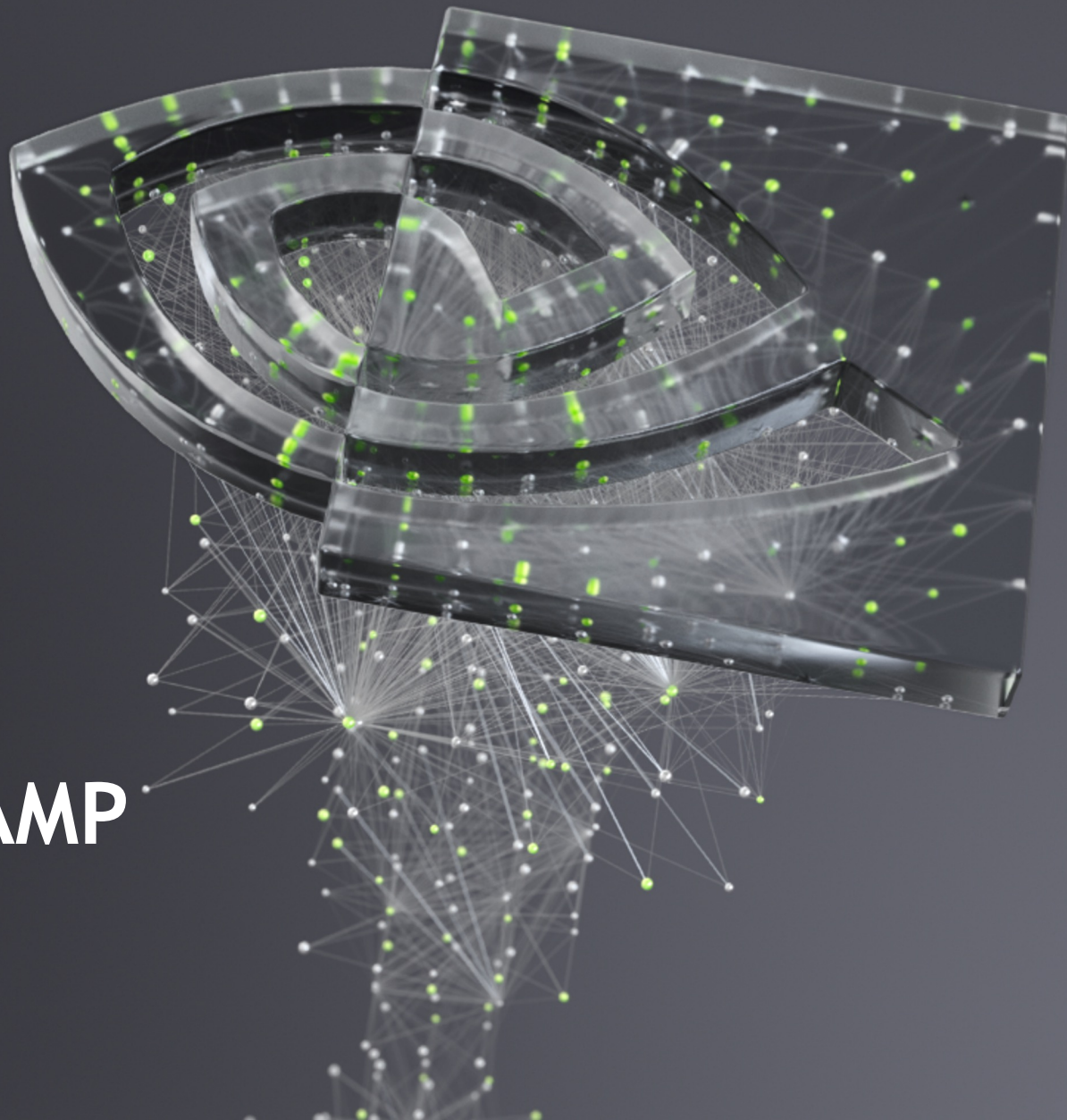




# N-WAYS GPU BOOTCAMP

CUDA C/FORTRAN

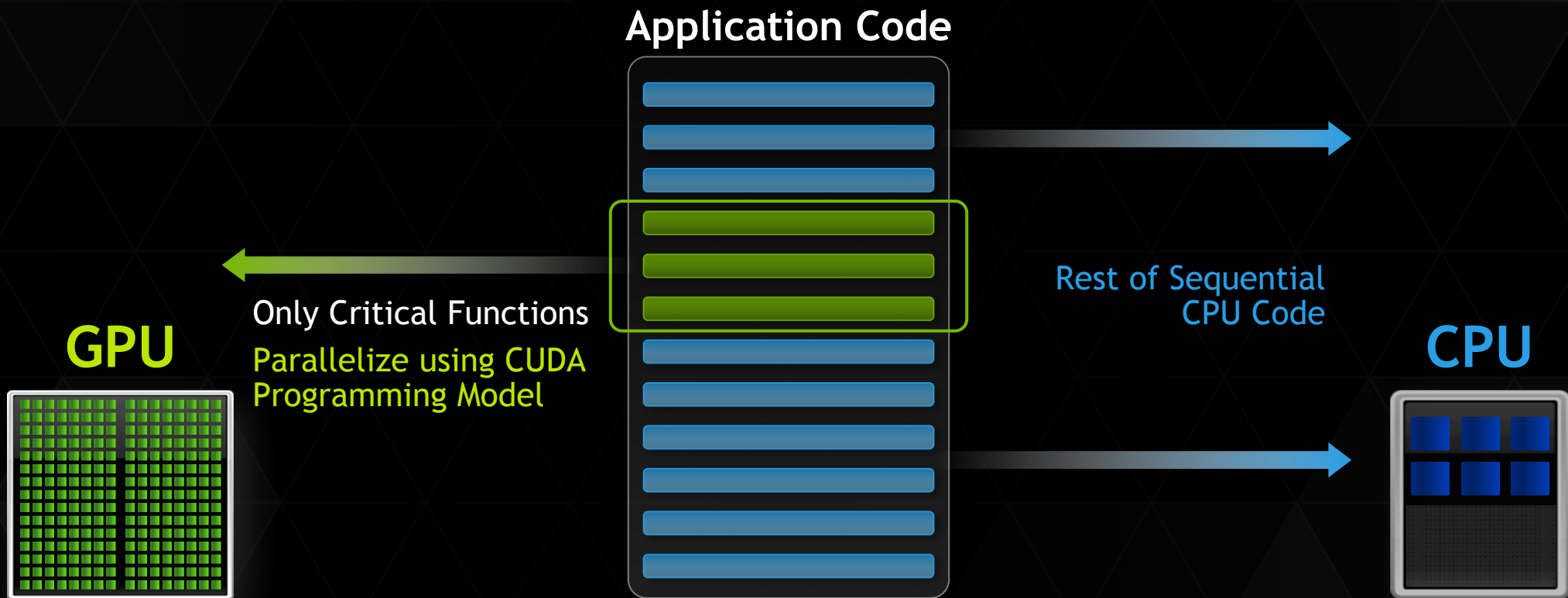


# CUDA C/FORTRAN

## What to expect?

- Basic introduction to GPU Architecture
- GPU Memory and Programming Model
- CUDA C/Fortran programming

# GPU COMPUTING





# CUDA ARCHITECTURE PROGRAMMING MODEL

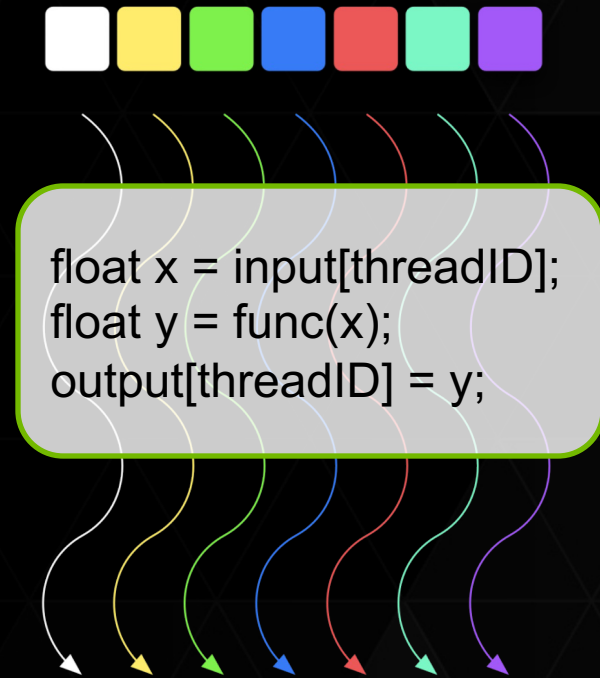
# CUDA KERNELS

- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads
- CUDA threads:
  - Lightweight
  - Fast switching
  - Tens of thousands execute simultaneously

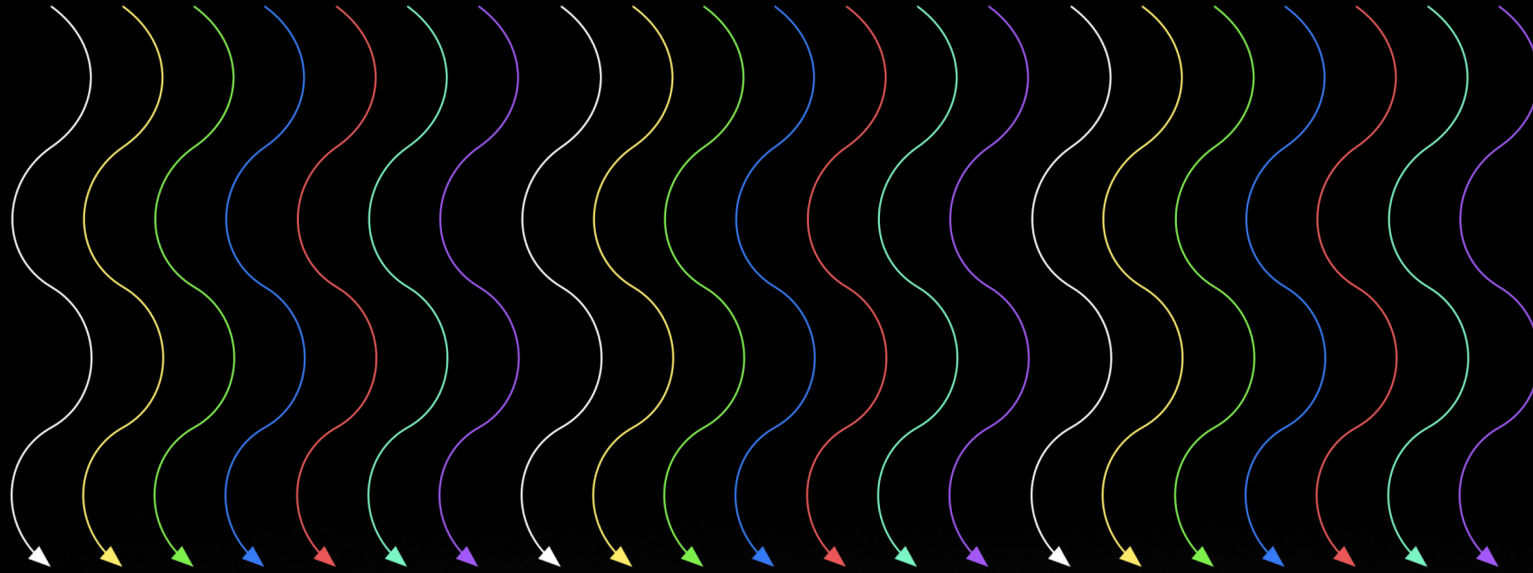
CPU	Host	Executes functions
GPU	Device	Executes kernels

# CUDA KERNELS: PARALLEL THREADS

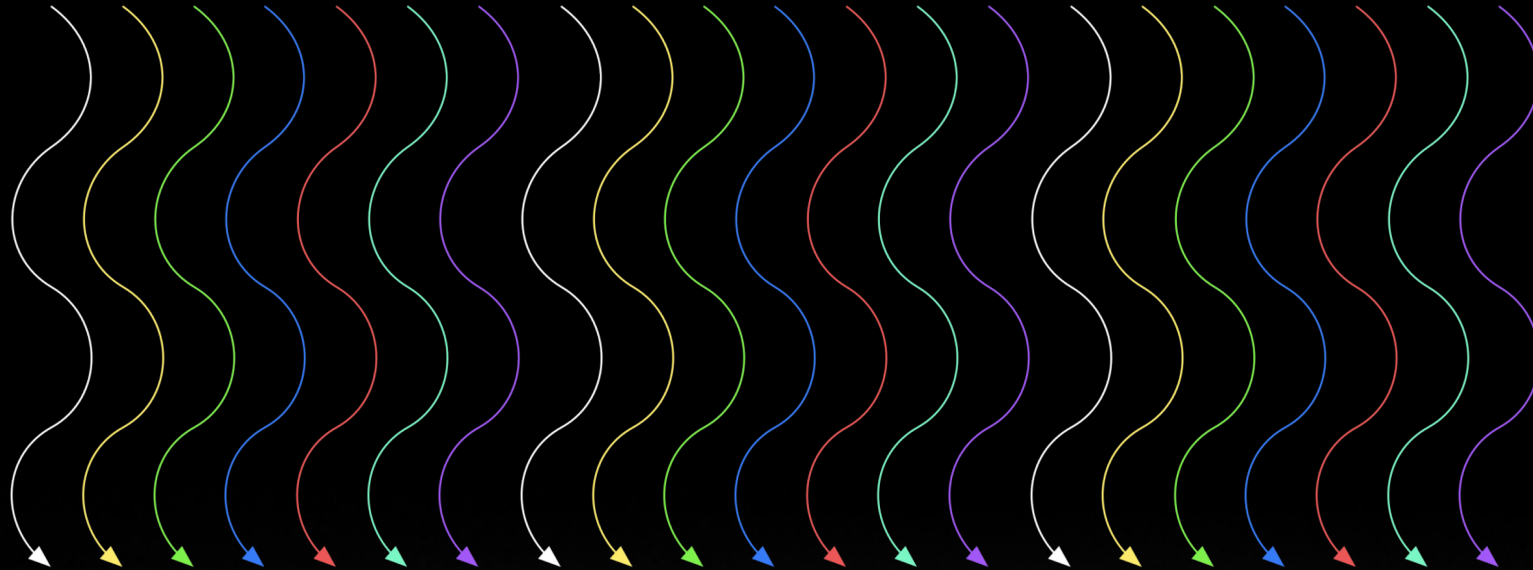
- A **kernel** is a function executed on the GPU
  - Array of threads, in parallel
- All threads execute the same code, can take different paths
  - Each thread has an ID
  - Select input/output data
  - Control decisions



# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



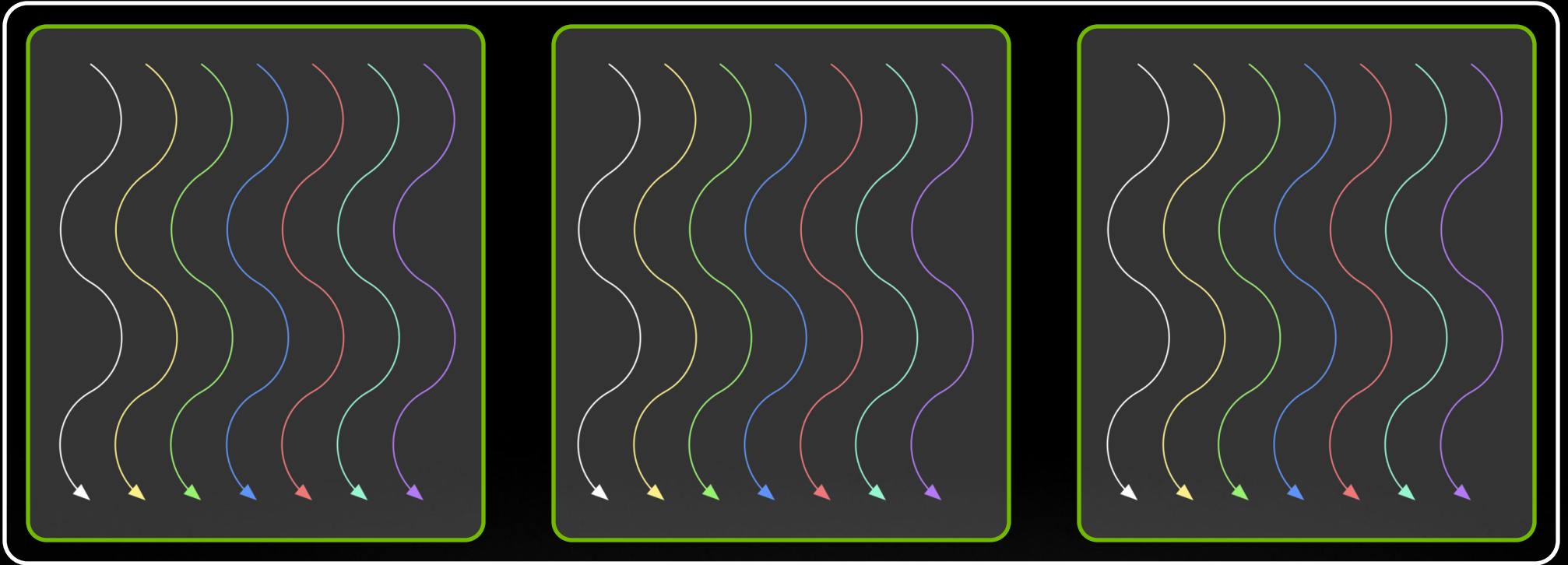
# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



- Threads are grouped into **blocks**

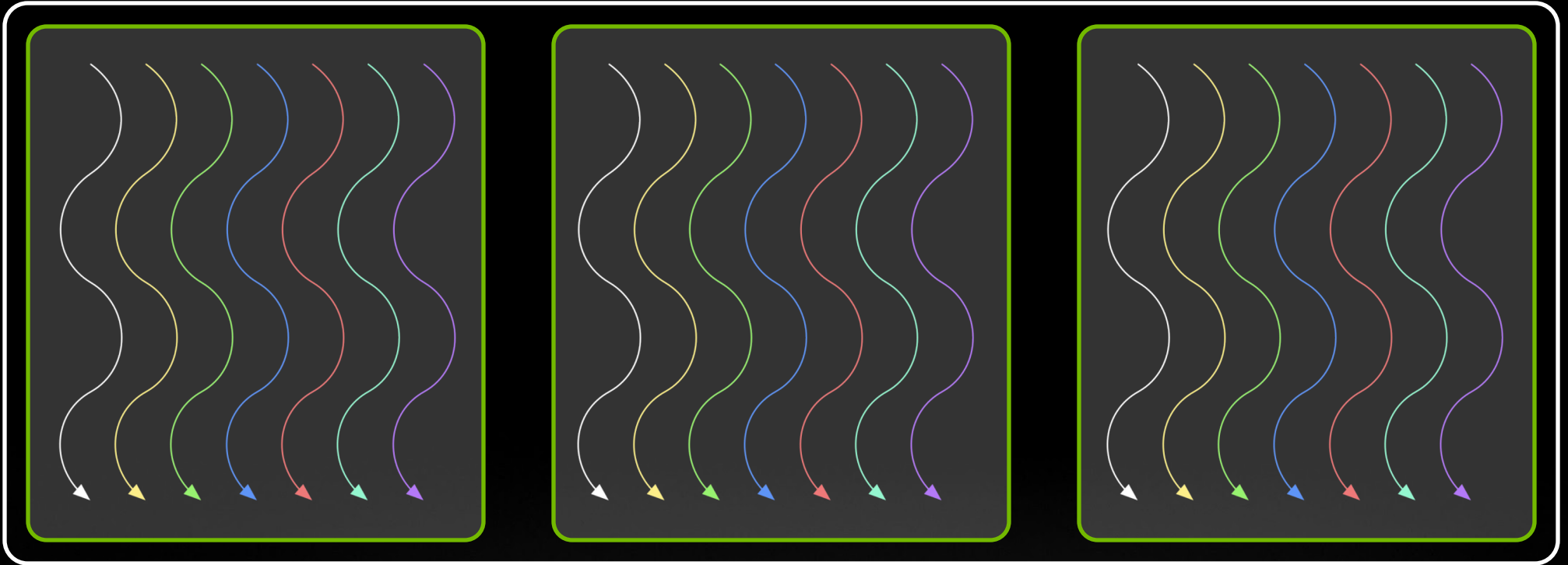


# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



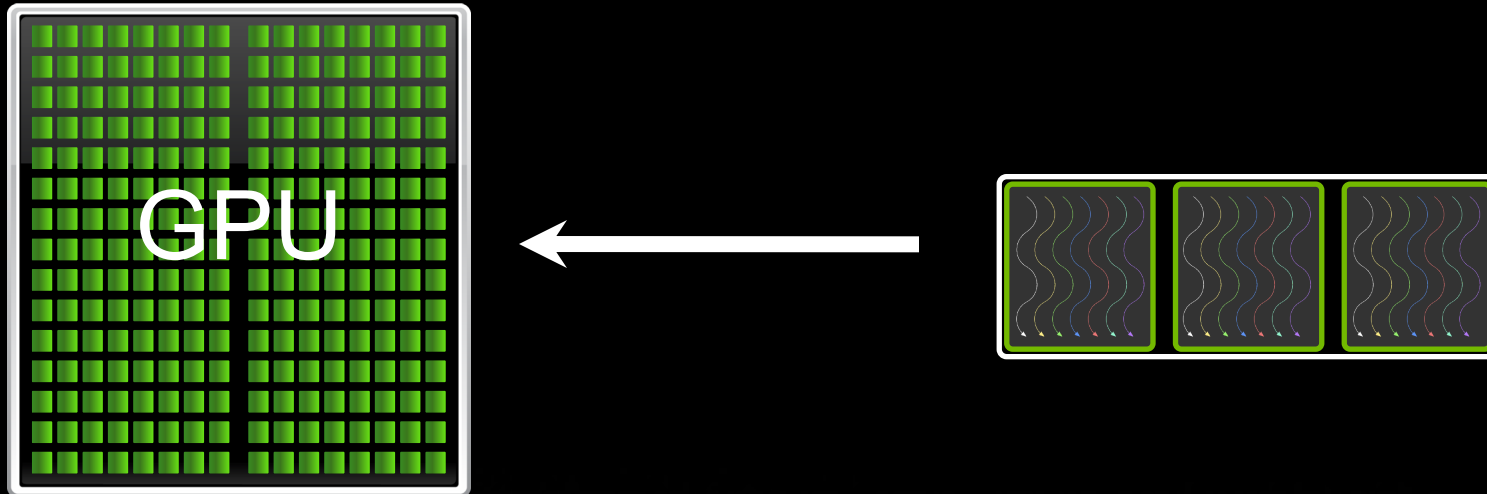
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**

# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



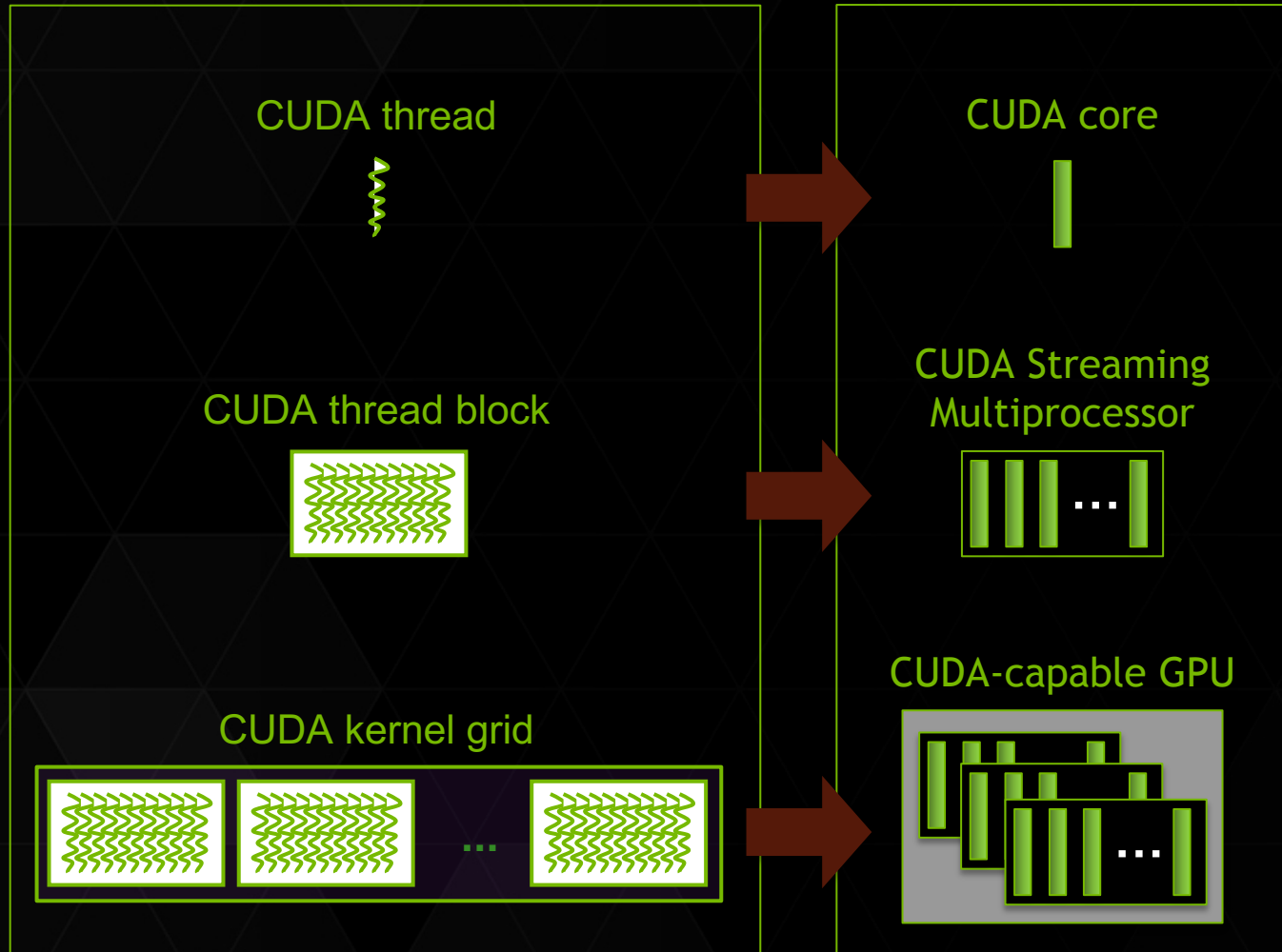
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**
- A **kernel** is executed as a **grid of blocks of threads**

# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



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

# KERNEL EXECUTION

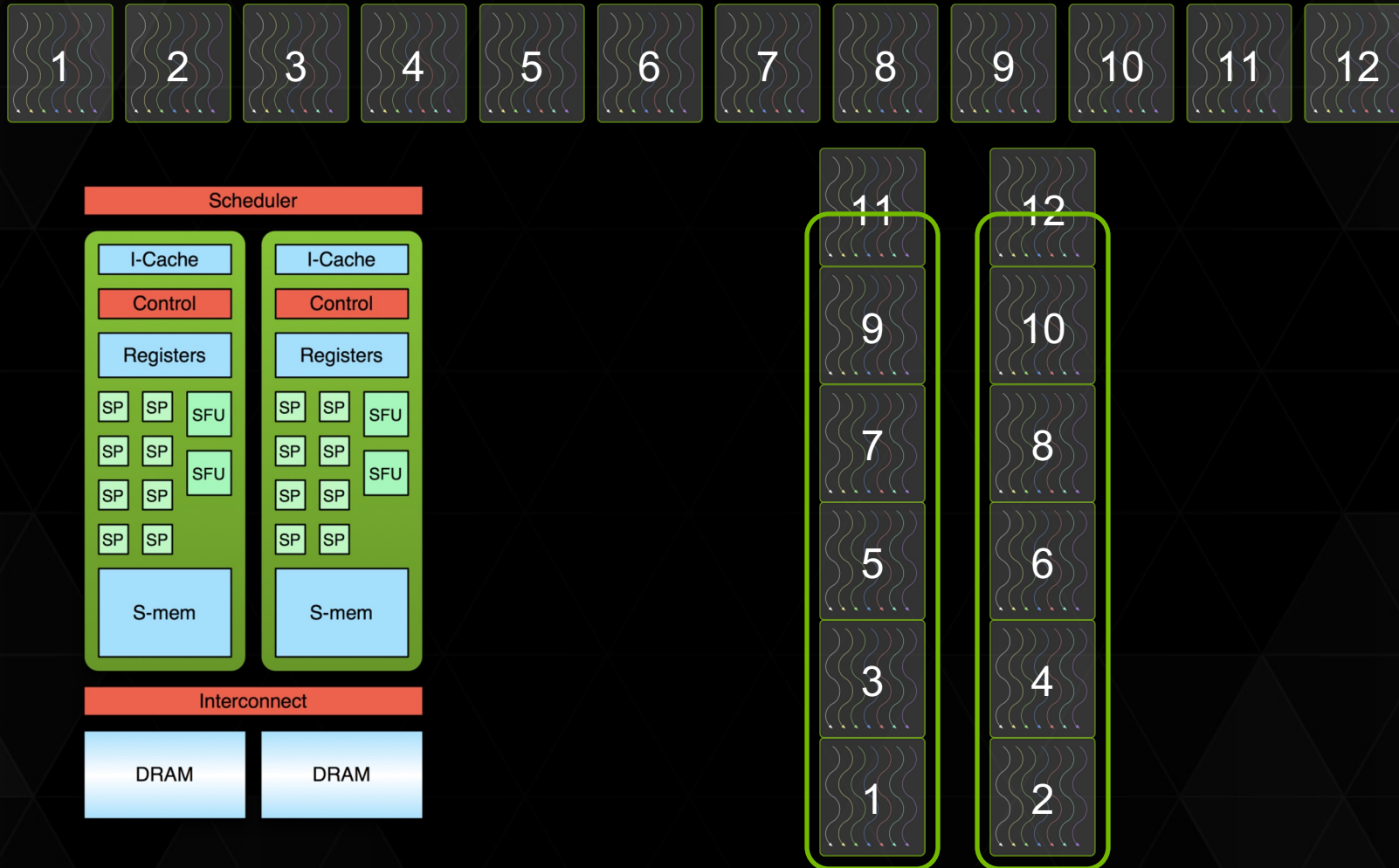


- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

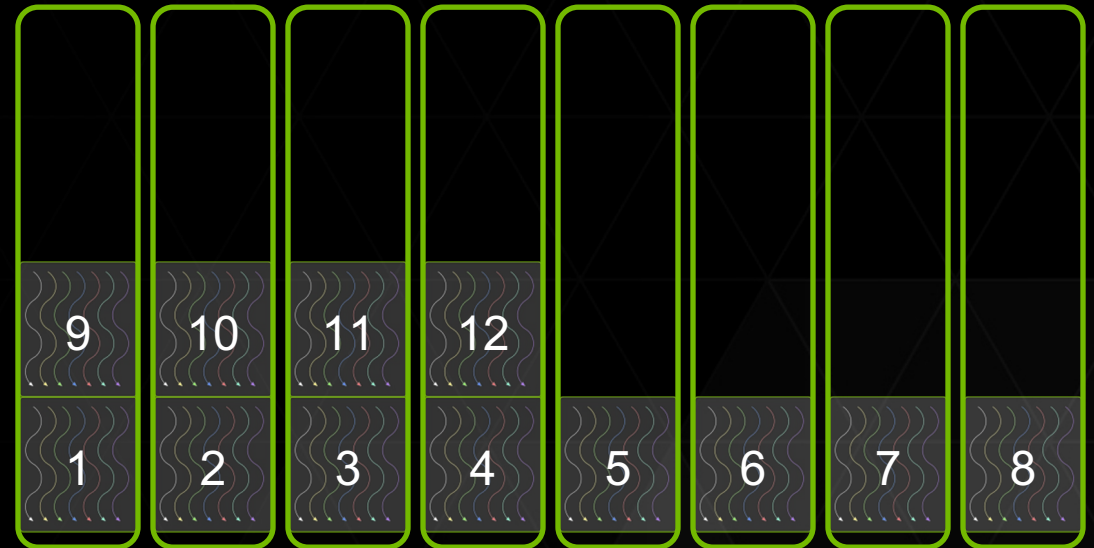
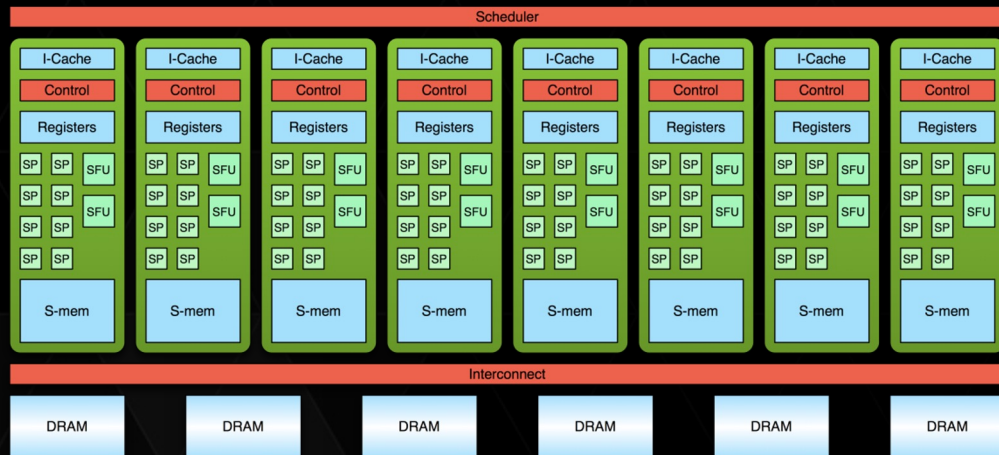
# COMMUNICATION WITHIN A BLOCK

- Threads may need to cooperate
  - Memory accesses
  - Share results
- Cooperate using **shared memory**
  - Accessible by all threads within a block
- Restriction to “within a block” permits scalability
  - Fast communication between  $N$  threads is not feasible when  $N$  large

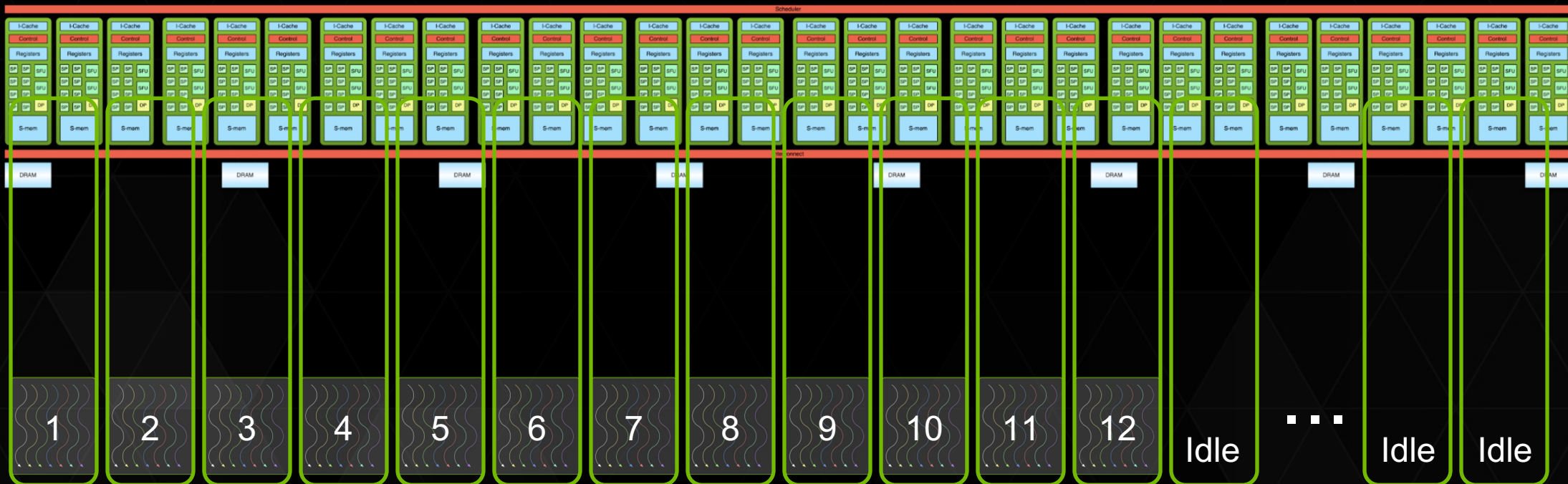
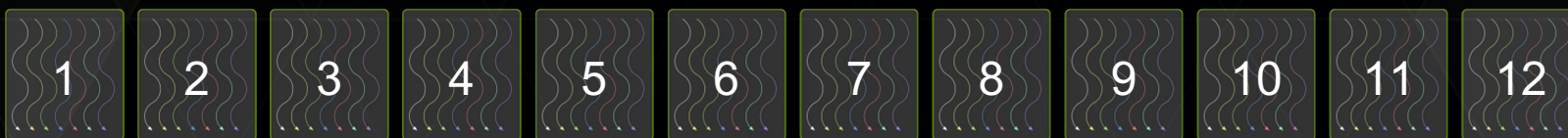
# TRANSPARENT SCALABILITY



# TRANSPARENT SCALABILITY



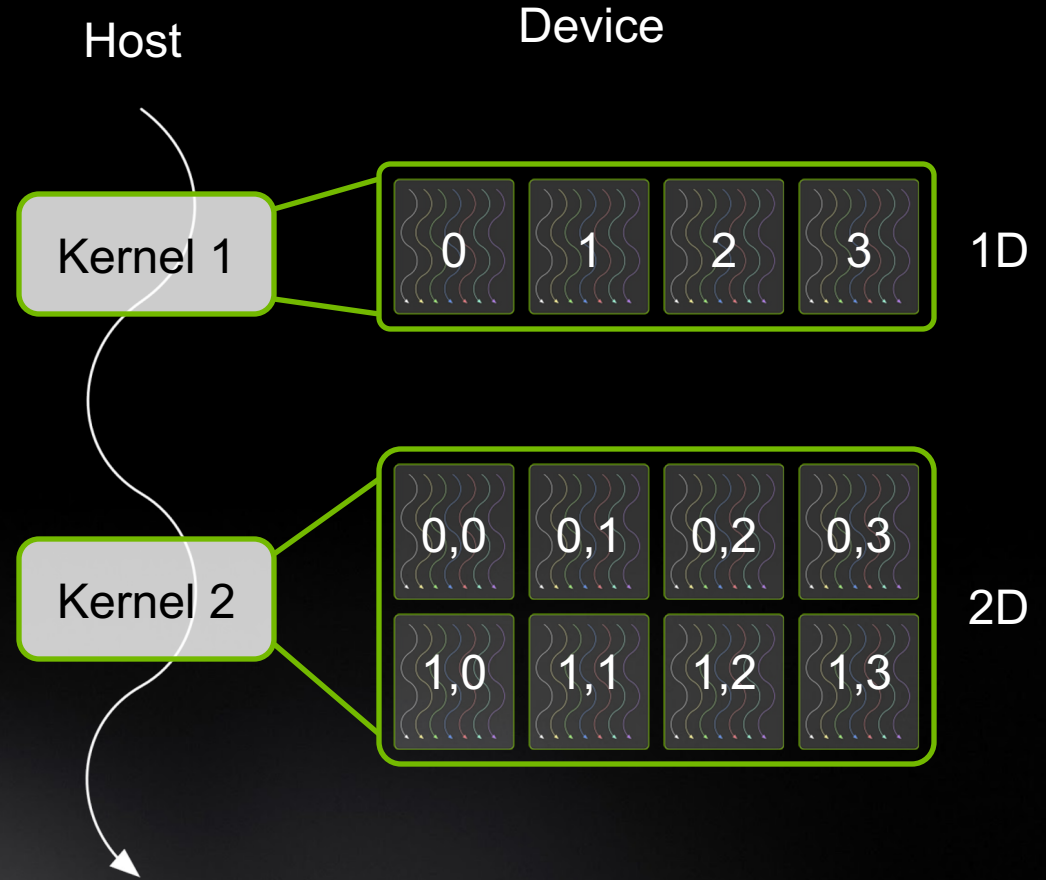
# TRANSPARENT SCALABILITY -





# CUDA PROGRAMMING MODEL - SUMMARY

- A kernel executes as a grid of thread blocks
- A block is a batch of threads
  - Communicate through shared memory
- Each block has a block ID
- Each thread has a thread ID





# CUDA ARCHITECTURE MEMORY MODEL

# GPU ARCHITECTURE

Two Main components

## Global memory

Analogous to RAM in a CPU server

Accessible by both GPU and CPU

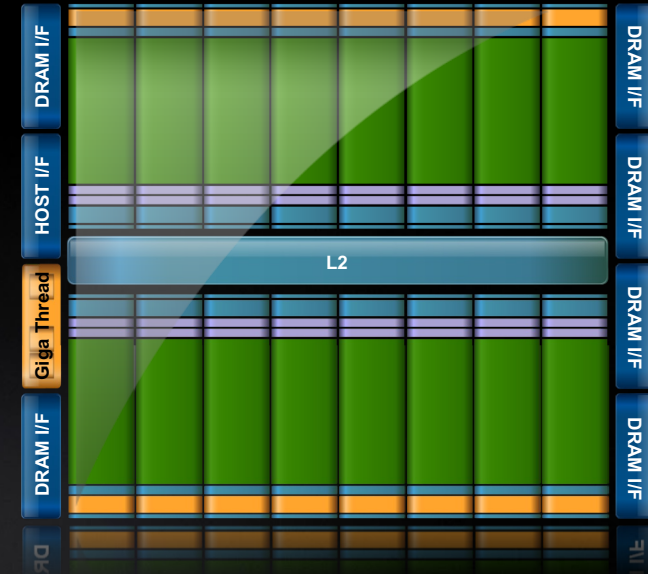
48GB with bandwidth currently up to 1 TB/s

## Streaming Multiprocessors (SMs)

SMs perform the actual computations

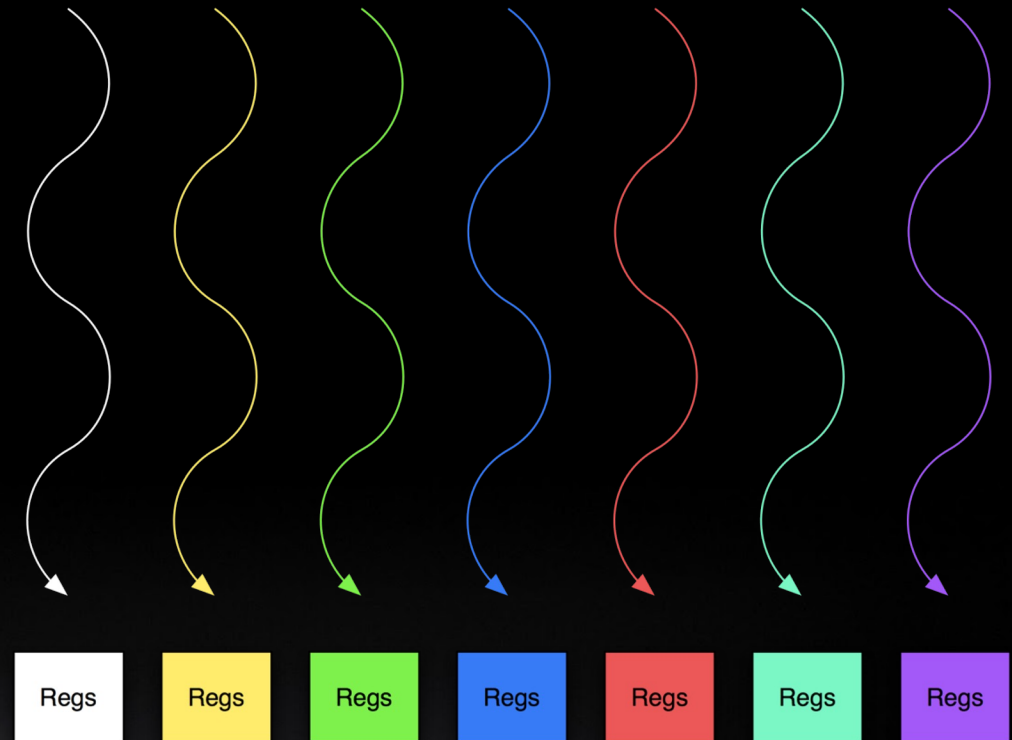
Each SM has its own:

Control units, registers, execution pipelines, caches



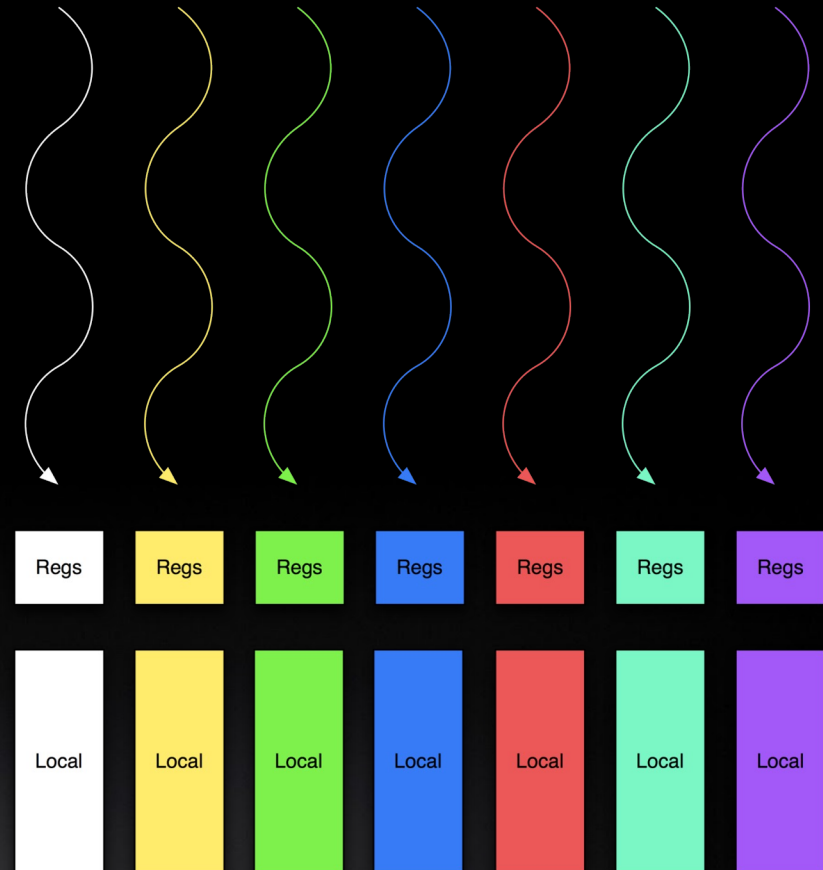
# MEMORY HIERARCHY

- Thread
  - Registers



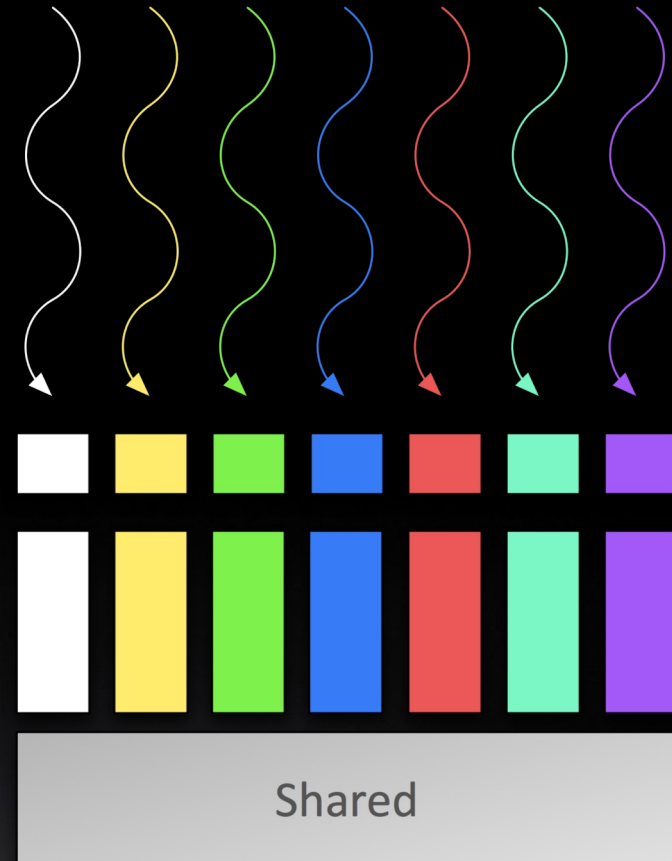
# MEMORY HIERARCHY

- Thread
  - Registers
- Thread
  - Local memory



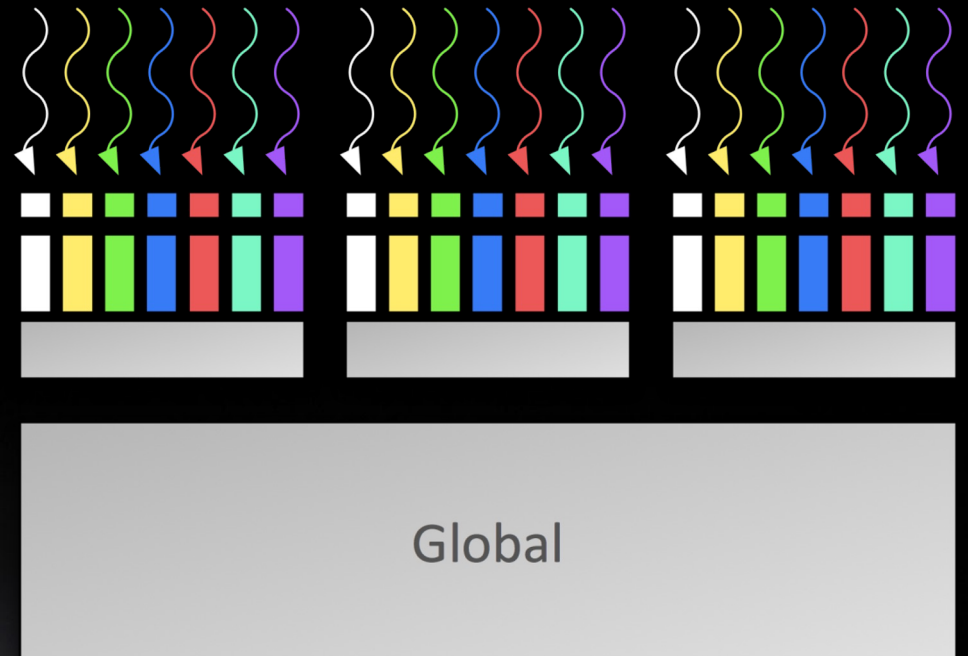
# MEMORY HIERARCHY

- Thread
  - Registers
- Thread
  - Local memory
- Block of threads
  - Shared memory



# MEMORY HIERARCHY

- Thread
  - Registers
- Thread
  - Local memory
- Block of threads
  - Shared memory
- All blocks
  - Global memory





# CUDA-C/Fortran INTRODUCTION



# WHAT IS CUDA?

- CUDA Architecture

- Expose general-purpose GPU computing as first-class capability
- Retain traditional DirectX/OpenGL graphics performance

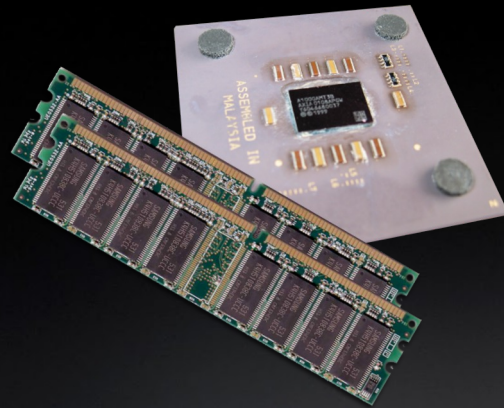
- CUDA C/Fortran

- Based on industry-standard C/Fortran
- A handful of language extensions to allow heterogeneous programs
- Straightforward APIs to manage devices, memory, etc.

# CUDA C/FORTRAN: THE BASICS

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

Host



Device



# HELLO, WORLD!

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

```
program main  
    implicit none  
    print *, "Hello World"  
end program main
```

- This basic program is just standard C/Fortran that runs on the *host*
- NVIDIA's compiler (`nvcc/nvfortran`) will not complain about CUDA programs with no *device* code
- At its simplest, CUDA C/Fortran is just C/Fortran!

# HELLO, WORLD! WITH DEVICE CODE

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

```
module printgpu  
contains  
    attributes(global) subroutine print_form_gpu()  
        implicit none  
    end subroutine print_form_gpu  
end module printgpu  
  
program testPrint  
    use printgpu  
    use cudafor  
    implicit none  
  
    call print_form_gpu<<<1, 1>>>()  
  
end program testPrint
```

- Two notable additions to the original “Hello, World!”

# NVIDIA HPC SDK

- Comprehensive suite of compilers, libraries, and tools used to GPU accelerate HPC modeling and simulation application
- The NVIDIA HPC SDK includes the new NVIDIA HPC compiler supporting CUDA C and Fortran
  - The command to compile CUDA C/C++ code is `'nvcc'`
  - The command to compile CUDA Fortran code is `'nvfortran'`

```
nvcc main.cu
```

```
nvfortran -cuda main.f90
```

# HELLO, WORLD! WITH DEVICE CODE

```
__global__ void kernel( void ) {  
}
```

```
attributes(global) subroutine print_form_gpu()  
end subroutine print_form_gpu
```

- Keyword `__global__` in C and attribute `global` in Fortran indicates that a function
  - Runs on the device
  - Called from host code
- `nvcc/nvfortran` splits source file into host and device components
  - NVIDIA's compiler handles device functions like `kernel()`
  - Standard host compiler handles host functions like `main()`
    - `gcc`
    - Microsoft Visual C

# HELLO, WORLD! WITH DEVICE CODE

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

```
program testPrint  
    use printgpu  
    use cudafor  
    implicit none  
  
    call print_form_gpu<<<1, 1>>>()  
  
end program testPrint
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Sometimes called a “kernel launch”
  - We’ll discuss the parameters inside the angle brackets later
- This is all that’s required to execute a function on the GPU!

# A MORE COMPLEX EXAMPLE

- A simple kernel to add two integers:
- As before, `__global__` is a CUDA keyword meaning
  - `add()` will execute on the device ... so `a`, `b`, and `c` must point to device memory
  - How do we allocate memory on the GPU?
  - `add()` will be called from the host

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

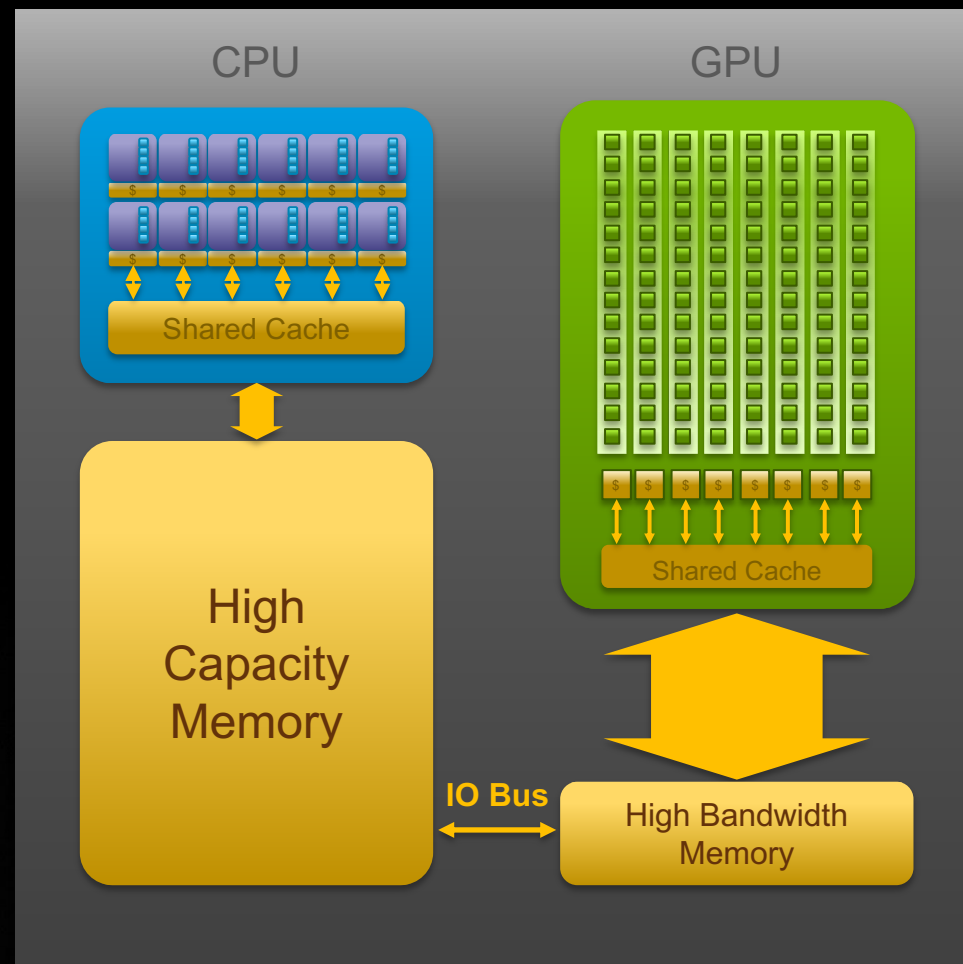
```
attributes(global) subroutine add( a, b, c )  
    int, device :: a(1), b(1), c(1)  
  
    c(1) = a(1) + b(1)  
  
end subroutine vecAdd_kernel
```



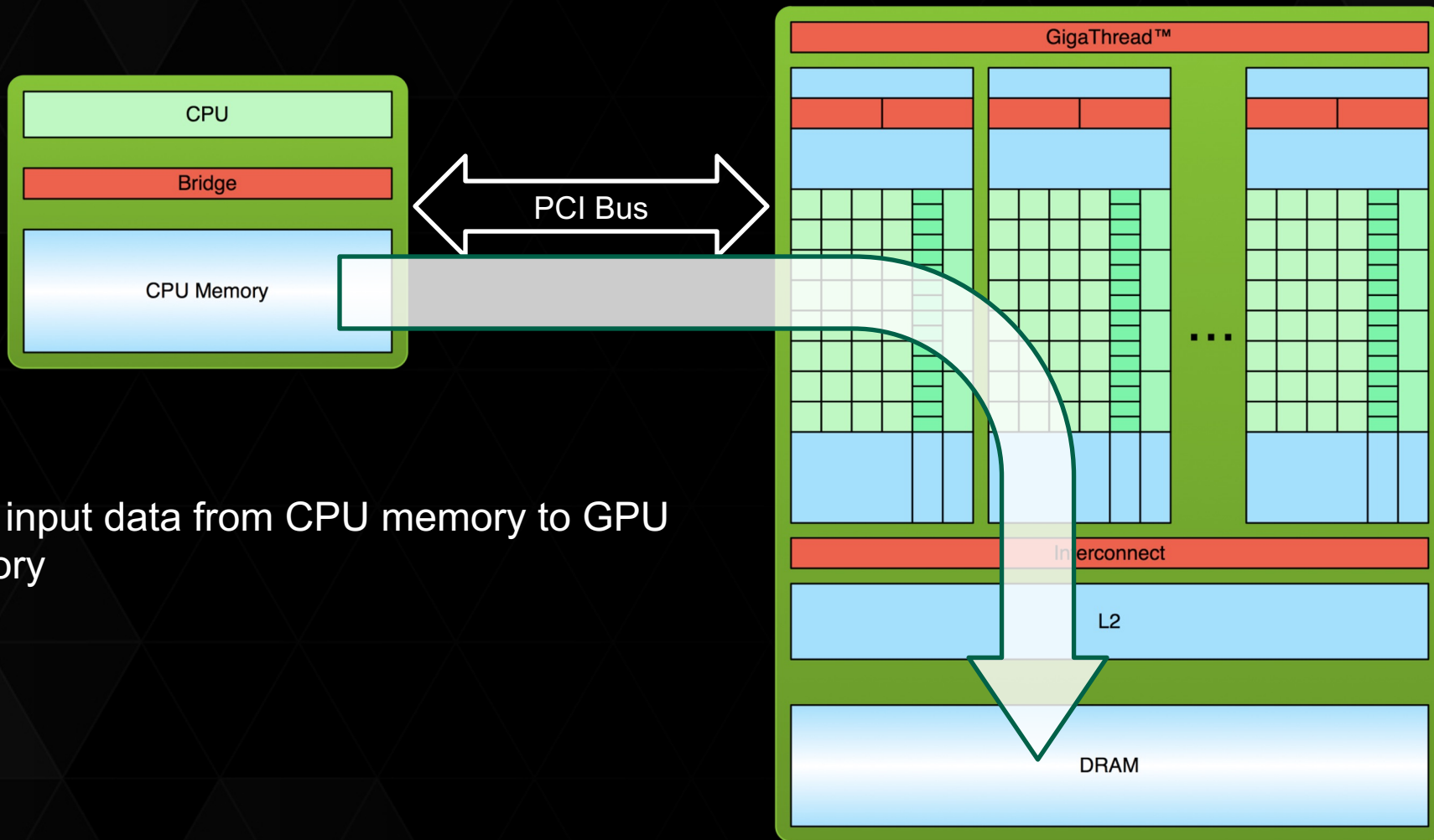
# CPU + GPU

## Physical Diagram

- CPU memory is larger, GPU memory has more bandwidth
- CPU and GPU memory are usually separate, connected by an I/O bus (traditionally PCIe)
- Any data transferred between the CPU and GPU will be handled by the I/O Bus
- The I/O Bus is relatively slow compared to memory bandwidth
- The GPU cannot perform computation until the data is within its memory

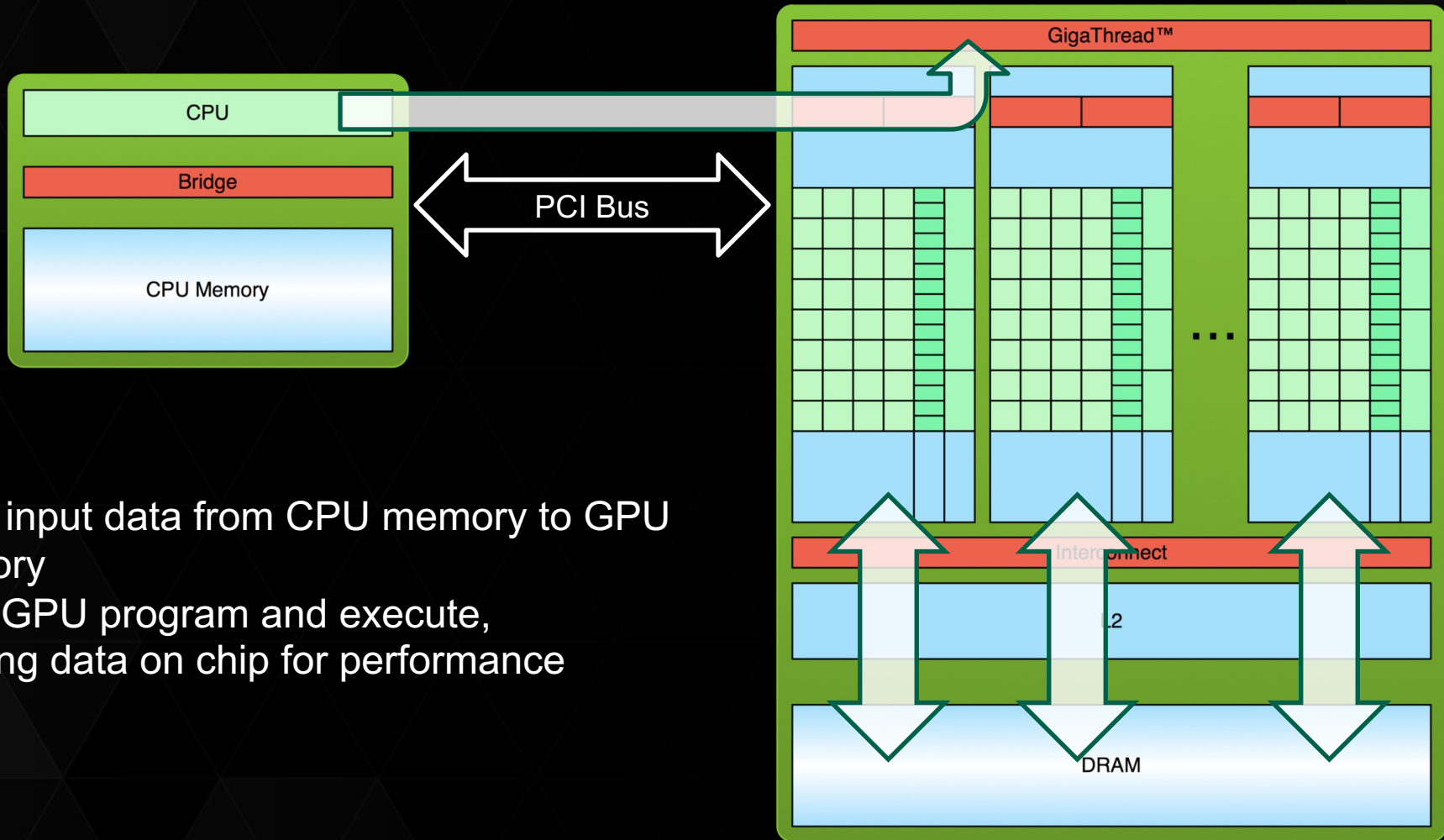


# PROCESSING FLOW - STEP 1



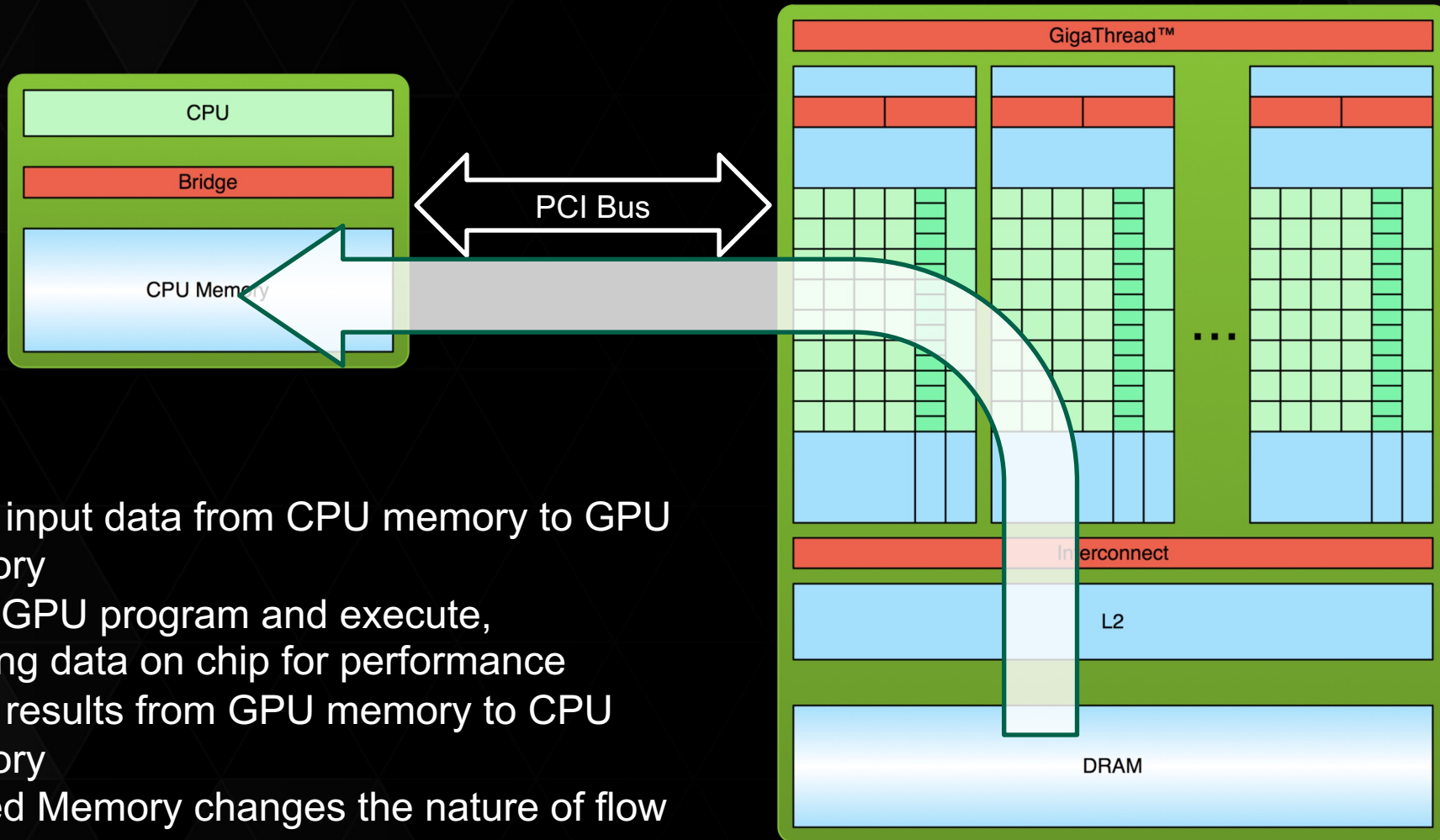
1. Copy input data from CPU memory to GPU memory

# PROCESSING FLOW - STEP 2



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

# PROCESSING FLOW - STEP 3



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
4. Unified Memory changes the nature of flow
  - Some of the basics remains same

# MEMORY MANAGEMENT

- Host and device memory are distinct entities
- Basic CUDA API for dealing with explicit device memory management
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to their C equivalents, `malloc()`, `free()`, `memcpy()`
  - Similar to their Fortran equivalents, `allocate()`, `deallocate`
    - Array should be defined as allocatable
- CUDA API for using Unified memory is
  - C API: `cudaMallocManaged()`, `cudaFree()`
  - Fortran: Declare variable with `managed`, `allocatable` attribute
    - `real, managed, allocatable, dimension(:,:) :: A, B, C`
  - \* For this session we will be making use of Unified Memory

# A MORE COMPLEX EXAMPLE: MAIN ( )

```
int main( void ) {
    int *a, *b, *c;
    int size = sizeof( int );

    cudaMallocManaged( &a, size );
    cudaMallocManaged( &b, size );
    cudaMallocManaged( &c, size );

    add<<< 1, 1 >>>( a, b, c );

    cudaFree( a ); cudaFree( b );
    cudaFree( c );

    return 0;
}
```

```
program main
    use cudafor

    real, managed, allocatable, dimension(:,:) ::
a, b, c

    allocate(a(n))
    allocate(b(n))
    allocate(c(n))

    call add<<<1, 1>>>(n, a, b, c)

    deallocate(d_a)
    deallocate(d_b)
    deallocate(d_c)

end program main
```

# PARALLEL PROGRAMMING IN CUDA

- But wait...GPU computing is about massive parallelism
- So how do we run code in parallel on the device?
- Solution lies in the parameters between the triple angle brackets:

```
add<<< 1, 1 >>>( a, b, c );
```



```
add<<< N, 1 >>>( a, b, c );
```

- Instead of executing add() once, add() executed N times in parallel

# PARALLEL PROGRAMMING IN CUDA

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



# PARALLEL PROGRAMMING IN CUDA

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

```
attributes(global) subroutine add(n, a, b, c)
    integer, value :: n
    real(8), device :: a(n), b(n), c(n)

    c(blockIdx%x) = a(blockIdx%x) +
b(blockIdx%x)

    end subroutine add
}
```

Block 0

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

Block 1

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

Block 2

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

Block 3

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

This is what runs in parallel on the device

# THREADS

- Terminology: A block can be split into parallel *threads*
- Let's change vector addition to use parallel threads instead of parallel blocks:
- We use `threadIdx.x` instead of `blockIdx.x` in `add()`

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

```
attributes(global) subroutine add(n, a, b, c)
    integer, value :: n
    real(8), device :: a(n), b(n), c(n)

    c(threadidx%x) = a(threadidx%x) +
b(threadidx%x)

end subroutine add
```

```
add<<<N,1>>>( a, b, c )
```



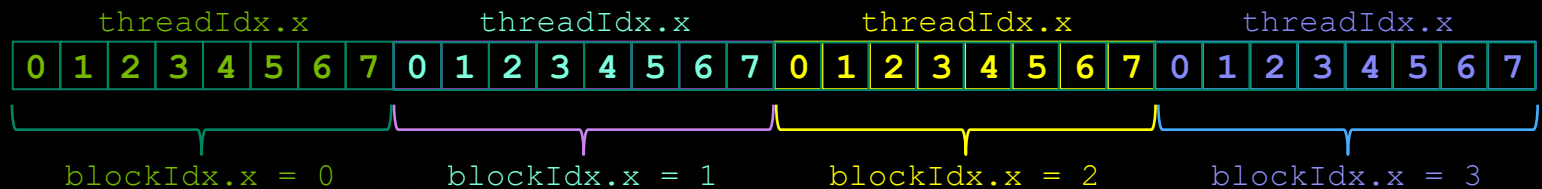
```
add<<<1,N>>>( a, b, c );
```

# USING THREADS AND BLOCKS

- We've seen parallel vector addition using
  - Many blocks with 1 thread apiece
  - 1 block with many threads
- Let's adapt vector addition to use lots of *both* blocks and threads
- After using threads and blocks together, we'll talk about *why* threads
- First let's discuss data indexing...

# INDEXING ARRAYS WITH THREADS AND BLOCKS

- No longer as simple as just using `threadIdx.x` or `blockIdx.x` as indices
- To index array with 1 thread per entry (using 8 threads/block)



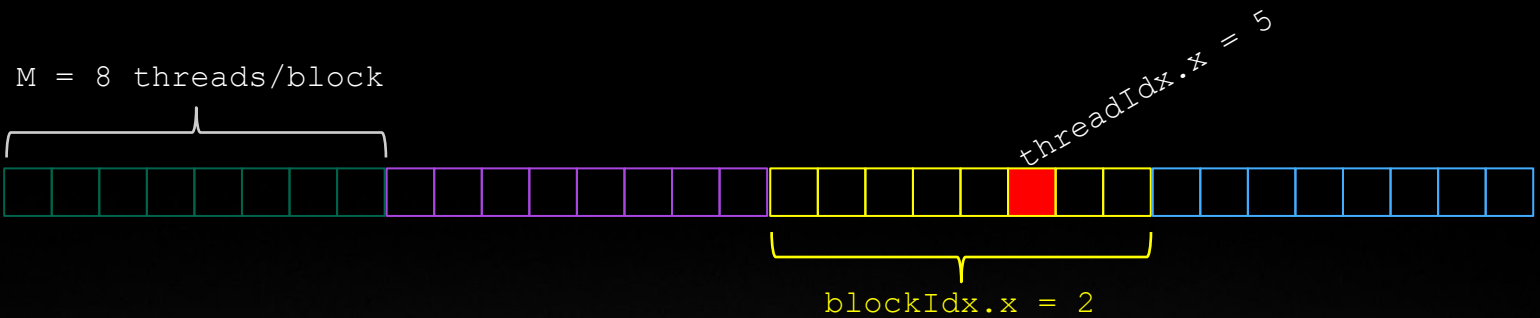
- If we have  $M$  threads/block, a unique array index for each entry given by

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

The diagram shows the mapping between the C++ code and the simplified code. In the first line, `threadIdx.x` is mapped to `x`, `blockIdx.x` is mapped to `y`, and `M` is mapped to `width`. Red arrows point from the code to the simplified code.

# INDEXING ARRAYS: EXAMPLE

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



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

# ADDITION WITH THREADS AND BLOCKS

- The `blockDim.x` is a built-in variable for threads per block:

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

- So what changes in `main()` when we use both blocks and threads?

```
__global__ void add( int *a, int *b, int *c )
{
    int index = threadIdx.x + blockIdx.x
* blockDim.x;

    c[index] = a[index] + b[index];
}
```

```
attributes(global) subroutine add(n, a, b, c)

    integer, value :: n
    real(8), device :: a(n), b(n), c(n)
    integer :: id

    id = (blockidx%x-1)*blockdim%x +
threadidx%x

    c(id) = a(id) + b(id)

end subroutine add
```

Note we used `(blockidx%x-1)` for Fortran as Fortran numbering starts from 1

# PARALLEL ADDITION (BLOCKS/THREADS): MAIN ()

```
#define N (2048*2048)

#define THREADS_PER_BLOCK 512

add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( a, b, c );
```

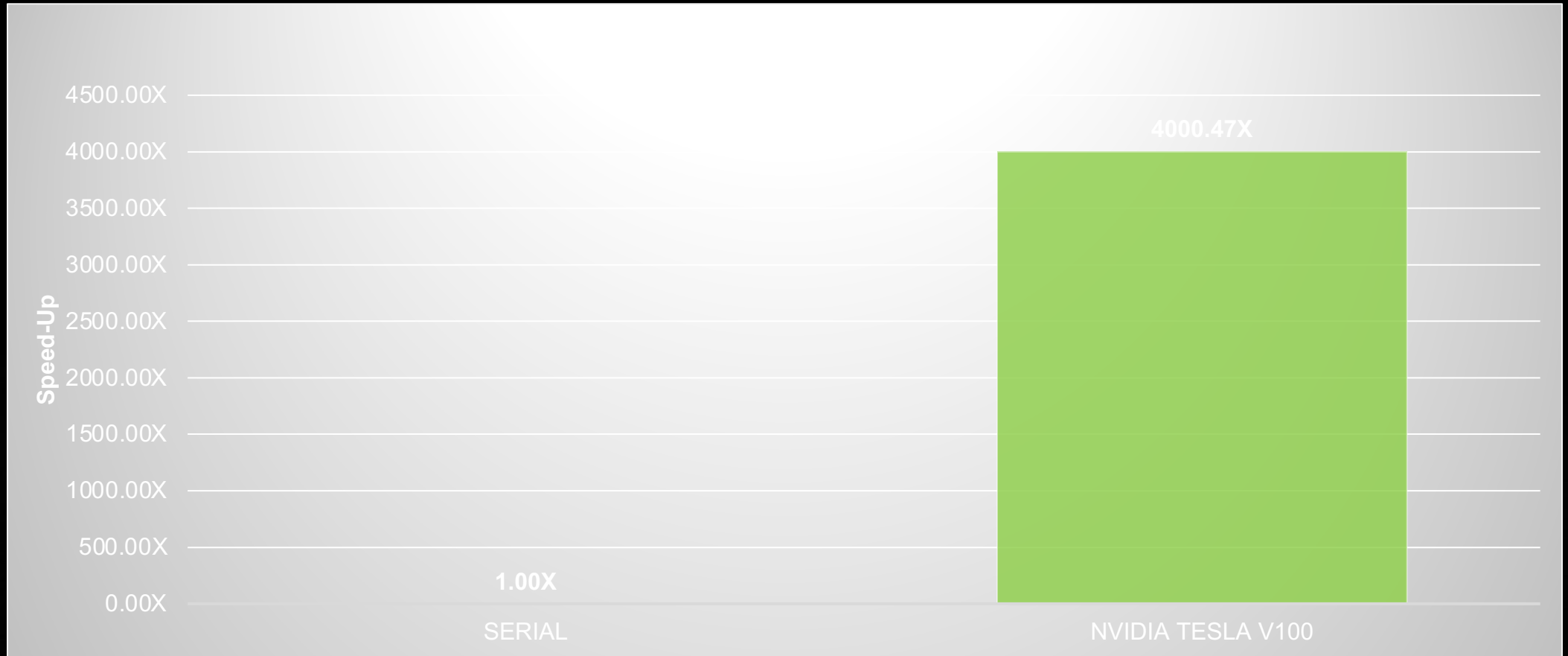
```
blockSize = dim3(512,1,1)

! Number of thread blocks in grid

gridSize = dim3(ceiling(real(n)/real(blockSize%x)) ,1,1)

call add<<<gridSize, blockSize>>>(n, d_a, d_b, d_c)
```

# CUDA SPEEDUP



*HPC SDK 20.11, NVIDIA Tesla V100, DGX1*



# REFERENCES

<https://developer.nvidia.com/hpc-sdk>



**THANK YOU**

