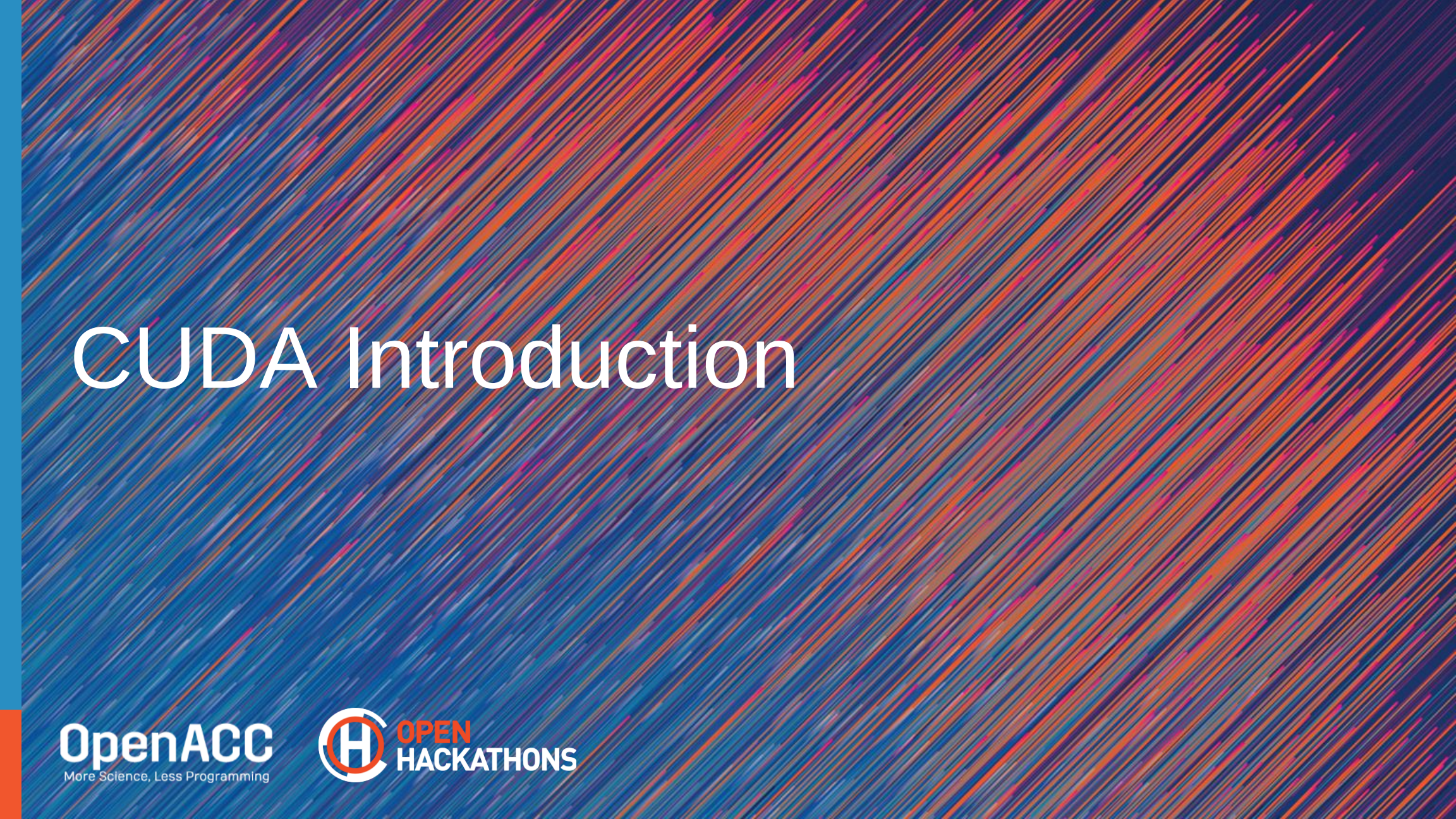
# N-WAYS GPU BOOTCAMP
## CUDA C/FORTRAN

OpenACC
More Science, Less Programming

OPEN HACKATHONS

# CUDA Introduction



# 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 |

# 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 NVIDIA HPC compiler supporting CUDA C and Fortran
  - The command to compile CUDA C code is ‘nvcc’
  - The command to compile C++ code is ‘nvc++’
  - The command to compile Fortran code is ‘nvfortran’

```
nvcc main.cu
```

```
nvfortran main.f90
```

# Hello, World!

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

```fortran
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 ) {
        printf("Hello from the GPU!");
}


int main( void ) {
        kernel<<<1,1>>>();
        cudaDeviceSynchronize();
        return 0;
}
```

```
module printgpu
contains
  attributes(global) subroutine print_from_gpu()
    implicit none
    print *, "Hello from the GPU!"
  end subroutine print_from_gpu
end module printgpu

program testPrint
  use printgpu
  use cudafor
  implicit none
  integer istat

  call print_from_gpu<<<1, 1>>>()
  istat = cudaDeviceSynchronize();

end program testPrint
```

Three notable additions to the original "Hello, World!"

# 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:

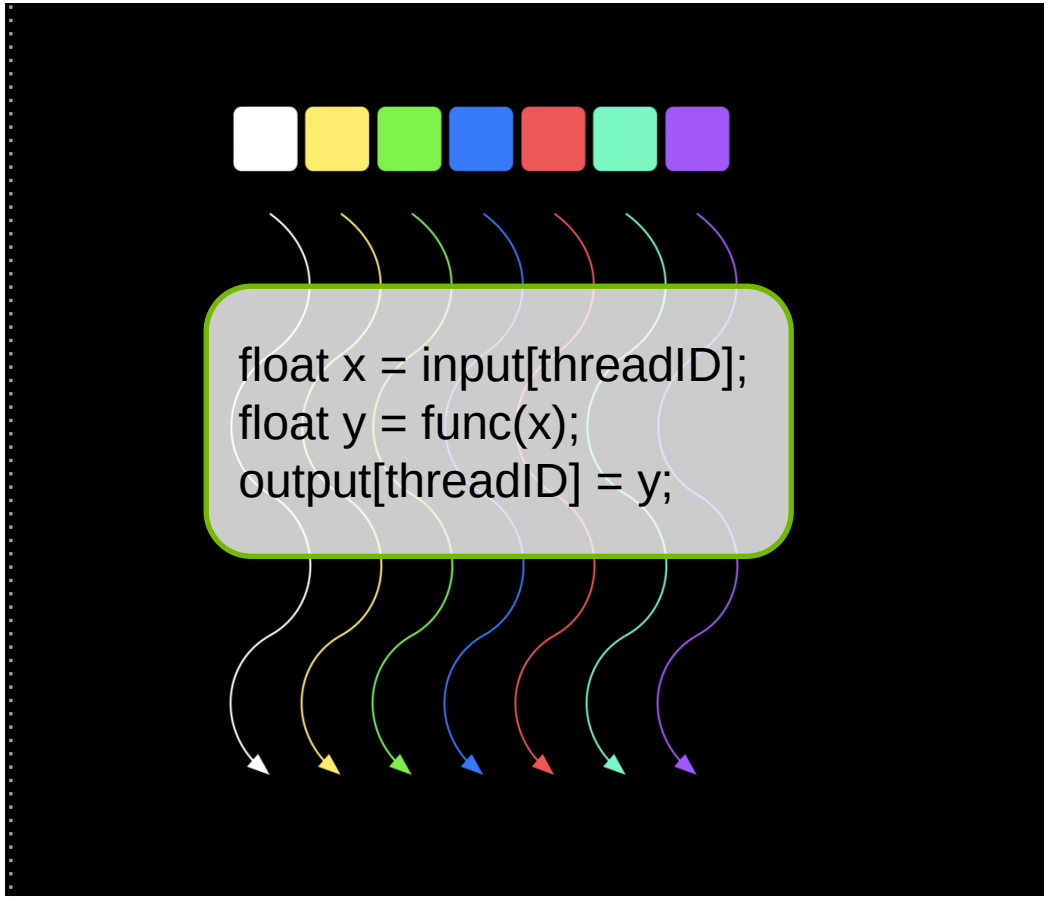
```
doSomethingOnce<<< 1, 1 >>>();
```

```
doSomethingLots<<< NumBlocks, NumThreads >>>();
```

- Instead of executing once, doSomethingLots() executes NumBlocks * NumThreads times, in parallel

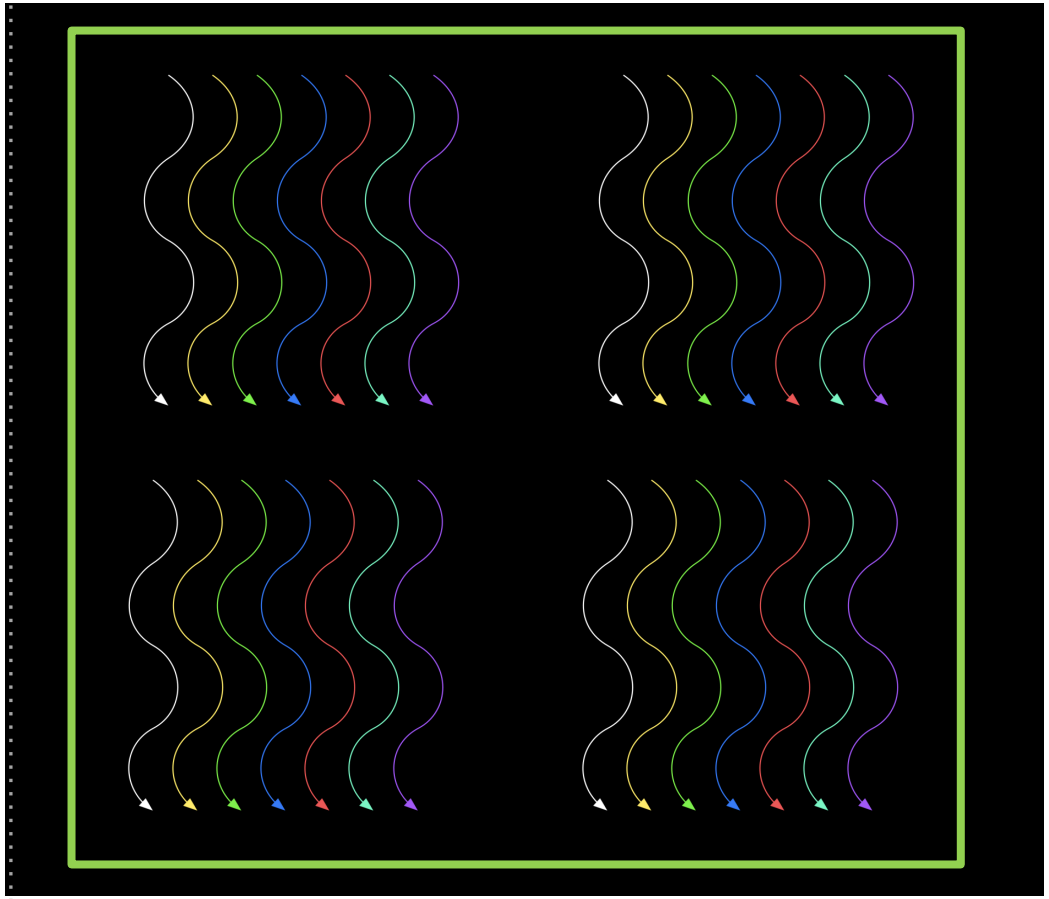
# CUDA KERNELS: PARALLEL THREADS

- All threads execute the same code, can take different paths (SIMT)
- A warp is a group of 32 threads
- Instructions are issued to warps rather than individual threads
- Can have non-multiple of 32 threads – but GPU is optimised for 32 threads in a warp
- The threads within a warp execute in lockstep – try to minimize branching if possible
- `doSomethingLots<<< 1, 32 >>>()` would launch a single warp, i.e. 32 threads
- Thus our kernel would execute 32 times



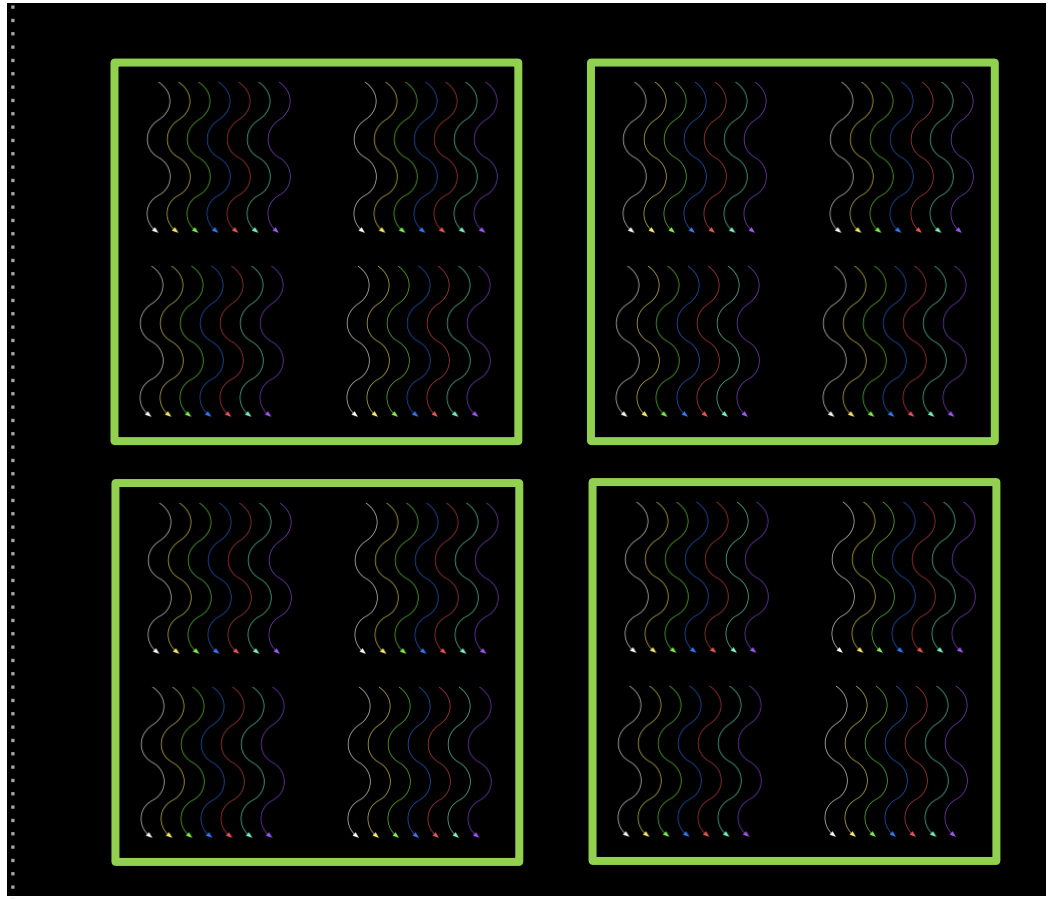
# CUDA KERNELS: BLOCKS

- A block is collection of warps (usually up to 32 warps - so up to 1024 threads in a block)
- When executing, a block resides on a particular streaming multiprocessor on the GPU – it will not migrate
- Enables local cooperation for threads within a block via shared memory (memory local to the SM)
- This permits scalability – fast communications between N threads is not feasible when N is large
- `doSomethingLots<<< 10, 1 >>>()` would launch 10 blocks, with 1 thread per block
- Our kernel would execute 10 times



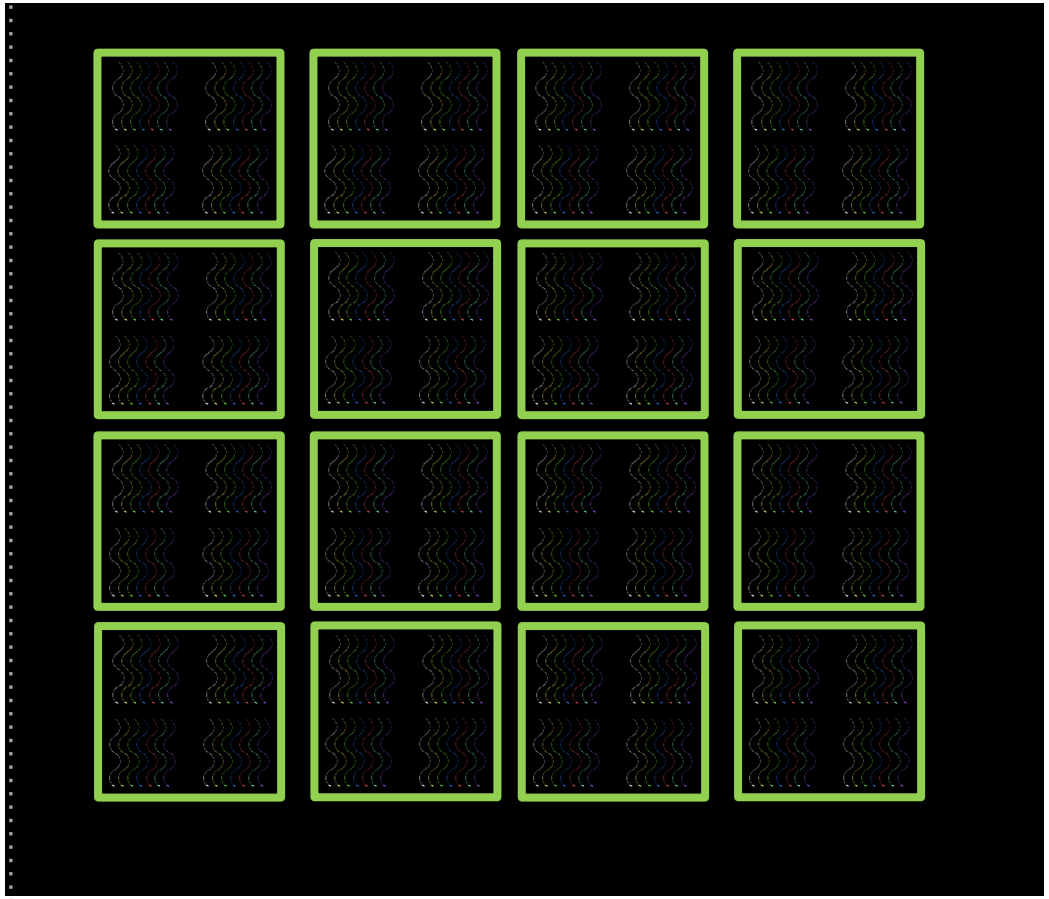
# CUDA KERNELS: GRIDS

- Blocks are grouped into a grid – this is the entirety of work that we want to get done by the kernel
- Grid is determined by the execution configuration – the angled brackets, triple chevrons – giving the number of blocks and the number of threads per block
- `doSomethingLots<<< 10, 256 >>>()` would launch 10 blocks, with 256 threads per block
- Our kernel would execute 2560 times



# CUDA KERNELS: DISTRIBUTION OF WORK

- We can now get our kernel to execute several times
- How to distribute the workload? One way would be to have each thread operate on one data element
- But how do we determine which thread, in which block, is currently executing? …



# CUDA-DEFINED VARIABLES FOR INDEXING

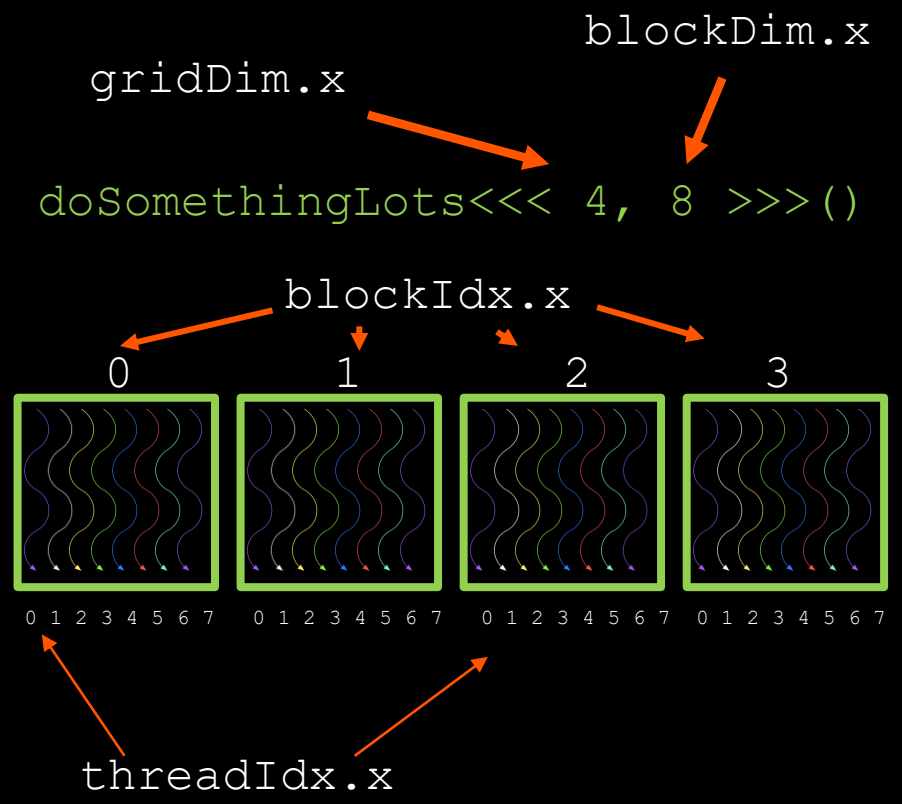
- gridDim.x - number of blocks in the grid
- blockIdx.x - index of the block in the grid (takes values from 0 to gridDim.x-1)
- blockDim.x - number of threads per block
- threadIdx.x - index of a thread within a block (0 to blockDim.x-1)

- Then we can define a global thread ID:

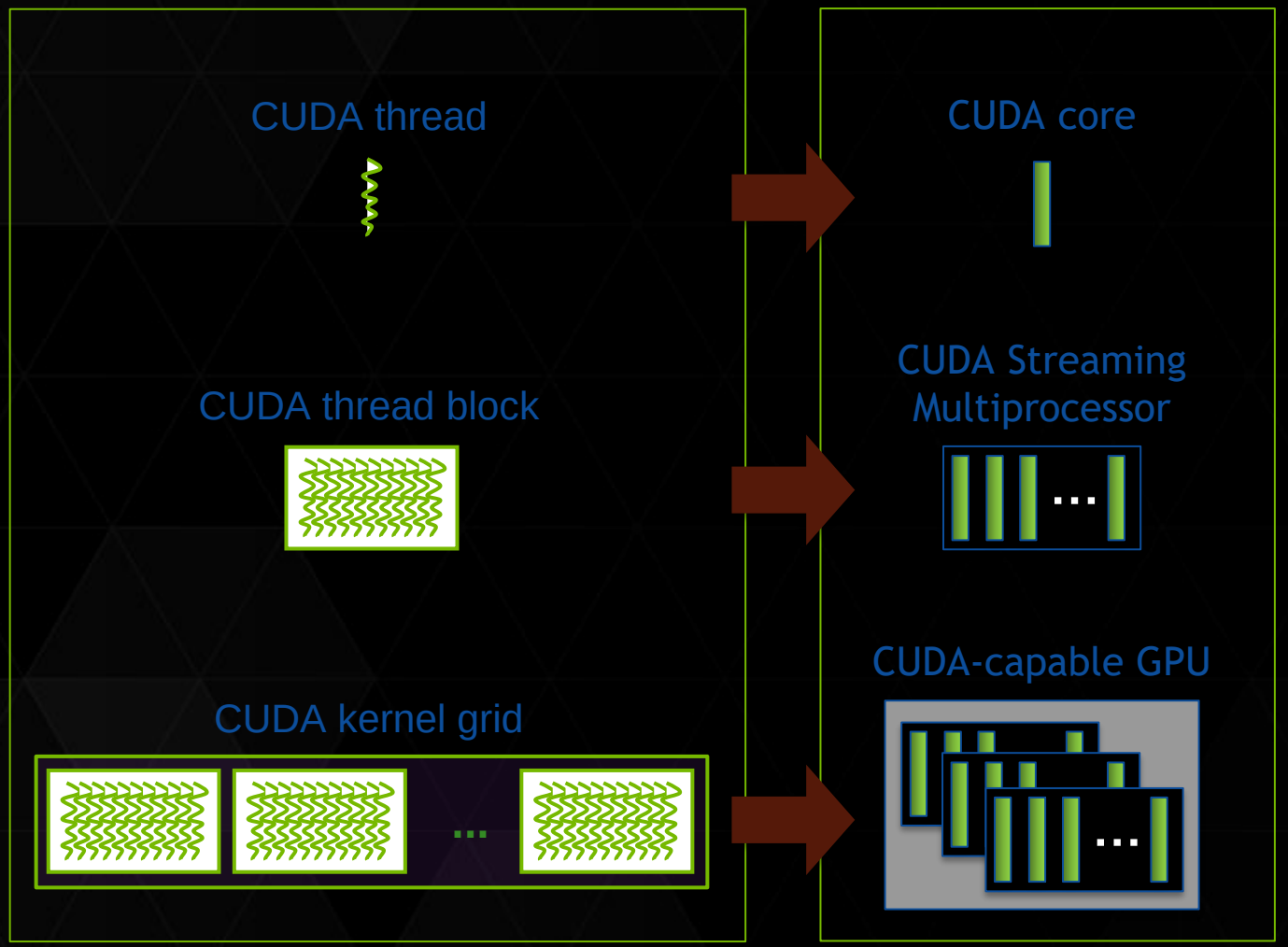
  globalThreadID = threadIdx.x + blockIdx.x * blockDim.x

- Similarly in Fortran:

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



# 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' requirements and the SM's resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

# GENERAL RULE FOR CHOICE OF NUM THREADS/NUM BLOCKS

- Today's examples – numThreads*numBlocks chosen to match dataset size
  - Grid-stride loops are an alternative approach which allows us to work with arbitrary size datasets

- numThreads = 32*k, k=1, or 2, … or 32
  - Instructions can be issued to 32 threads at once (a warp)
  - Device-specific, but max number of threads *per block* is usually 1024 (i.e. 32 x 32) link
- numBlocks = numSM*m, m=1, 2, …
  - numSM = number of streaming multiprocessors on the device (cudaGetDeviceProperties)
  - Multiples like this help with load-balancing
- On V100 <<<80, 64>>> would create same number of threads as there are cores
  - But ideally want more threads than this … e.g. <<<16*80, 256>>>
- There is an API to help: Occupancy API blog

# Vector addition

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

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

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

   integer, value :: n
   integer, device :: a(n), b(n), c(n)
   integer :: myid

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

     c(myid) = a(myid) + b(myid)

end subroutine add
```

OpenACC More Science, Less Programming

OPEN HACKATHONS

# MEMORY MANAGEMENT

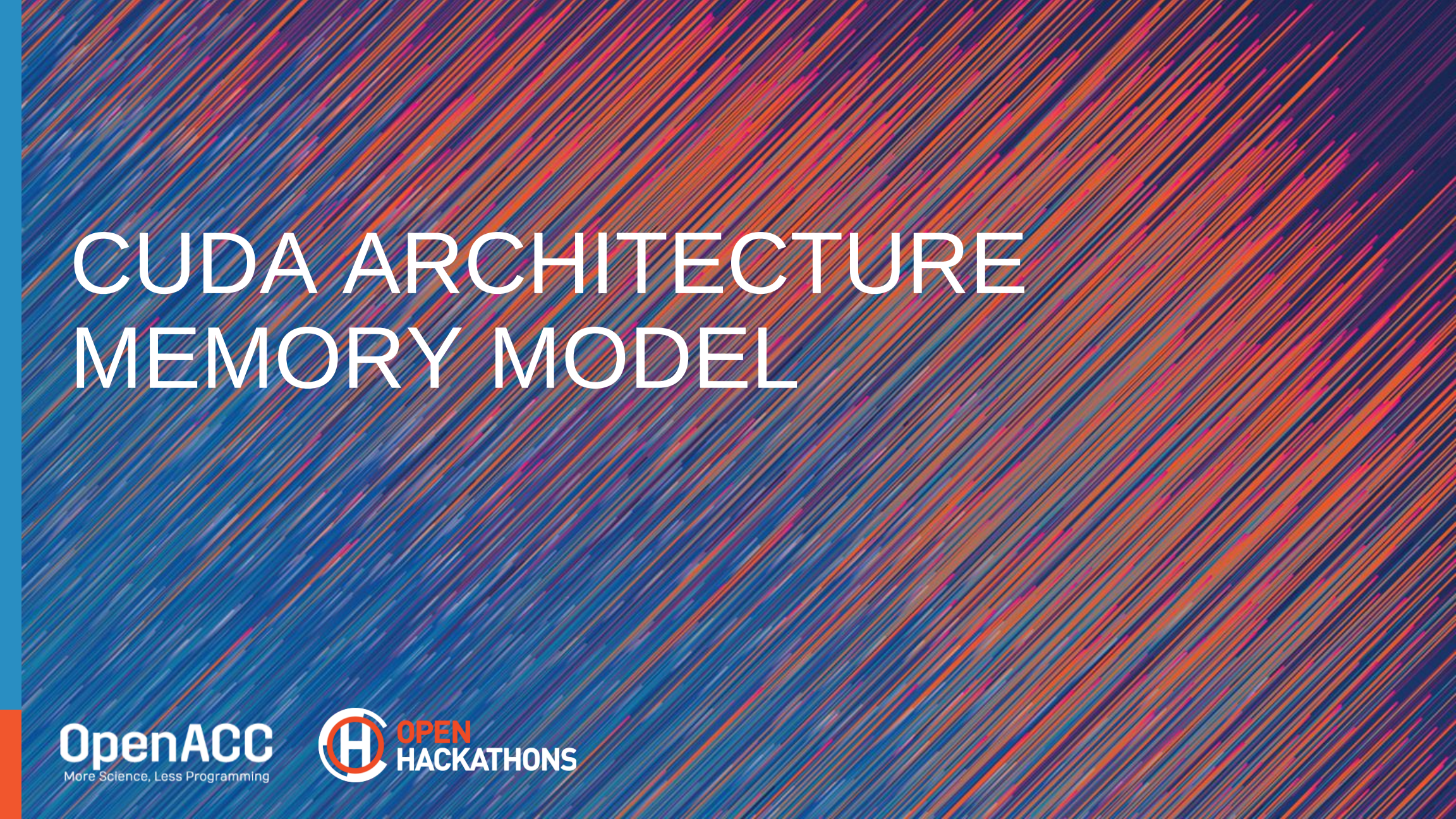
- Recall host and device memory are distinct entities
- There is a CUDA API for explicit device memory management
  - cudaMalloc(), cudaFree(), cudaMemcpy() ...
    - Similar to their C equivalents, malloc(), free(), memcpy()
    - Similar to their Fortran equivalents, allocate(), deallocate
- For today's session we are going to use managed memory (also known as unified memory)
  - This allows the developer to concentrate on parallelism and think about data movement as an optimisation
  - 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

# Vector addition : `main`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 256

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

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

    numBlocks = N/THREADS_PER_BLOCK;

    add<<< numBlocks,THREADS_PER_BLOCK>>>( a,
     b, c );

    cudaFree( a );
    cudaFree( b );
    cudaFree( c );
    return 0;
}
```

```
program main
    use cudafor

    integer, parameter:: numThreads = 256
    integer, parameter:: n = 2048*2048
    integer :: numBlocks
    real, managed, allocatable, dimension(:) :: a, b, c

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

    numBlocks = n / numThreads;

    call add<<< numBlocks, numThreads >>>(n, a, b, c)

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


end program main
```

# CUDA ARCHITECTURE MEMORY MODEL



# GPU ARCHITECTURE

## Two Main components

### Global memory

Analogous to RAM in a CPU server

Accessible by both GPU and CPU

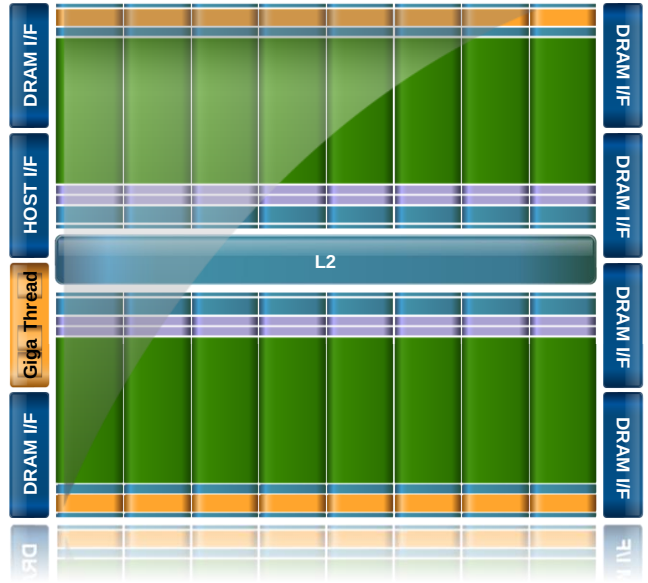
e.g. A100 80GB with bandwidth currently up to 2 TB/s

### Streaming Multiprocessors (SMs)

SMs perform the actual computations

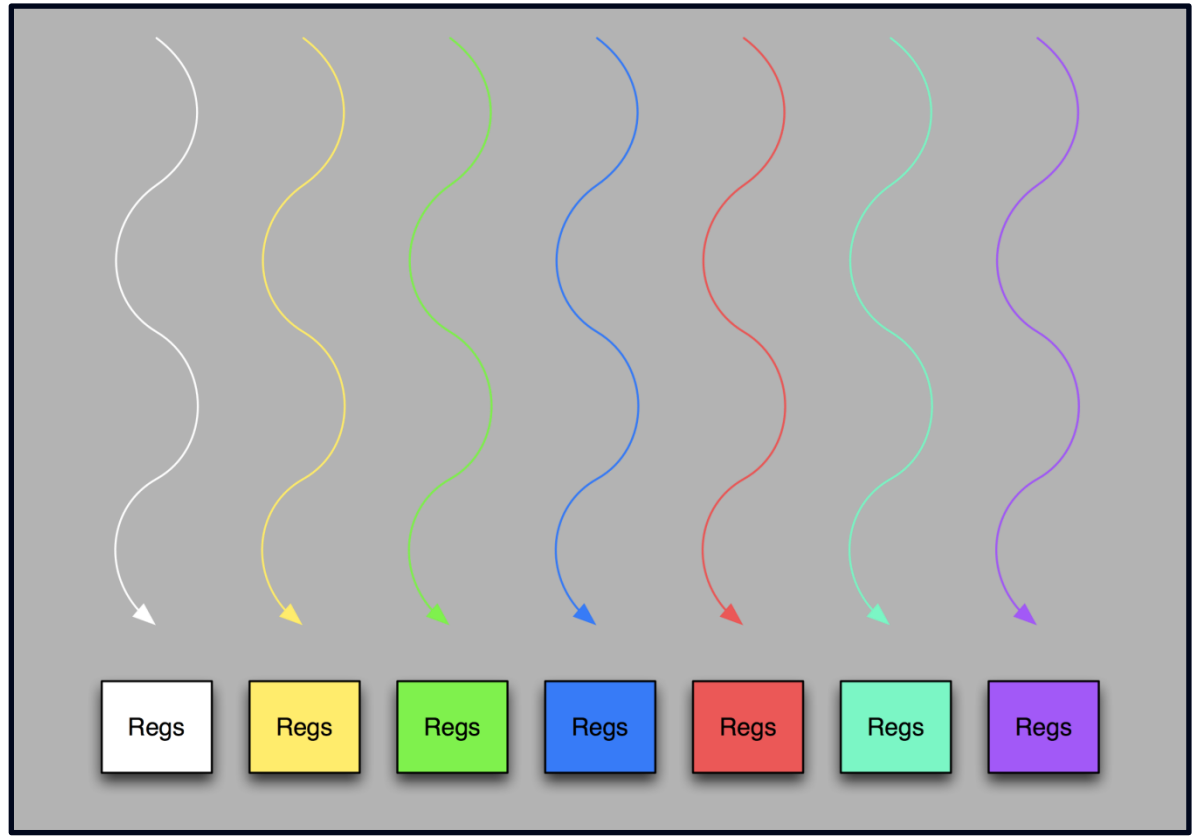
Each SM has its own:

Control units, registers, execution pipelines, caches



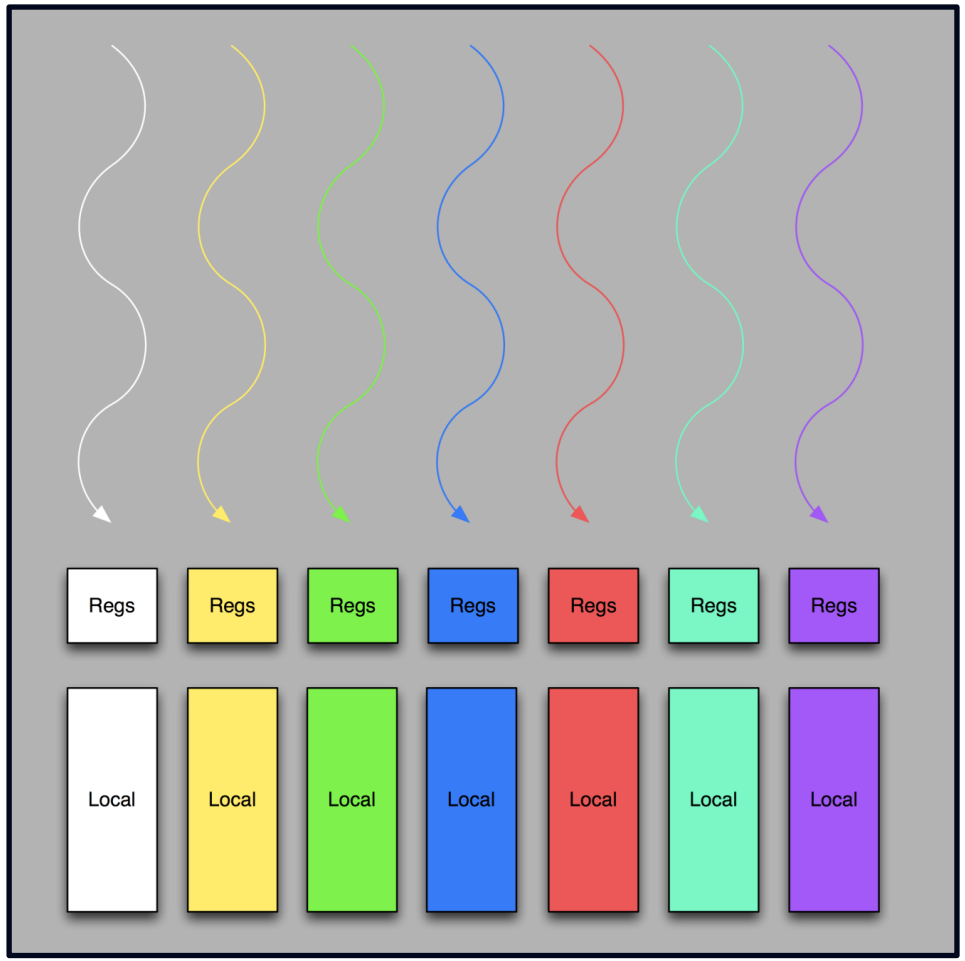
# MEMORY HIERARCHY

- Thread
  - Registers



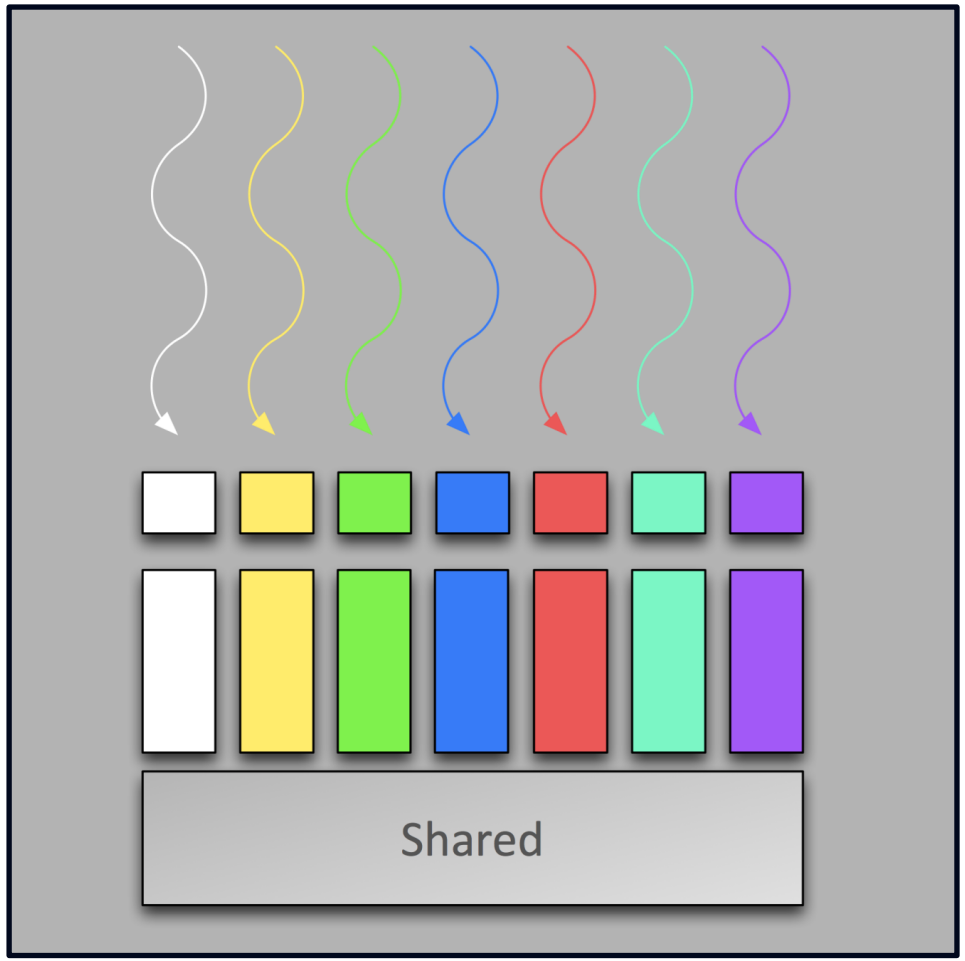
# MEMORY HIERARCHY

- Thread
  - Registers
- Thread
  - Local memory



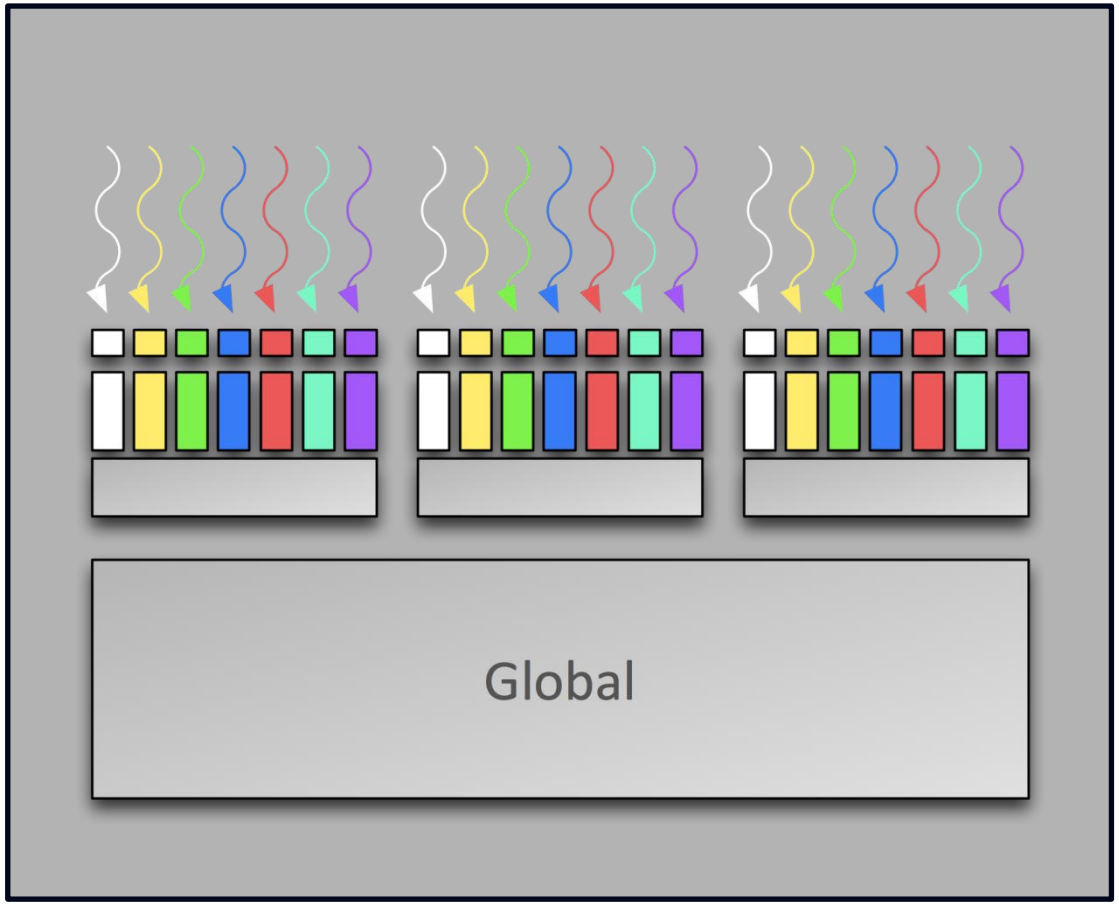
# MEMORY HIERARCHY

- 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



# Acknowledgment

Learn more at

# WWW.OPENHACKATHONS.ORG

OpenACC
More Science, Less Programming

OPEN HACKATHONS