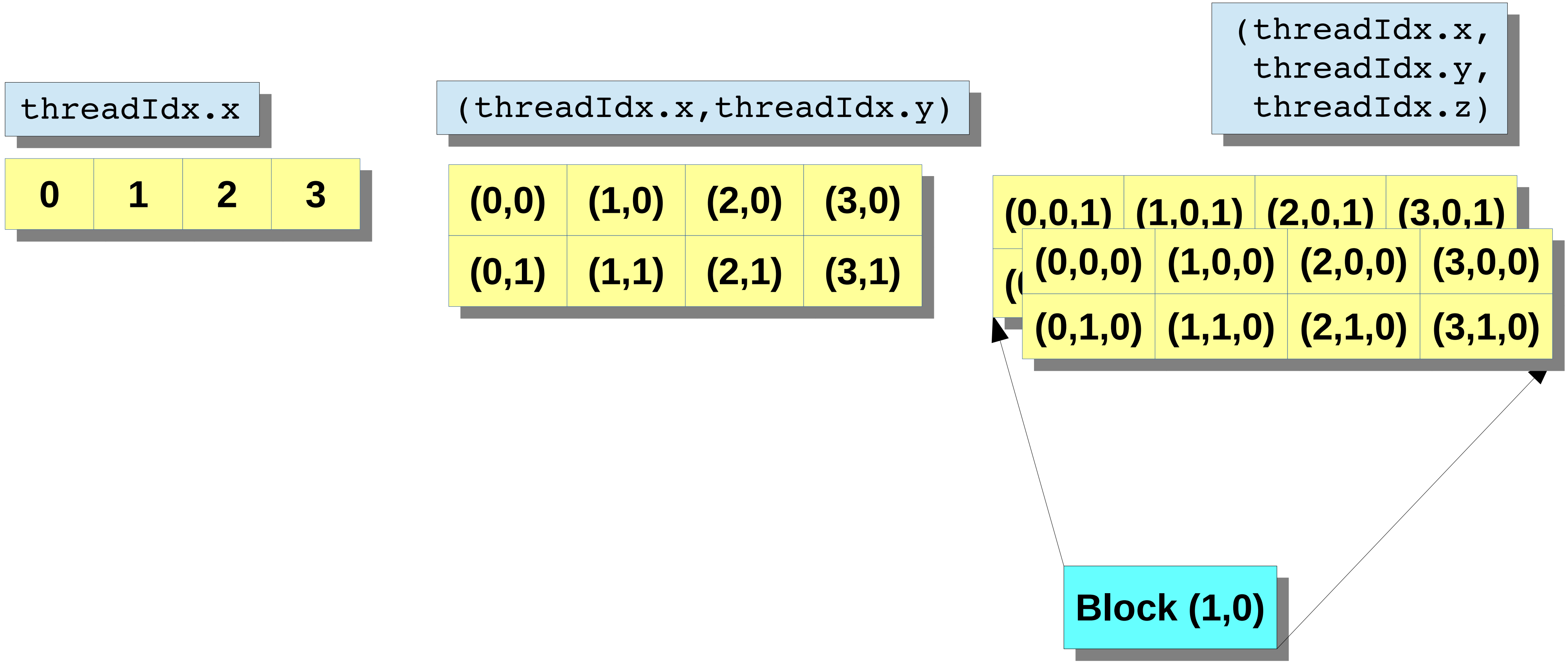
`(threadIdx.x,threadIdx.y)`

(0,0)	**(1,0)**	**(2,0)**	**(3,0)**
(0,1)	**(1,1)**	**(2,1)**	**(3,1)**

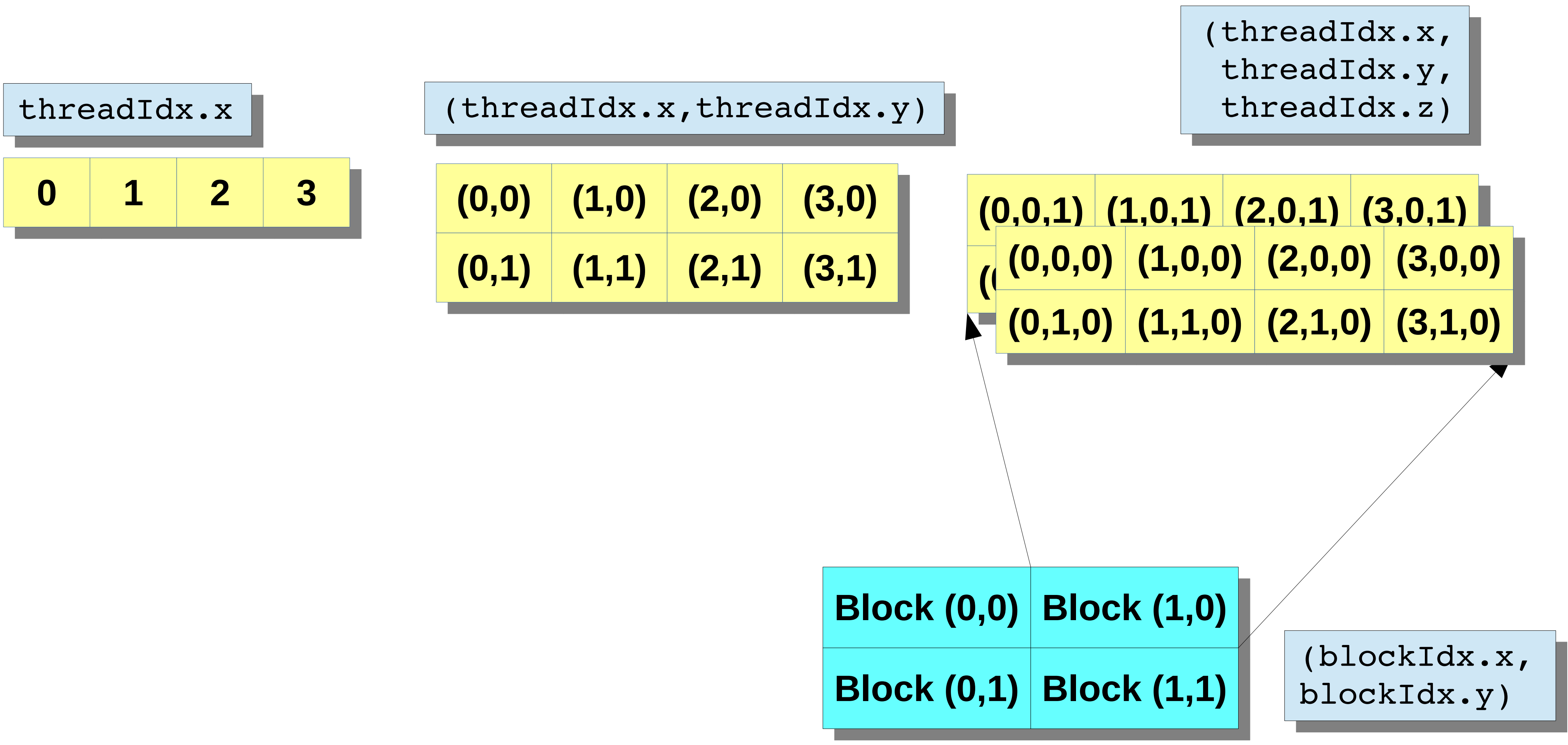
blockIdx and threadIdx



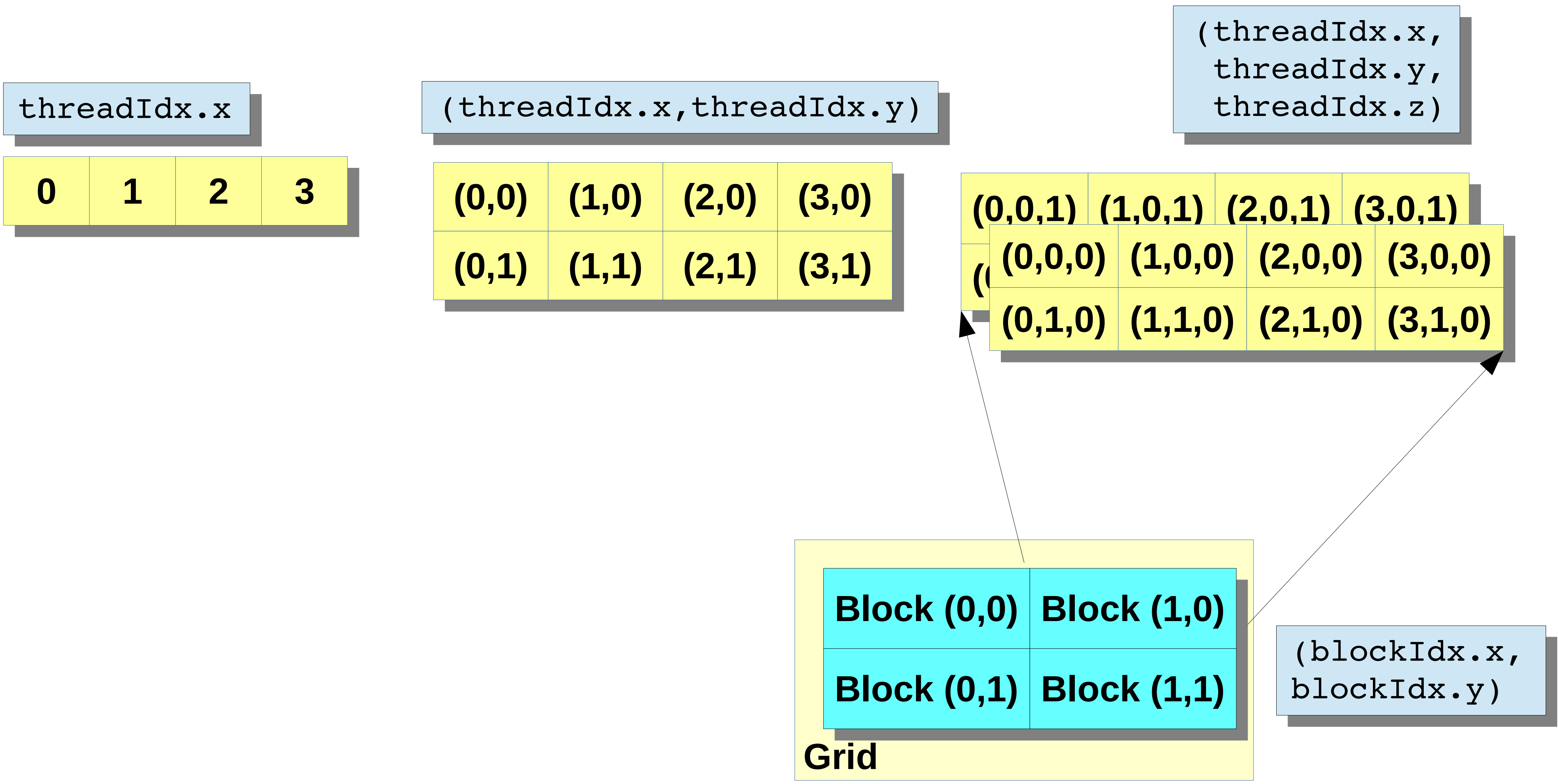
blockIdx and threadIdx



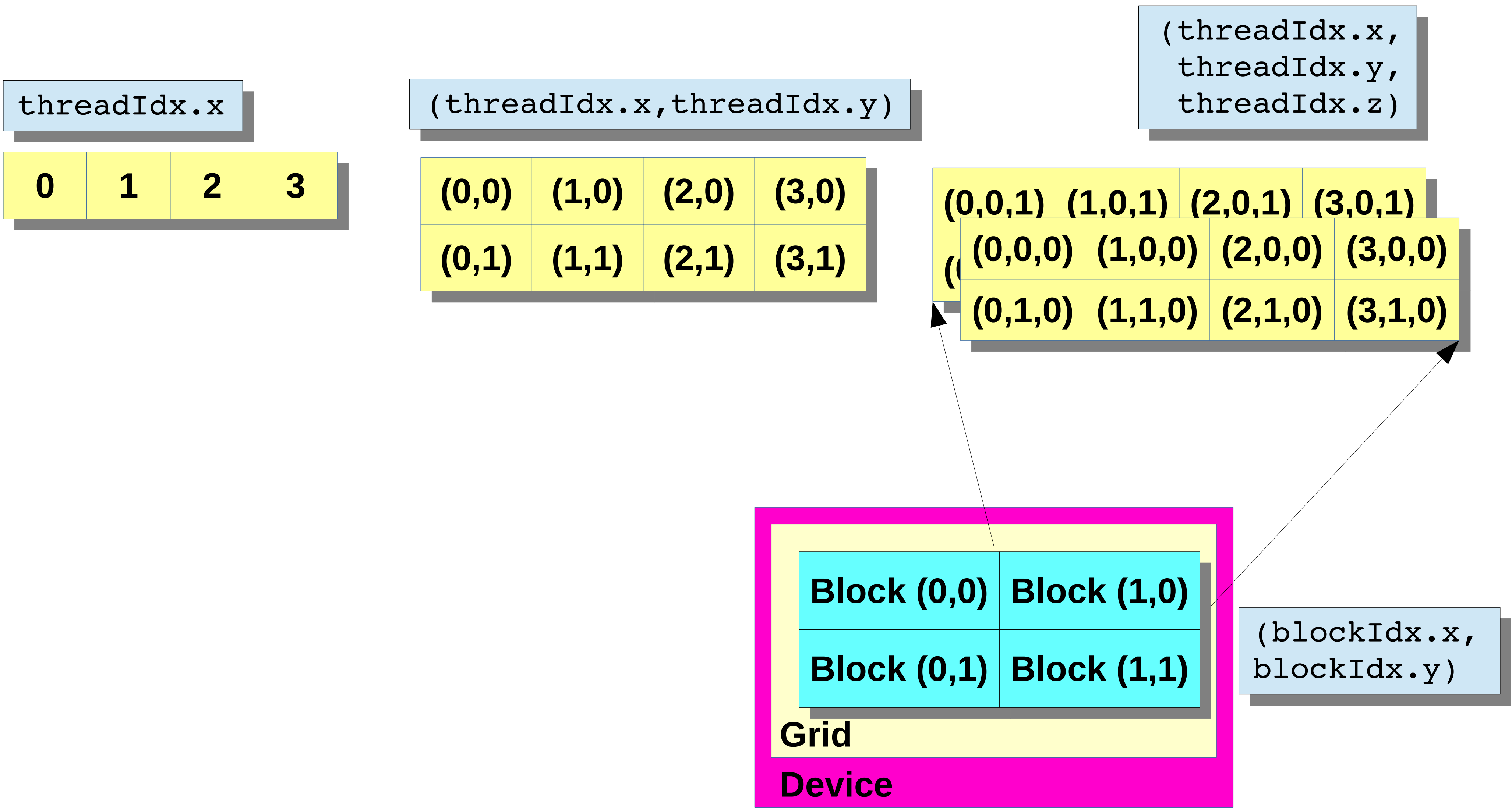
blockIdx and threadIdx



blockIdx and threadIdx



blockIdx and threadIdx



blockIdx and threadIdx

- Each thread uses indices to decide what data to work on

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
- blockIdx: 1D, 2D, or 3D (CUDA 4.0)

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
- blockIdx: 1D, 2D, or 3D (CUDA 4.0)
- threadIdx: 1D, 2D, or 3D

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
- blockIdx: 1D, 2D, or 3D (CUDA 4.0)
- threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes

CUDA C – Vector Addition Kernel

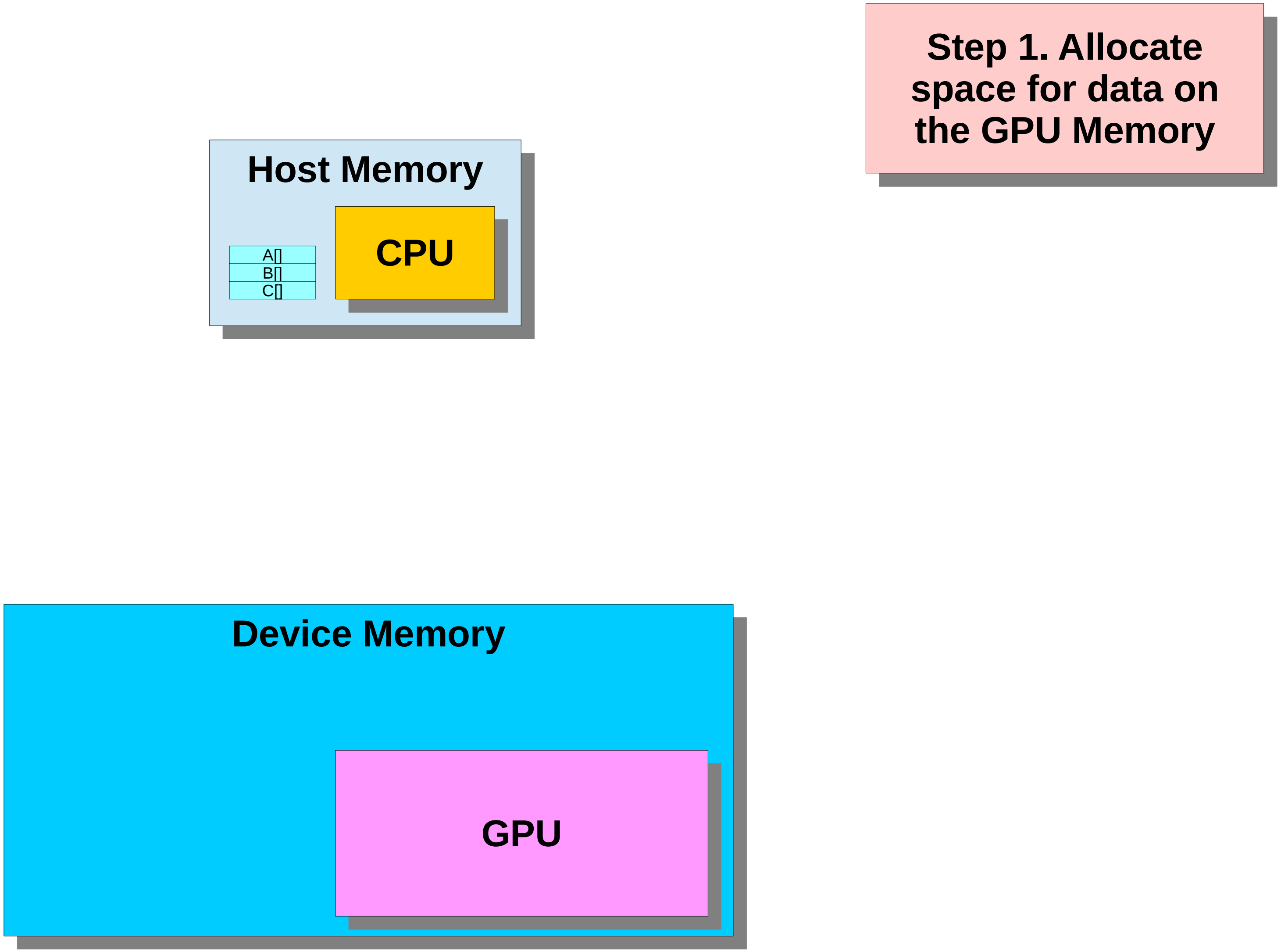
Vector Addition – C Code

```
int main()
{
   // Memory allocation for h_A, h_B, and h_C
   // I/O to read h_A and h_B, N elements
   ...
   vecAdd(h_A, h_B, h_C, N);
}
```

Vector Addition – C Code

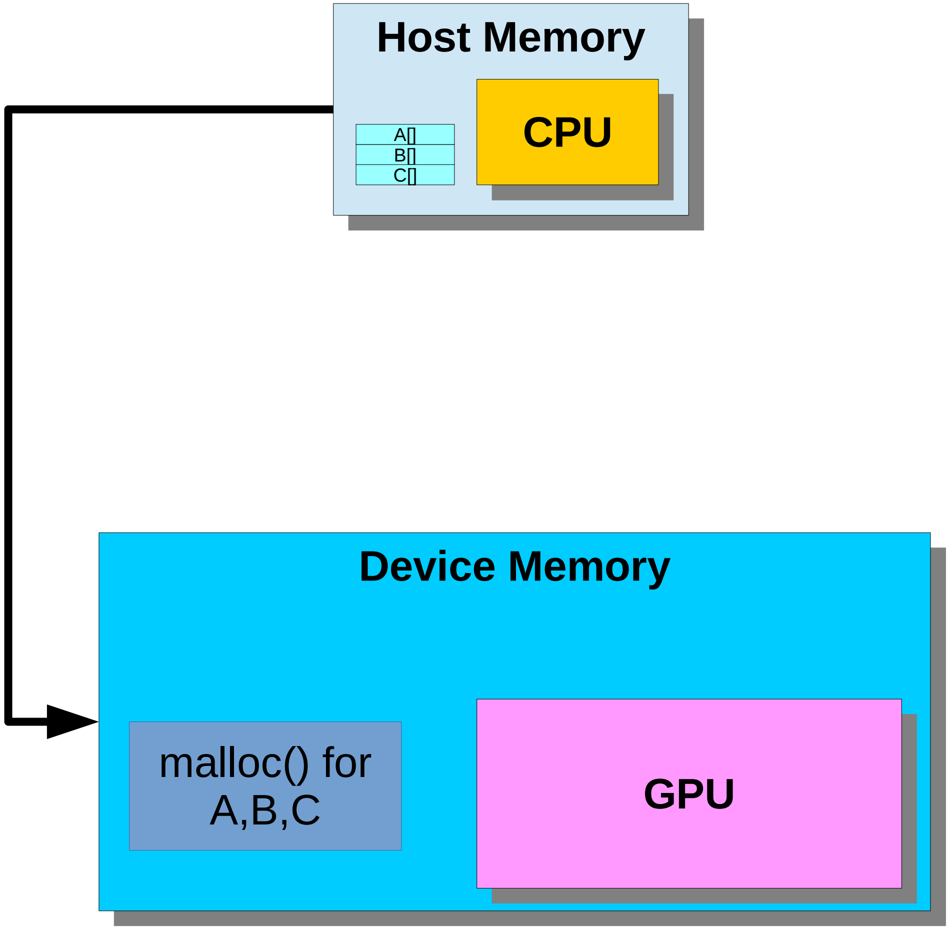
```
// Compute vector sum C = A+B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int i;
   for (i = 0; i<n; i++) h_C[i] = h_A[i]+h_B[i];
}

int main()
{
   // Memory allocation for h_A, h_B, and h_C
   // I/O to read h_A and h_B, N elements
   ...
   vecAdd(h_A, h_B, h_C, N);
}
```

vecAdd - CUDA Version

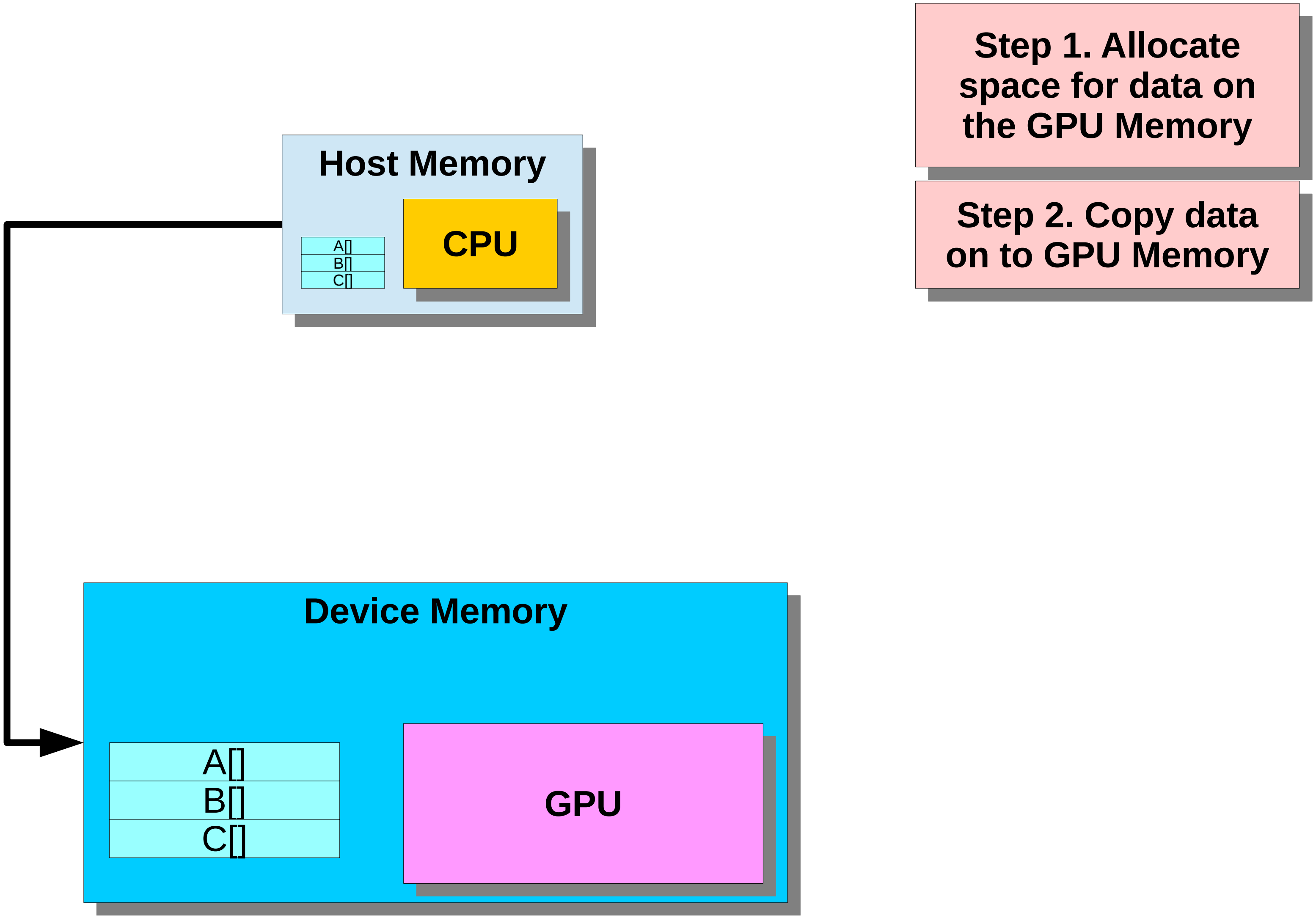


vecAdd - CUDA Version

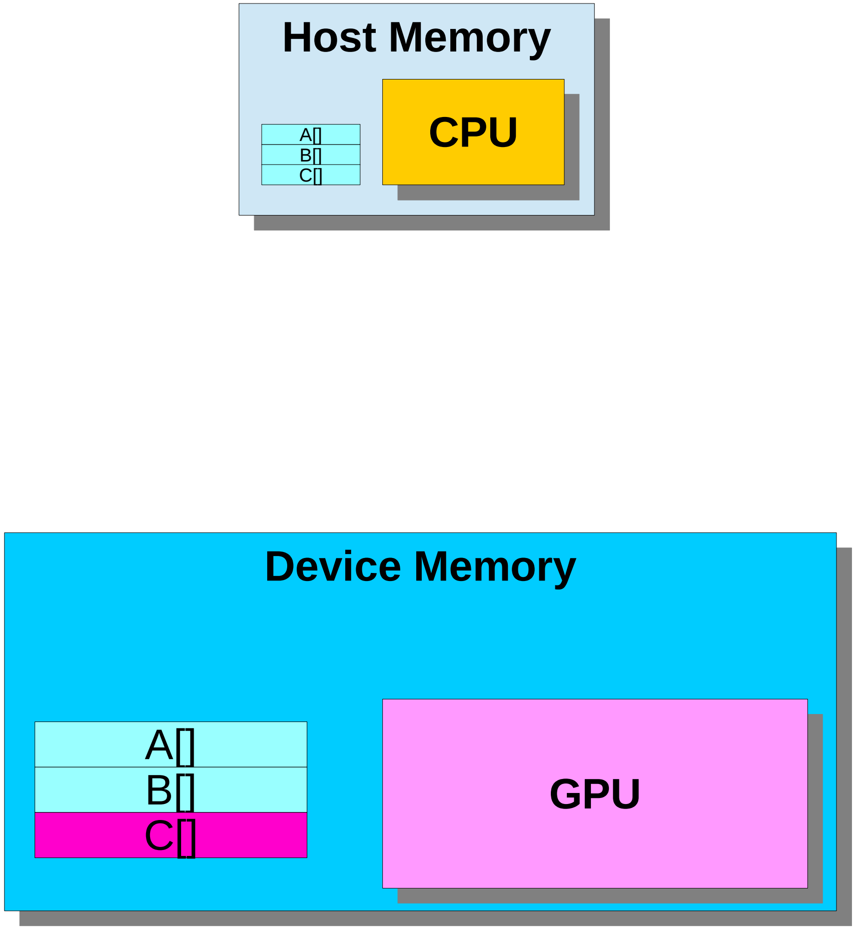
Step 1. Allocate space for data on the GPU Memory



vecAdd - CUDA Version



vecAdd - CUDA Version

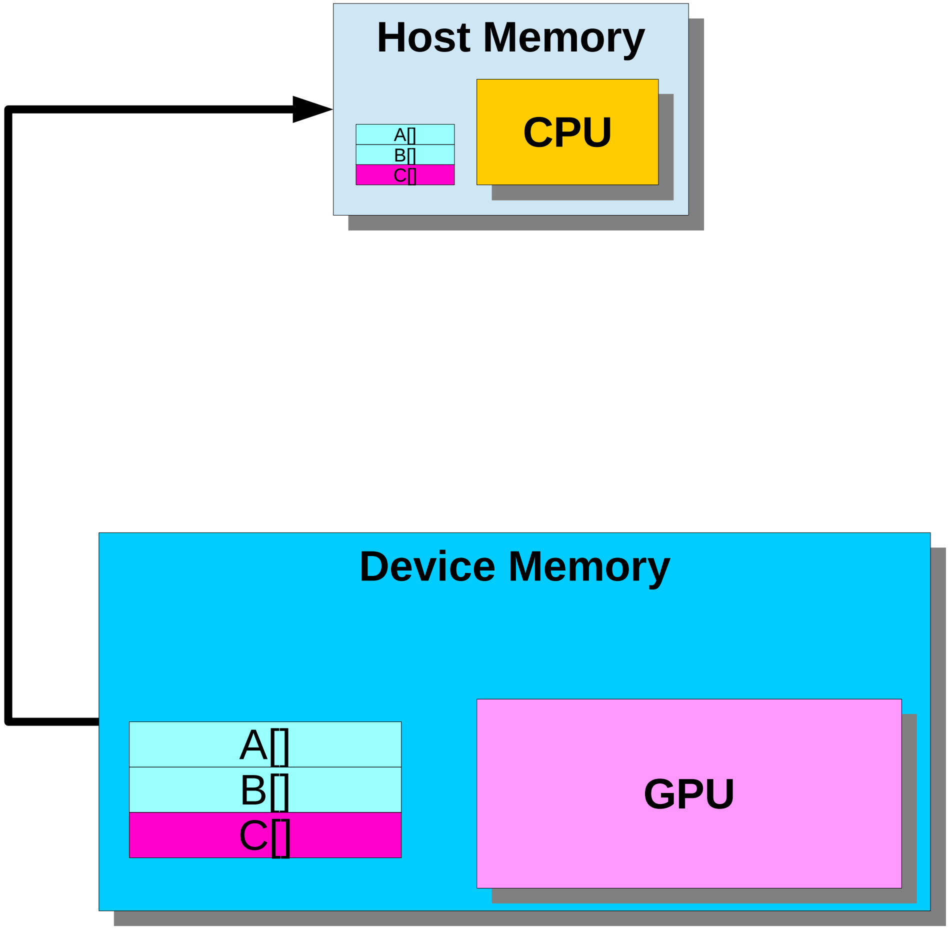


Step 1. Allocate space for data on the GPU Memory

Step 2. Copy data on to GPU Memory

Step 3. Kernel Launch.

vecAdd - CUDA Version



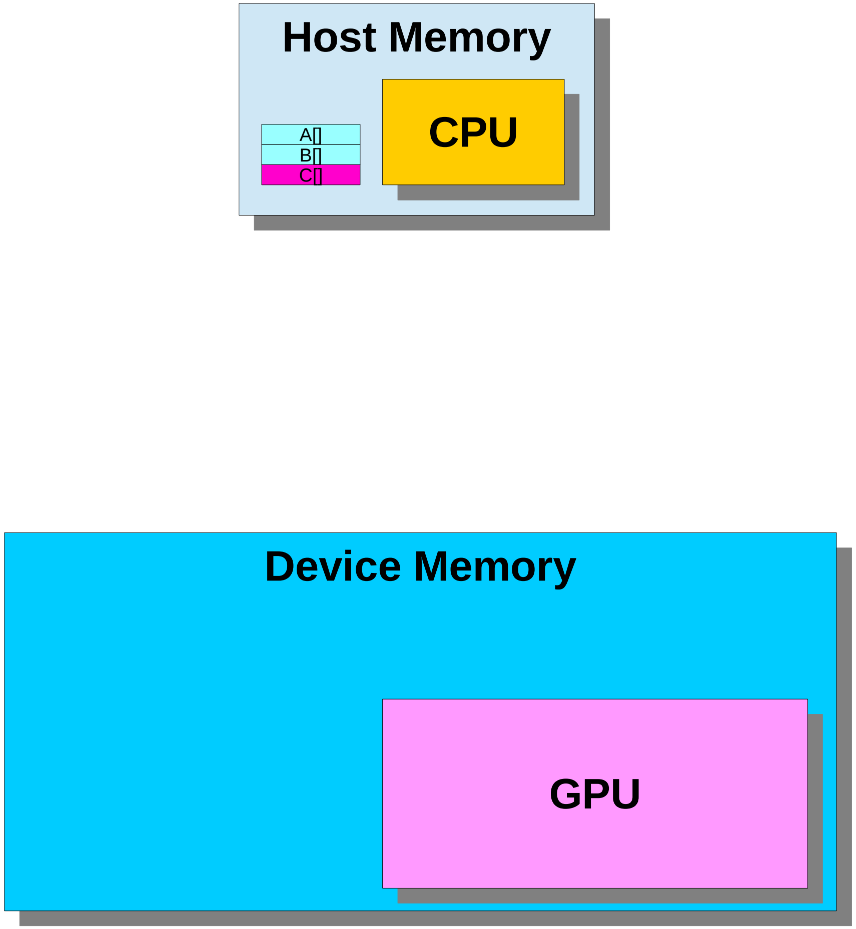
Step 1. Allocate space for data on the GPU Memory

Step 2. Copy data on to GPU Memory

Step 3. Kernel Launch.

Step 4. Copy result data back to Host Main Memory

vecAdd - CUDA Version



Step 1. Allocate space for data on the GPU Memory

Step 2. Copy data on to GPU Memory

Step 3. Kernel Launch.

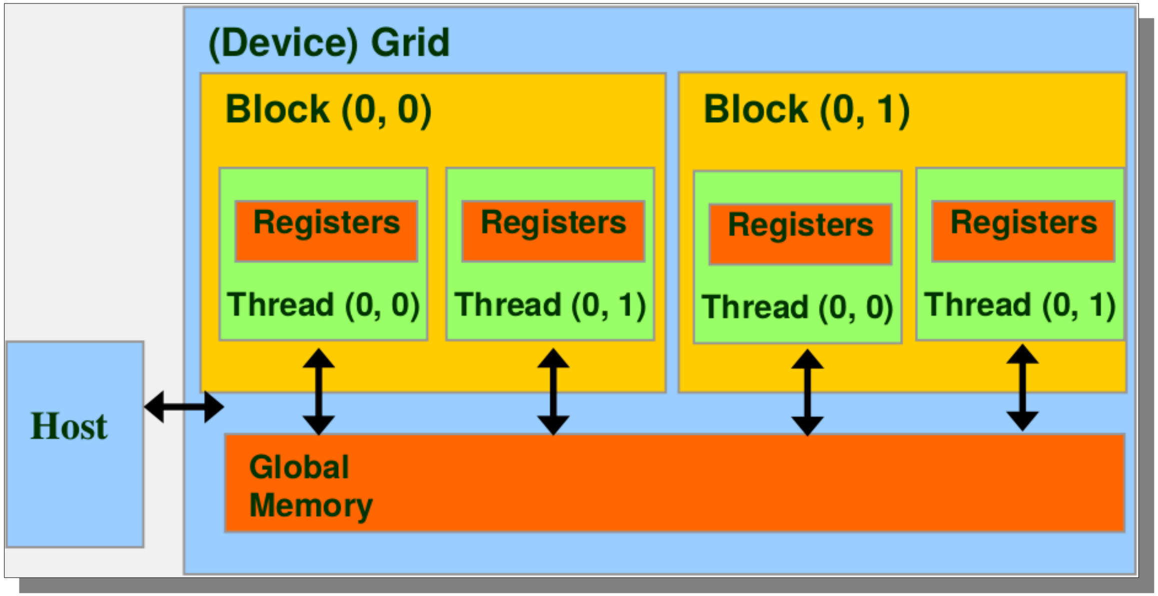
Step 4. Copy result data back to Host Main Memory

Step 5. Free device memory

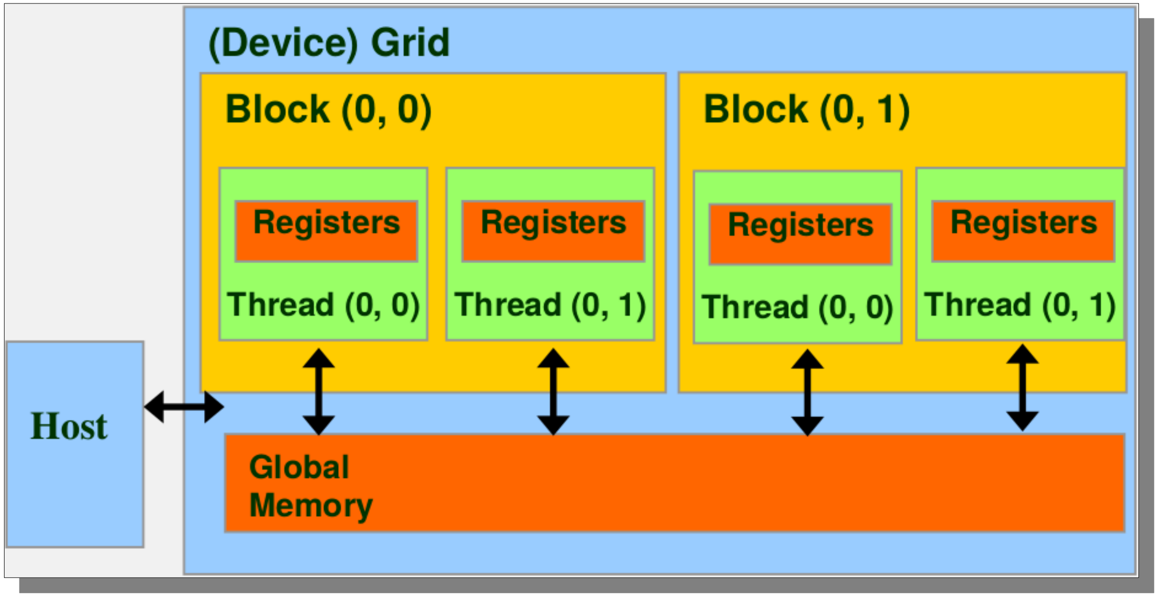
vecAdd - CUDA Host Code

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n* sizeof(float);
   float *d_A, *d_B, *d_C;
   1. // Allocate device memory for A, B, and C
      // copy A and B to device memory
   2. // Kernel launch code – the device performs the
      // actual vector addition
   3. // copy C from the device memory // Free device
      // vectors
}
```

CUDA Memories – Quick Overview

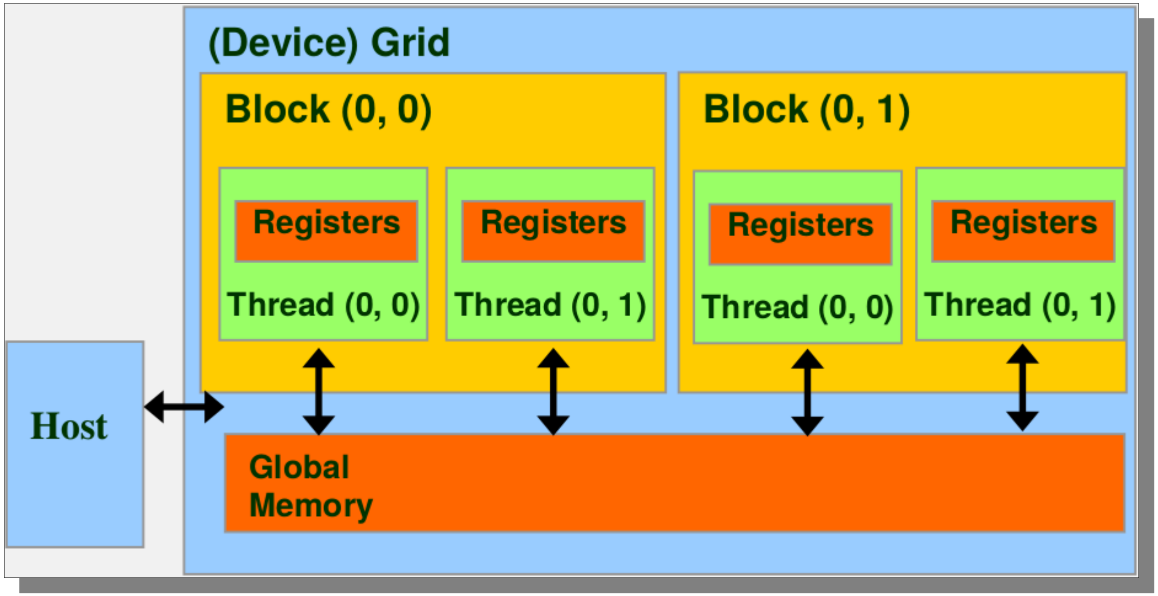


CUDA Memories – Quick Overview



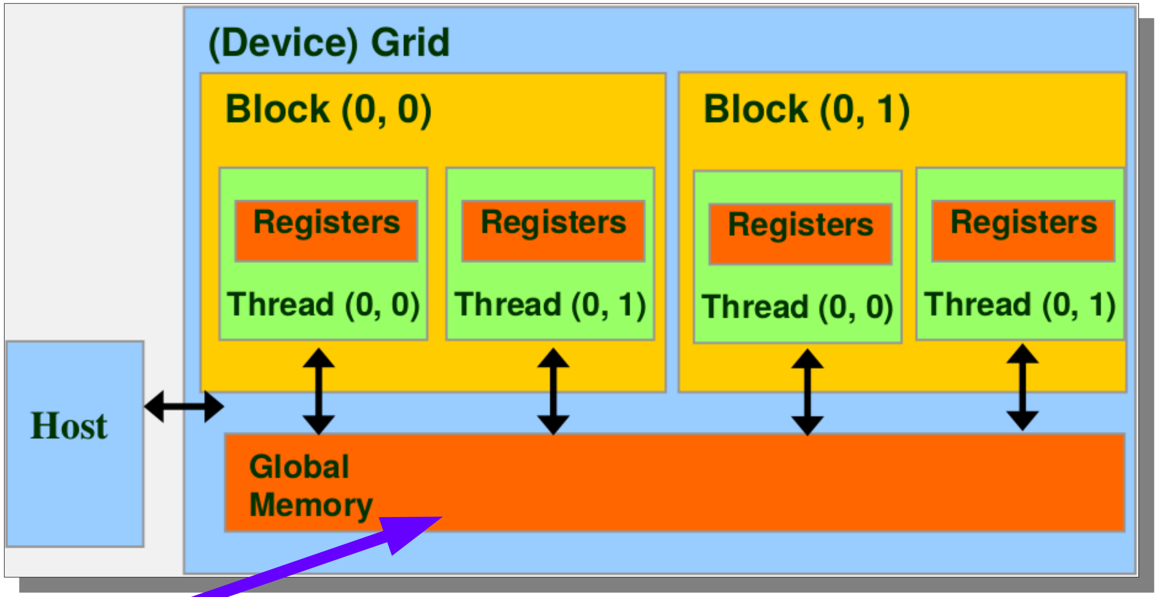
- Device code can:
  - R/W per-thread **registers**
  - R/W all-shared **global memory**

CUDA Memories – Quick Overview



- Device code can:
  - R/W per-thread **registers**
  - R/W all-shared **global memory**
- Host code can:
  - Transfer data to/from per grid **global memory**

CUDA Device Memory Management API



cudaMalloc()

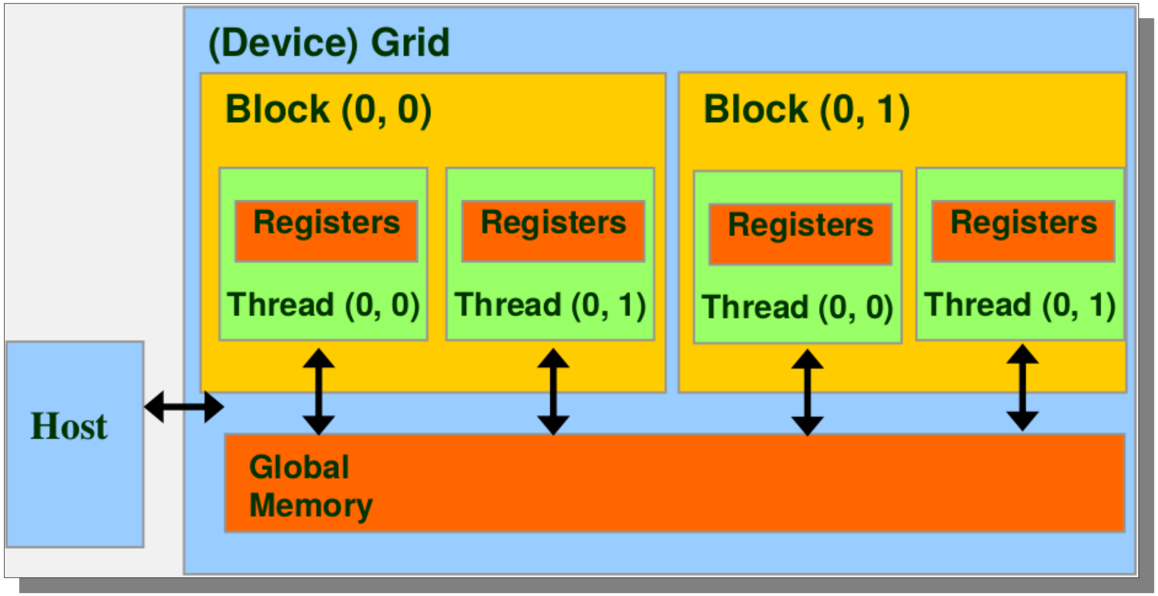
Allocates object in the device global memory

Two Parameters:

- **Address of a pointer to the allocated object**
- **Size of allocated object in terms of bytes**

```
cudaMalloc((void **) &d_A, size);
```

CUDA Device Memory Management API



cudaMalloc()

Allocates object in the device global memory

Two Parameters:

- Address of a pointer to the allocated object
- Size of allocated object in terms of bytes

```
cudaMalloc((void **) &d_A, size);
```

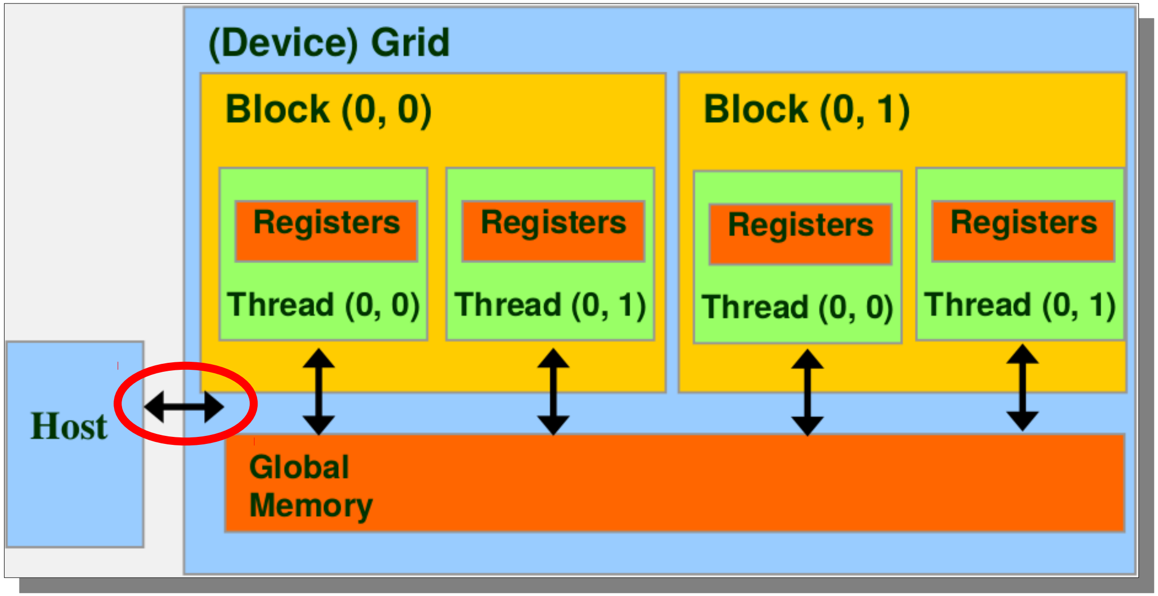
cudaFree()

Frees object from device global memory

Pointer to freed object

```
cudaFree(d_A);
```

Host-Device Data Transfer API



cudaMemcpy()

Memory data transfer

Four parameters

- **Pointer to destination, Pointer to source, bytes copied, Type/Direction of transfer**

Transfer to device is asynchronous

```
cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
```

Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n * sizeof(float);
   float *d_A, *d_B, *d_C;
   cudaMalloc((void **) &d_A, size);
   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &d_B, size);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &d_C, size);
// Kernel invocation code – not shown here
   cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
   cudaFree(d_A);
   cudaFree(d_B);
   cudaFree(d_C);
}
```

Error Checking for API

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess) {
   printf(“%s in %s at line %d\n”,
         cudaGetErrorString(err), __FILE__, __LINE__);
   exit(EXIT_FAILURE);
}
```

Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
   int i = threadIdx.x+blockDim.x*blockIdx.x;
   if(i<n)
      C[i] = A[i] + B[i];
}
```

Host Code – Kernel Launch

```
int vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
   // ..., cudaMalloc(), cudaMemcpy(), ...
   // ...
   // Run ceil(n/256.0) blocks of 256 threads each
   vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
   // ...
}
```

Better Kernel Launch Code

```
int vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
   // ..., cudaMalloc(), cudaMemcpy(), ...
   // ...
   // Run ceil(n/256.0) blocks of 256 threads each
   dim3 DimGrid((n-1)/256+1, 1, 1);
   dim3 DimBlock(256, 1, 1);
   vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
   // ...
}
```

Kernel Execution

```
__host__
int vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    ...
    dim3 DimGrid((n-1)/256+1, 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
    ...
}

__global__
void vecAddKernel(float *A, float *B, float *C, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i<n)   C[i] = A[i]+B[i];
}
```

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
  - kernel function must return void
- __device__ and __host__ can be used together
- __host__ is optional if used alone

Compiling a CUDA Program