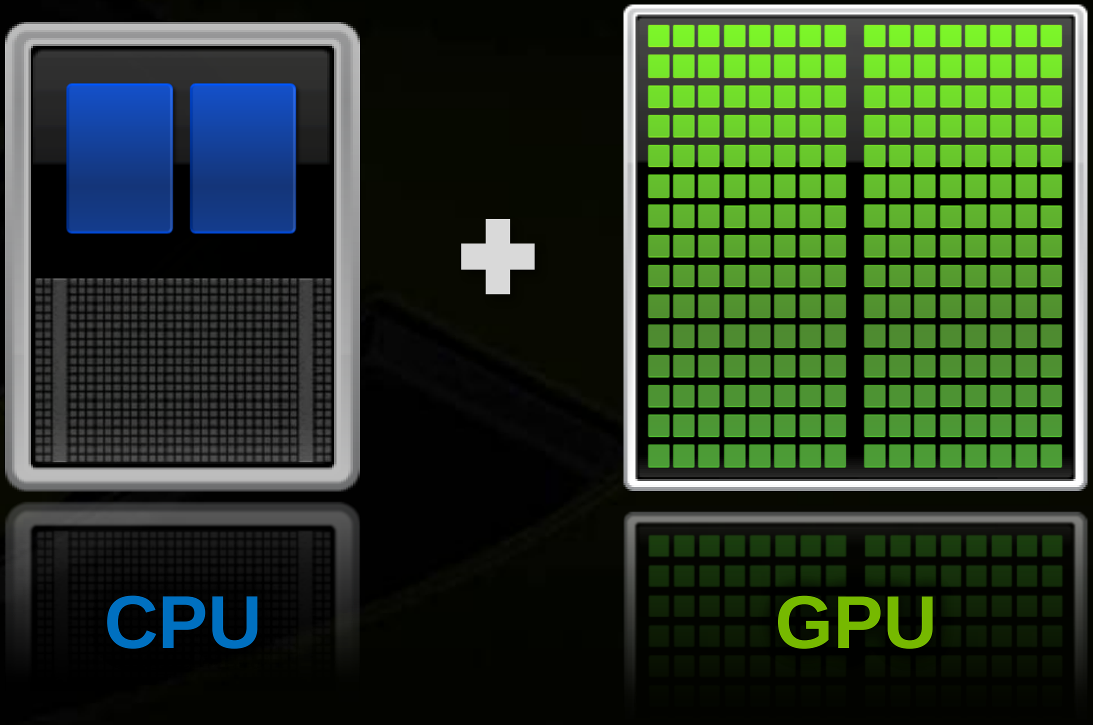
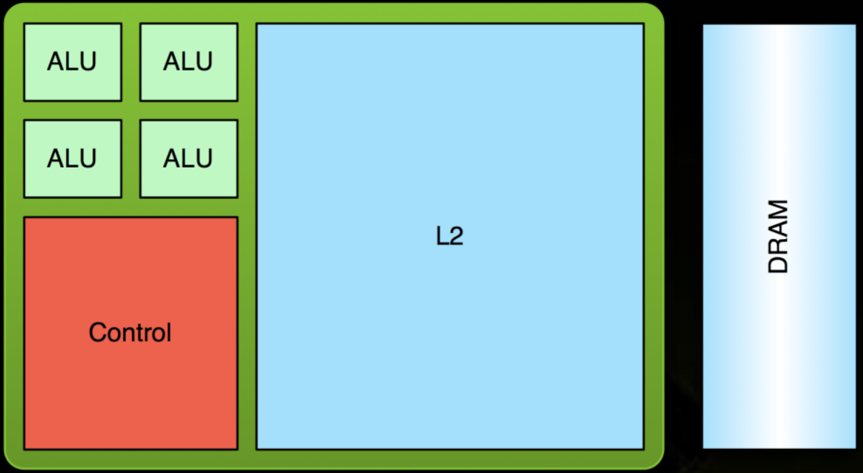
GPGPU Revolutionizes Computing

Latency Processor + Throughput processor

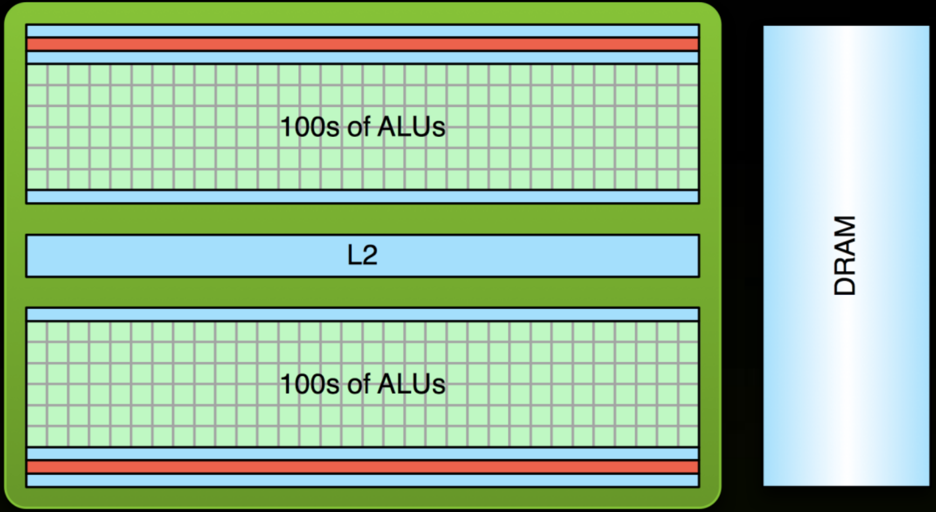


Low Latency or High Throughput?



CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

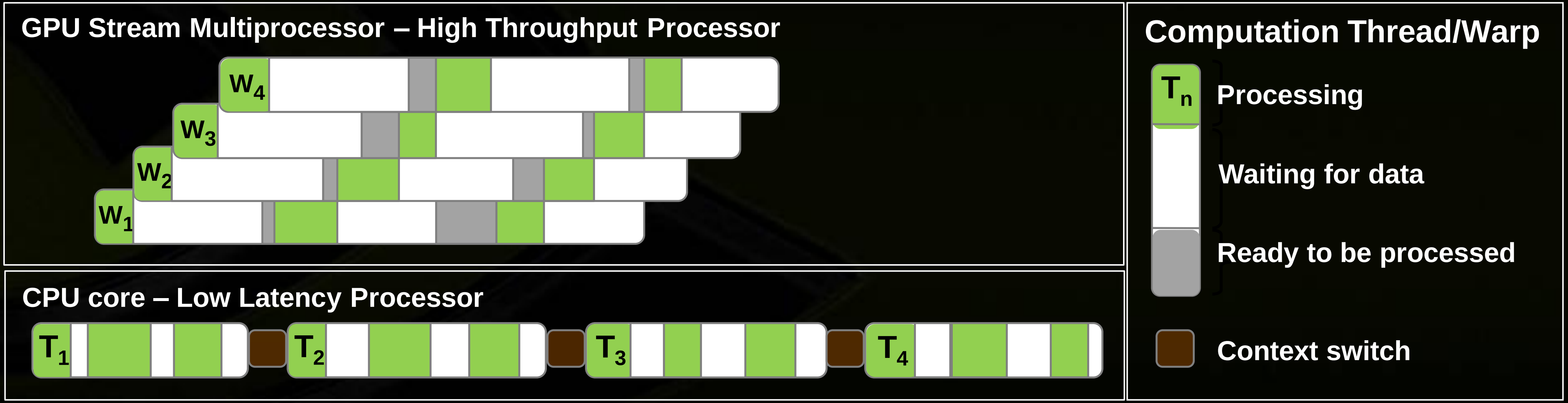


GPU

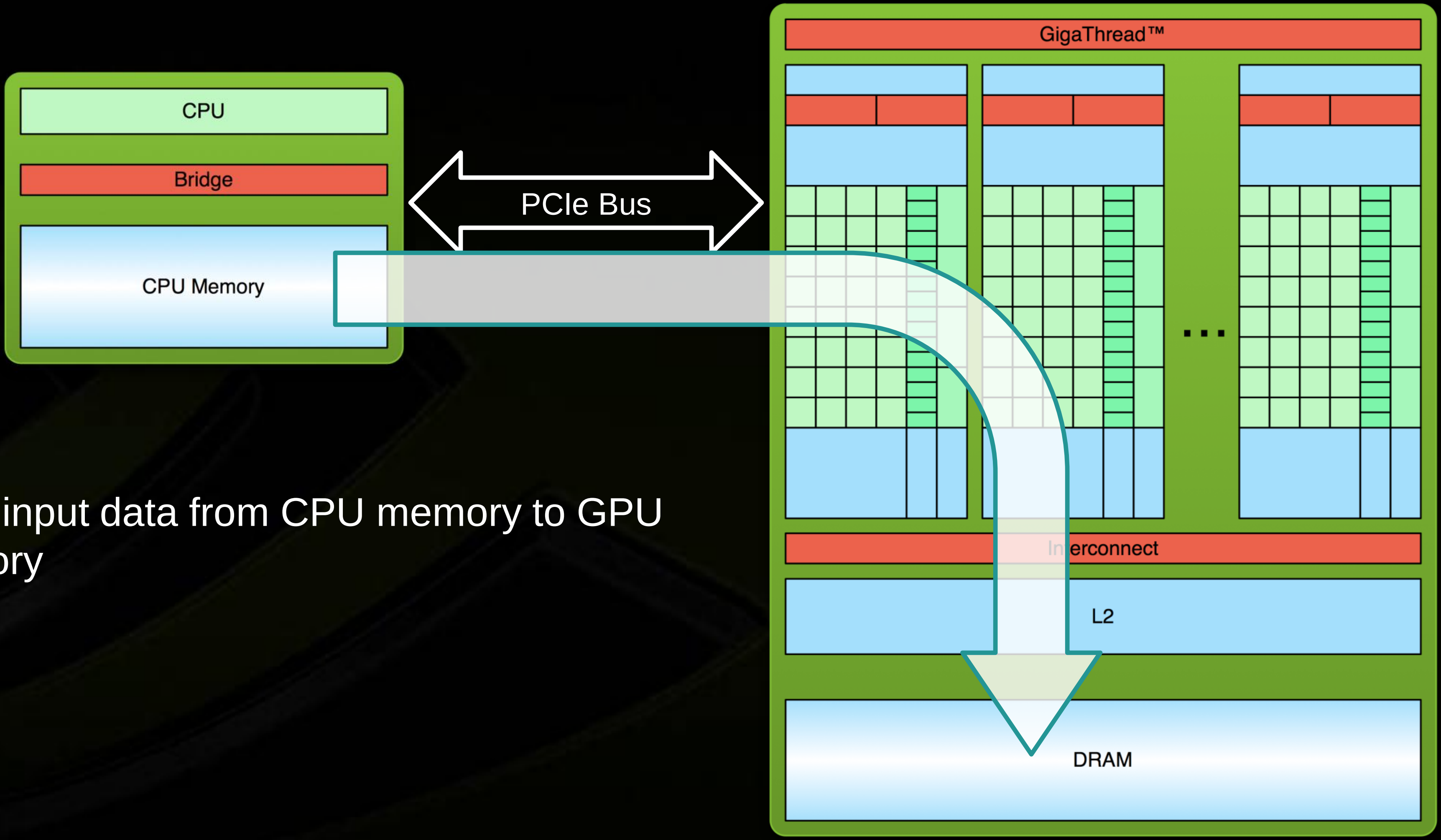
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps

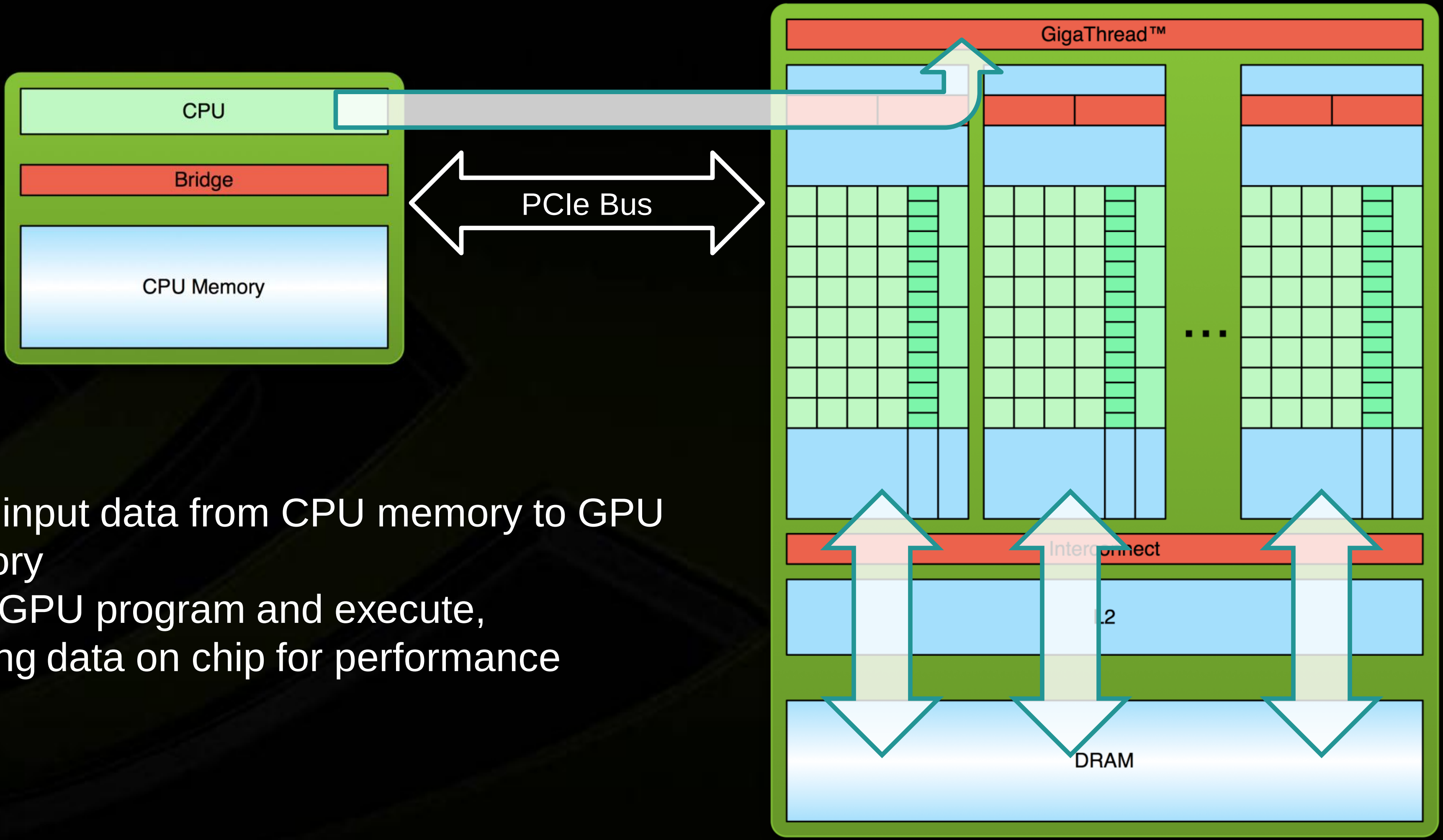


Processing Flow



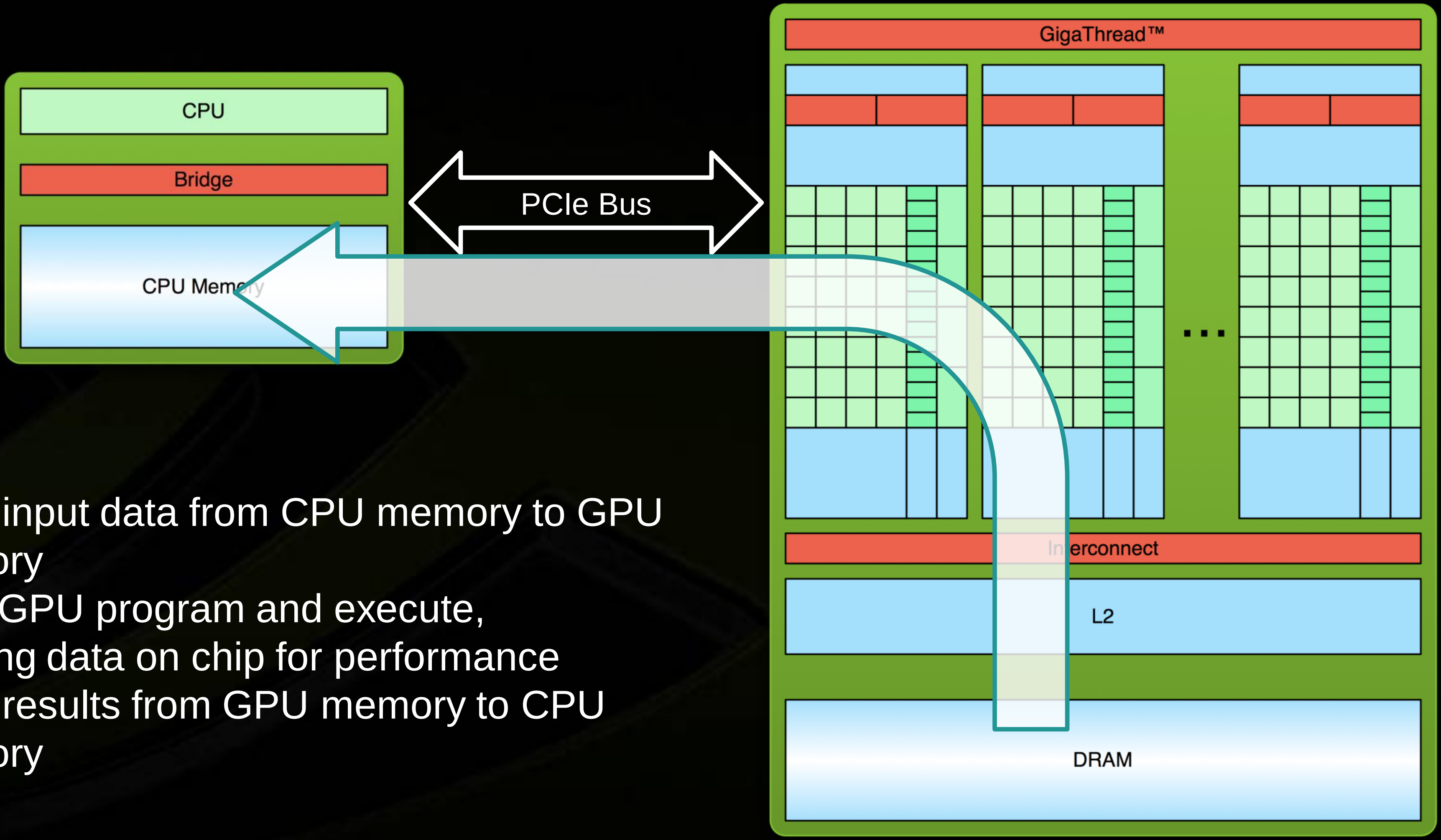
1. Copy input data from CPU memory to GPU memory

Processing Flow



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

Processing Flow



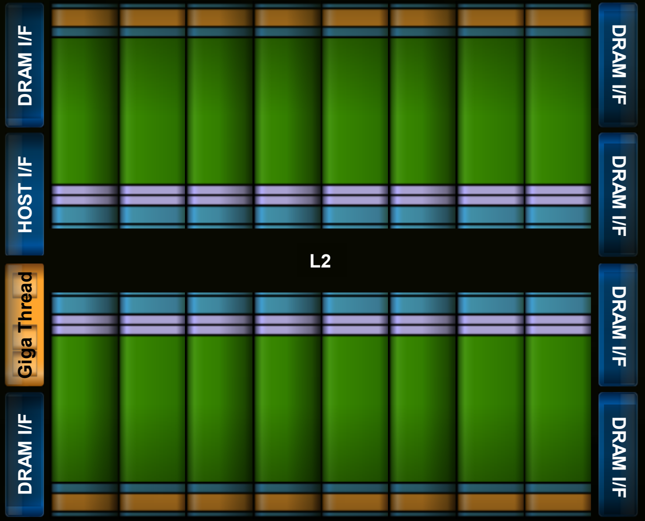
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

GPU ARCHITECTURE

GPU Architecture: Two Main Components

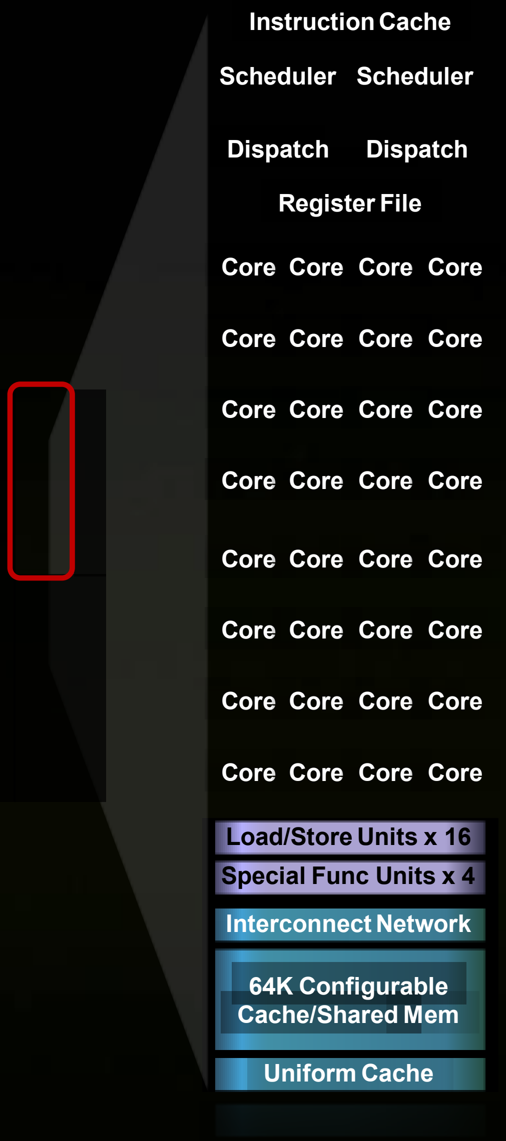
- **Global memory**
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 32 GB
  - Bandwidth currently up to 1.13 TB/s for Volta V100

  - ECC on/off option for Quadro and Tesla products
- **Streaming Multiprocessors (SMs)**
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



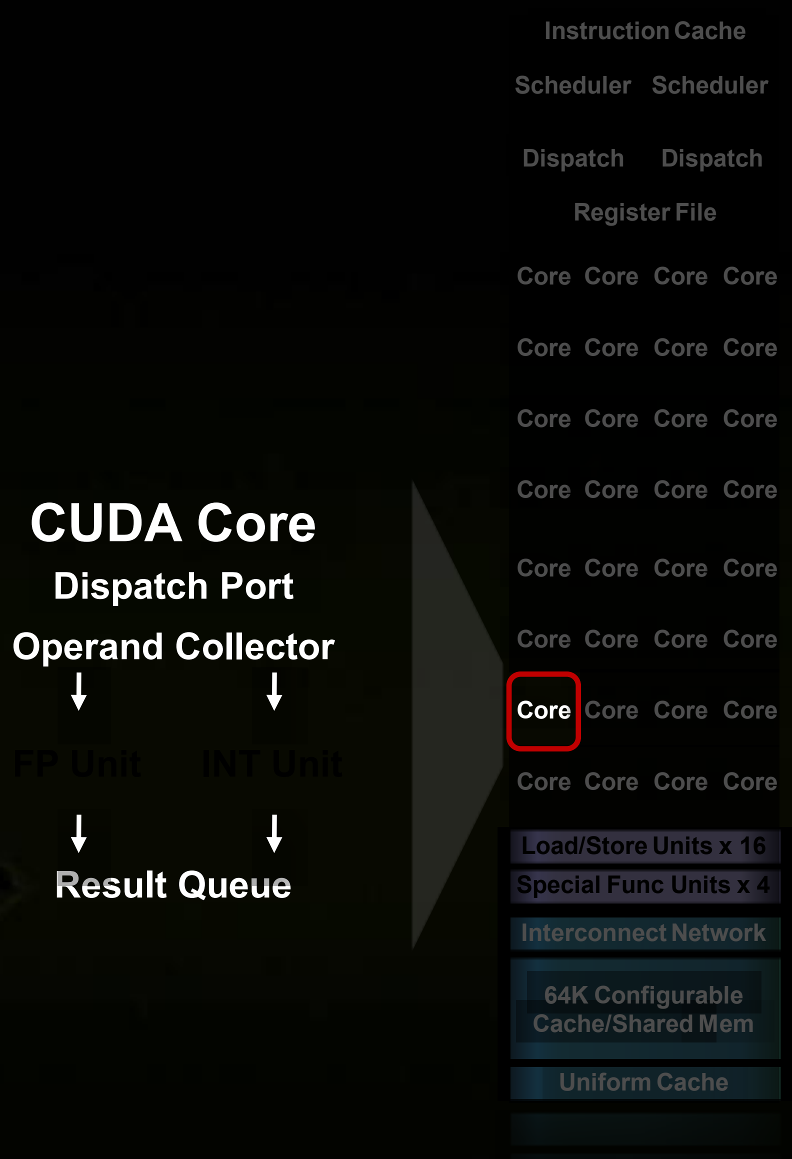
GPU Architecture – Volta Streaming Multiprocessor (SM)

- 64 CUDA cores per SM
  - 80 SMs
  - 16.4 Tops, fp32
  - 8.2 Tops, fp64
- 4 warp schedulers
  - Up to 2048 threads concurrently
- 4 special-function units
- Up to 96KB shared mem
- 64KB 32-bit registers



GPU Architecture – Volta: CUDA Core

- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit



GPU Architecture – Volta: Memory System

- **L1**
  - 128KB per SM   shared with texture
  - Hardware-managed
  - Aggregate bandwidth: 13TB/s
- **Shared memory**
  - User-managed scratch-pad
    - Hardware will not evict until threads overwrite
  - 0-96KB per SM   (managed by program)
  - Aggregate bandwidth per GPU: 10TB/s

GPU Architecture – Volta: Memory System

- **ECC protection:**
  - **DRAM**
    - ECC supported for GDDR5 memory
  - **All major internal memories are ECC protected**
    - Register file, L1 cache, L2 cache

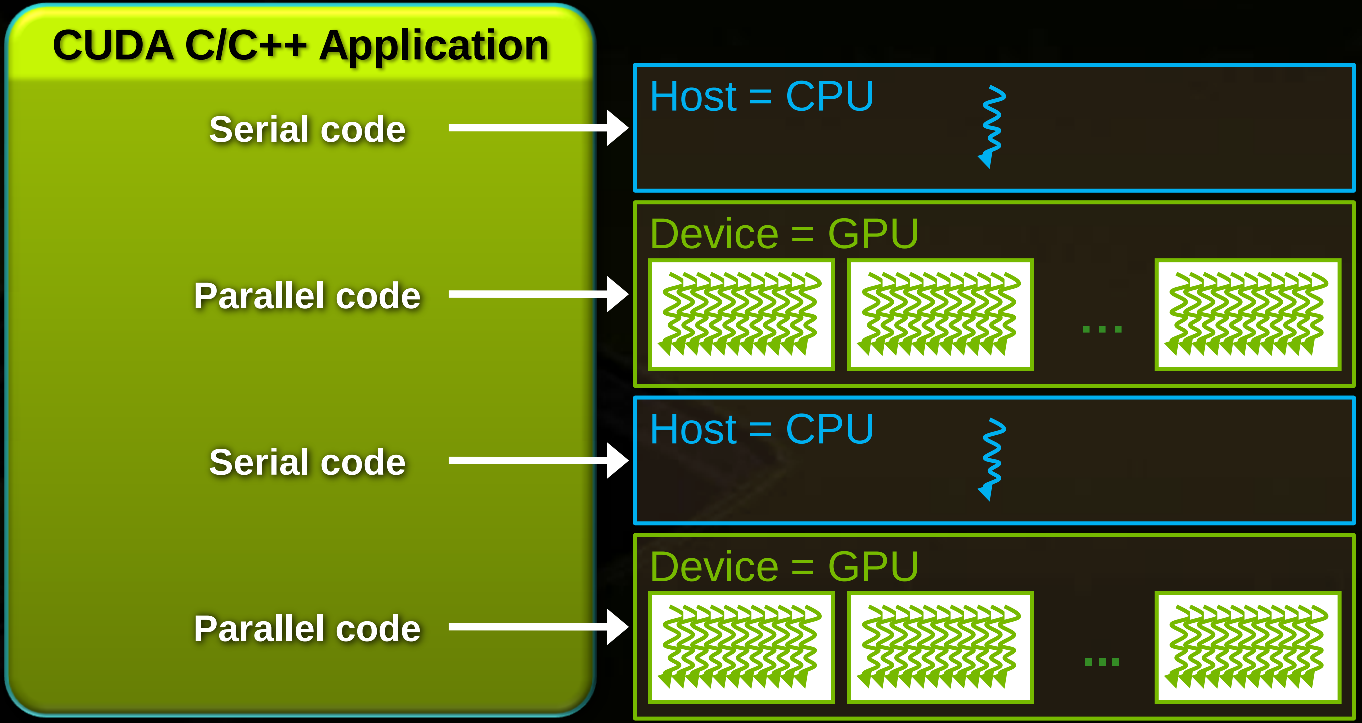
Overview of Volta V100-DGXS

C2050 Specifications	
Processor clock	1.53 GHz
# of CUDA cores	5120
Peak floating-point perf	16.4 Tflops (SP)
Memory clock	877 MHz
Memory bus width	4096 bits
Memory size	32 GB

CUDA PROGRAMMING MODEL

Anatomy of a CUDA C/C++ Application

- Serial code executes in a Host (CPU) thread
- Parallel code executes in many Device (GPU) threads across multiple processing elements

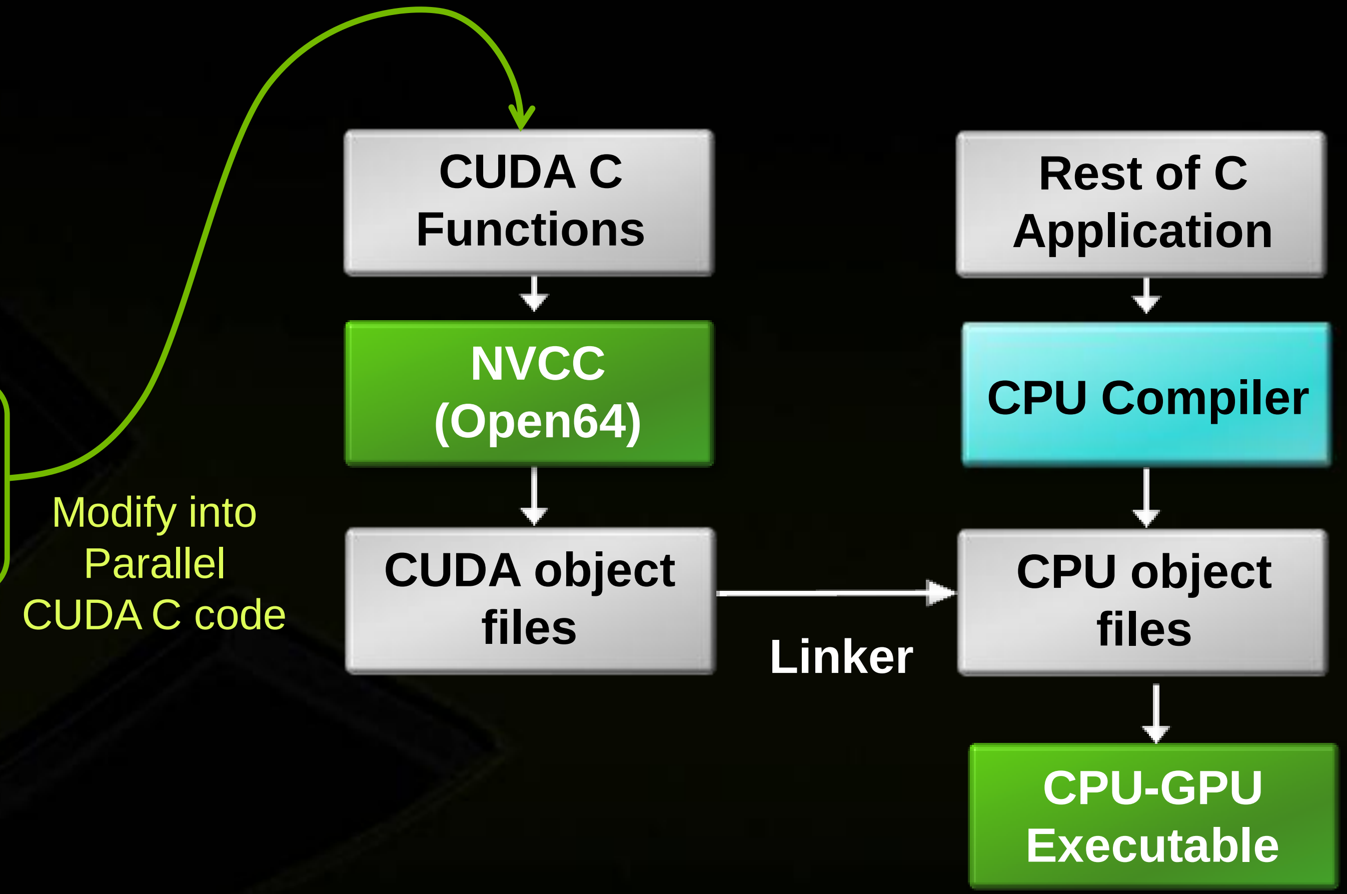


Compiling CUDA C Applications

```
void serial_function(… ) {
   ...
}
void other_function(int ... ) {
   ...
}

void saxpy_serial(float ... ) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

void main( ) {
   float x;
   saxpy_serial(..);
   ...
}
```



CUDA C : C with a few keywords

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

CUDA C : C with a few keywords

- Kernel: function called by the host that executes on the GPU
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
- Functions must be declared with a qualifier:
  - __global__ : GPU kernel function launched by CPU, must return void
  - __device__ : can be called from GPU functions
  - __host__ : can be called from CPU functions (default)
  - __host__ and __device__ qualifiers can be combined

CUDA Kernels

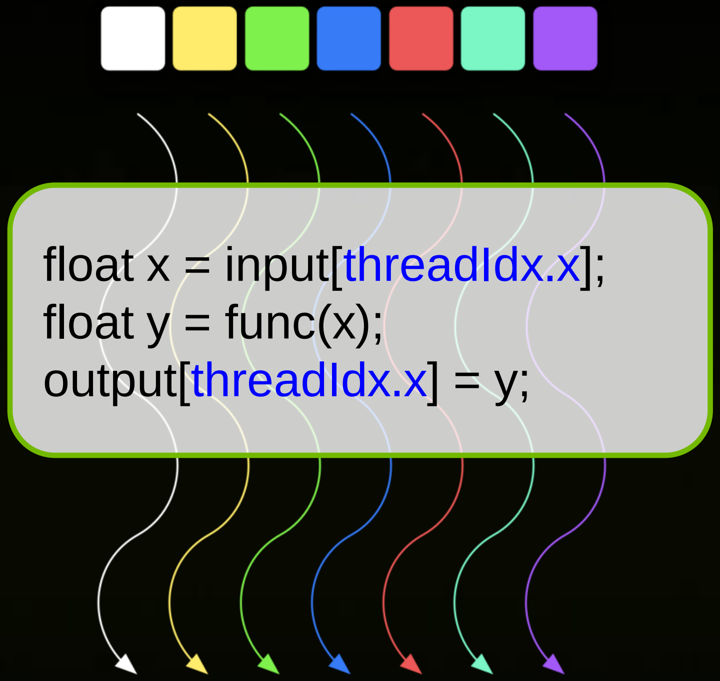
- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads

- CUDA threads:
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously

CPU	Host	Executes functions
GPU	Device	Executes kernels

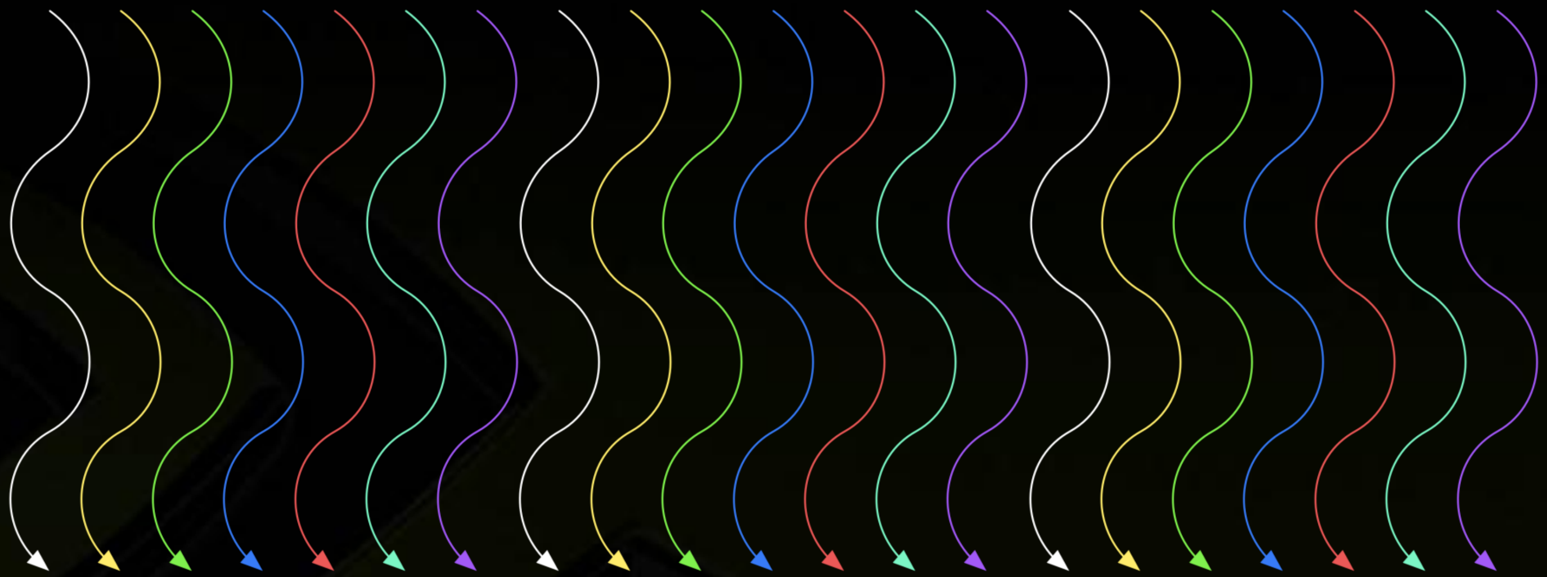
CUDA Kernels: Parallel Threads

- A kernel is a function executed on the GPU as an 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

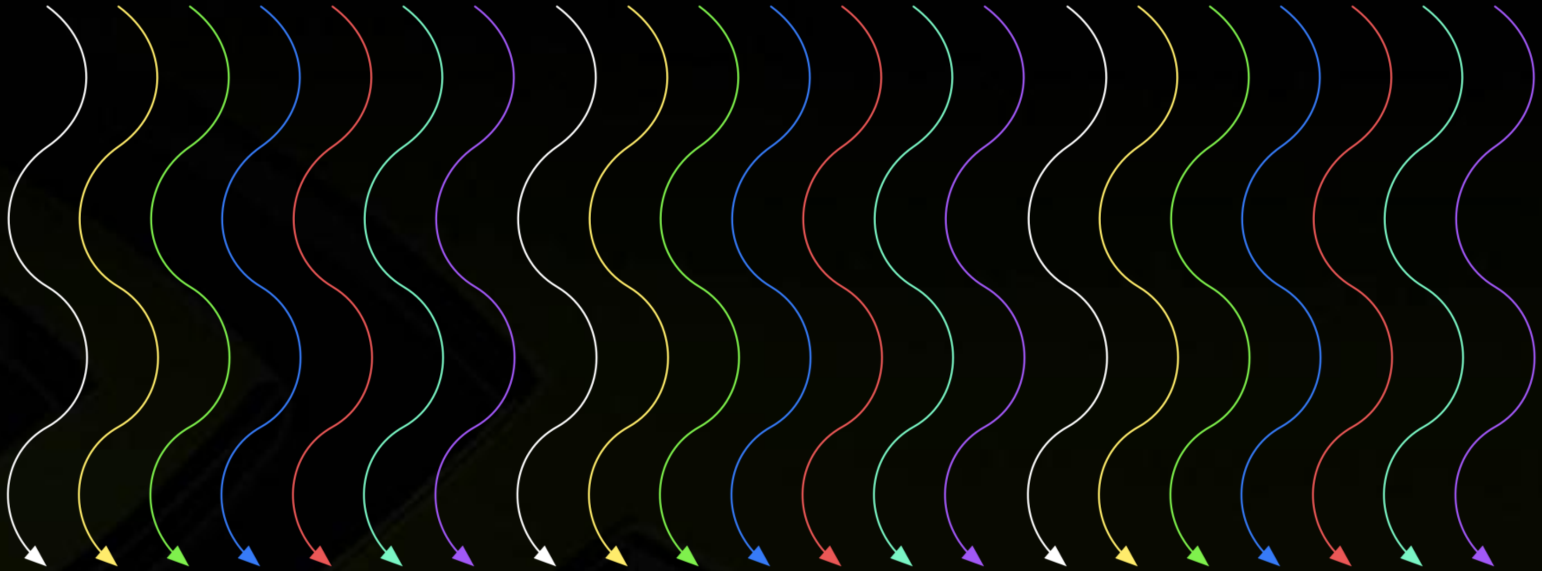


```
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```

CUDA Kernels: Subdivide into Blocks

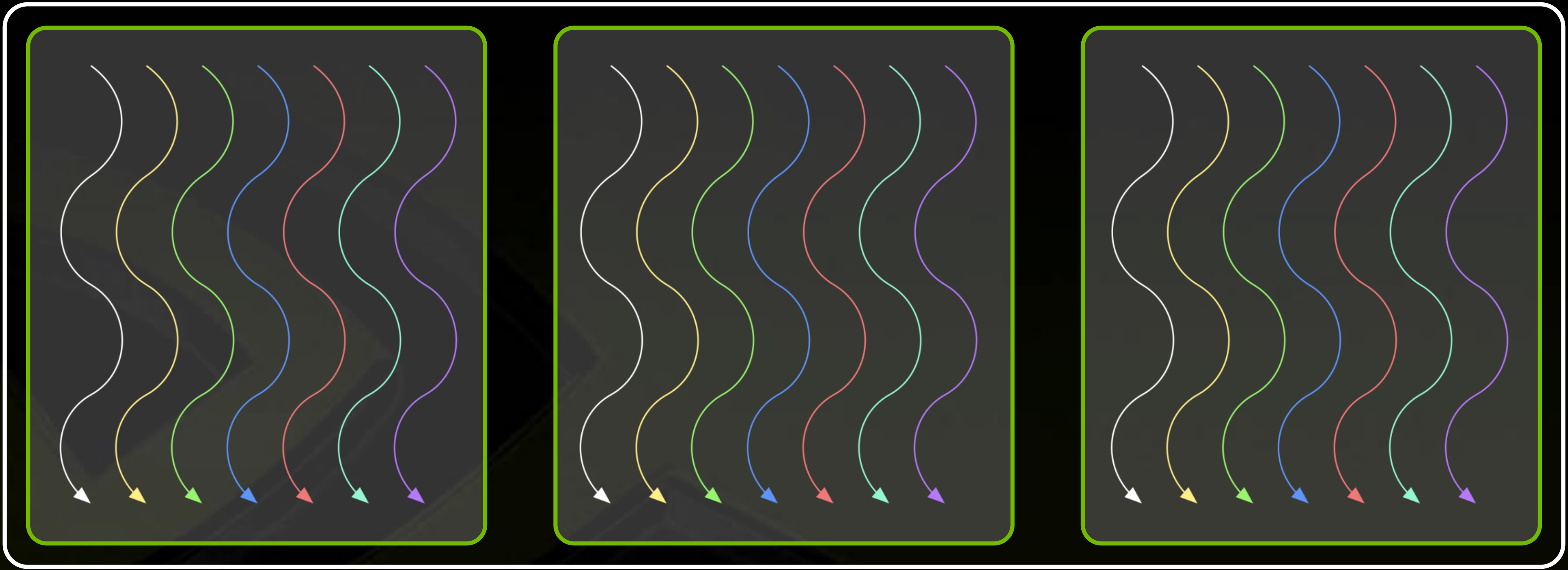


CUDA Kernels: Subdivide into Blocks



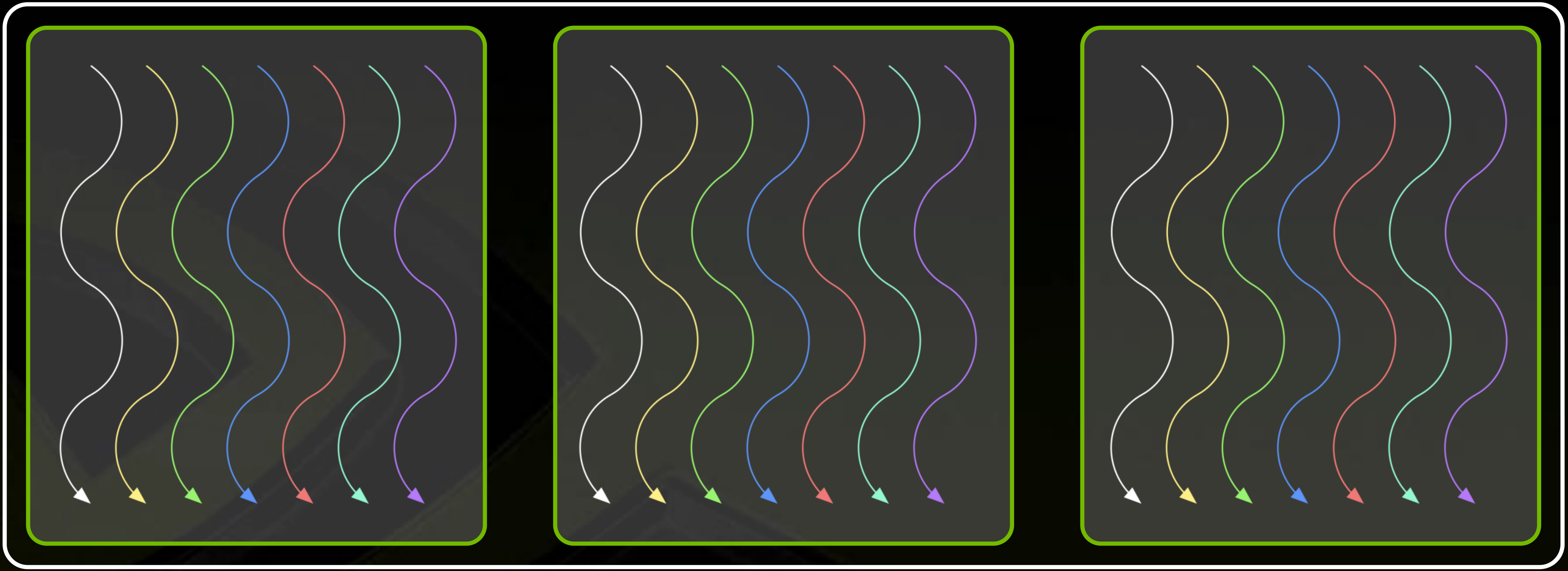
- Threads are grouped into **blocks**

CUDA Kernels: Subdivide into Blocks



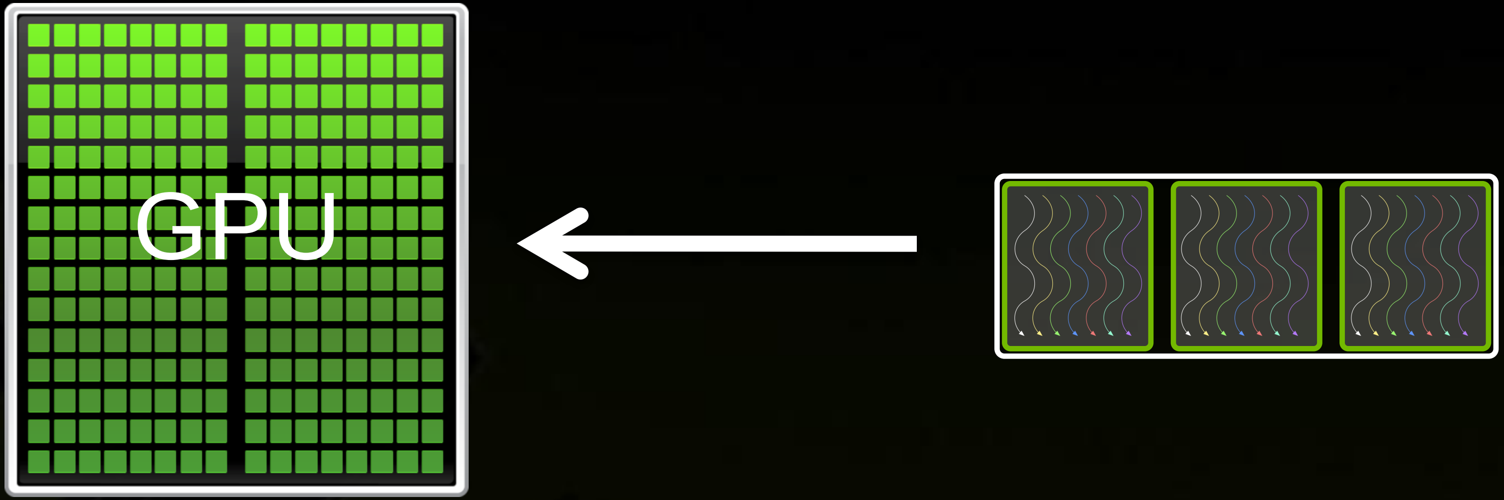
- Threads are grouped into blocks
- Blocks are grouped into a grid

CUDA Kernels: Subdivide into Blocks



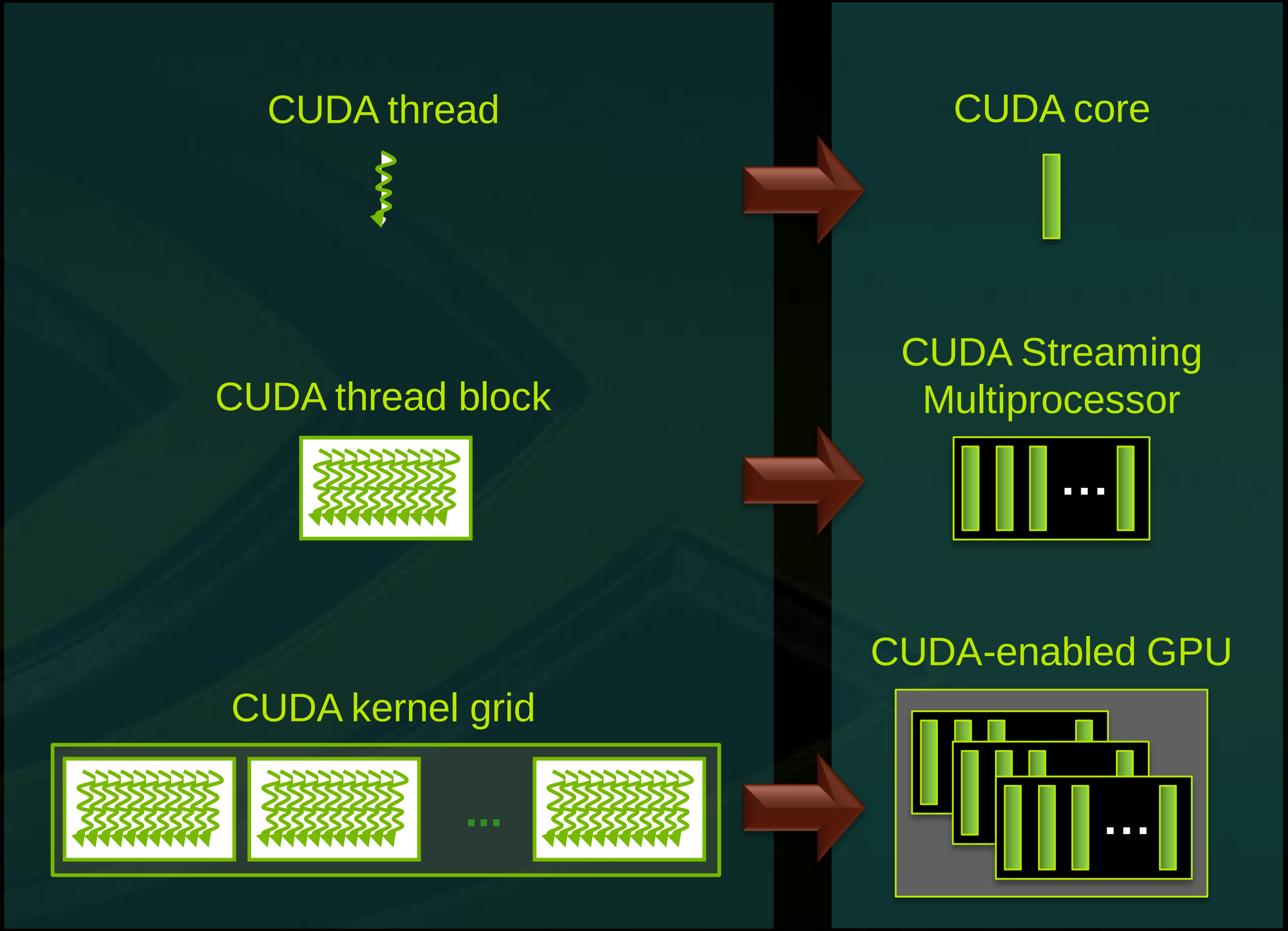
- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

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



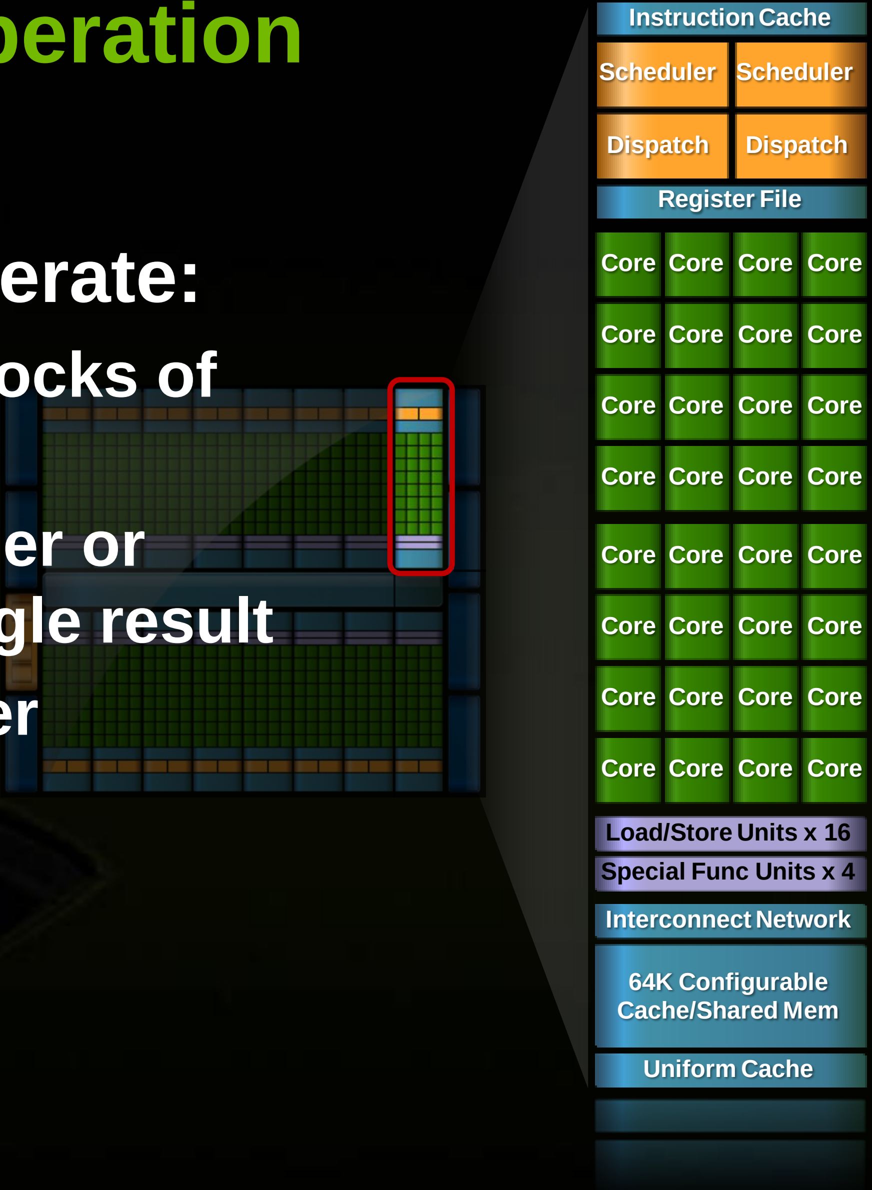
- Each thread is executed by a core

- 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

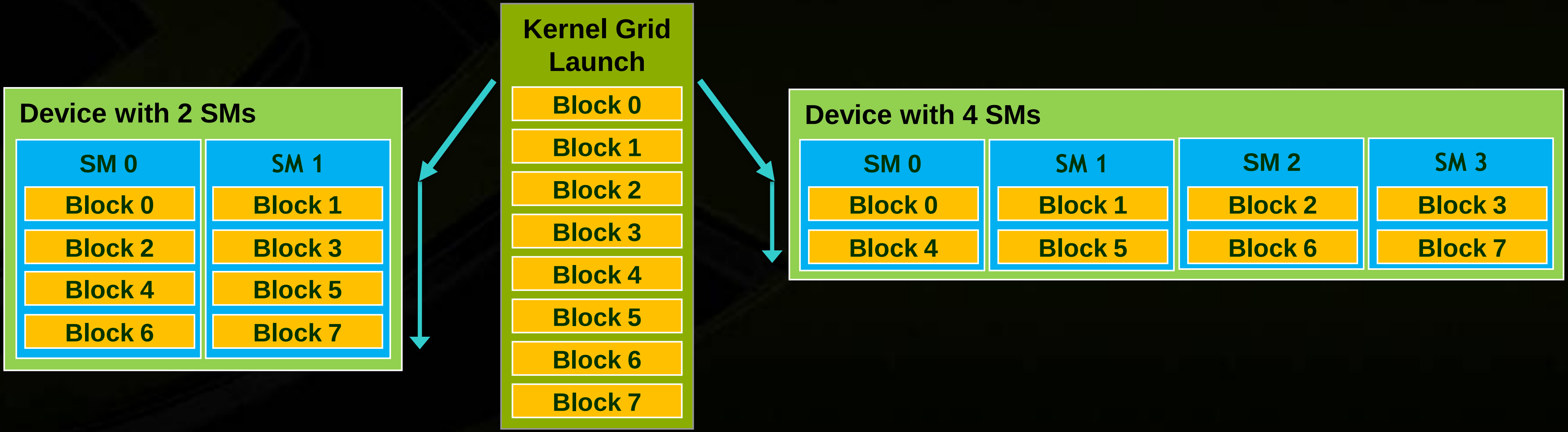
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory all will use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other



Thread blocks allow scalability

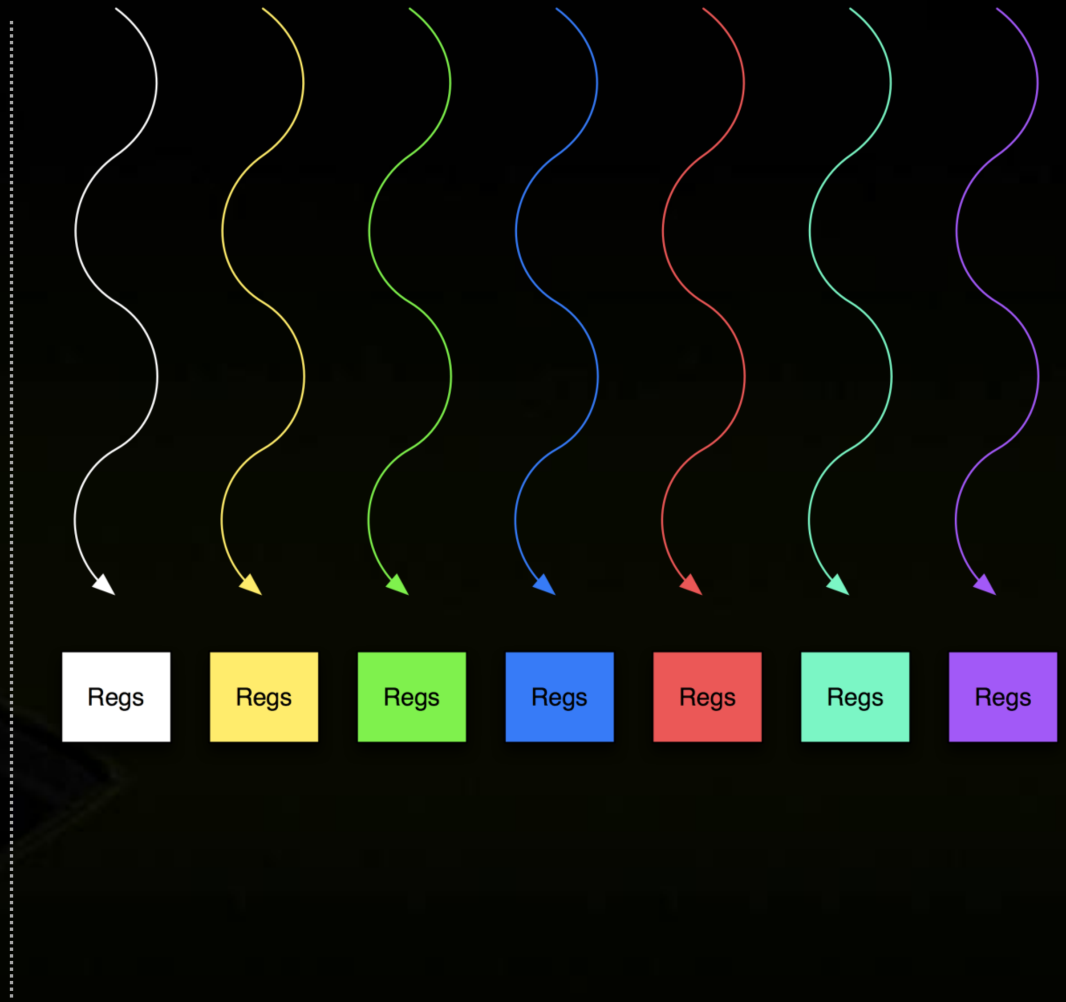
- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs



CUDA MEMORY SYSTEM

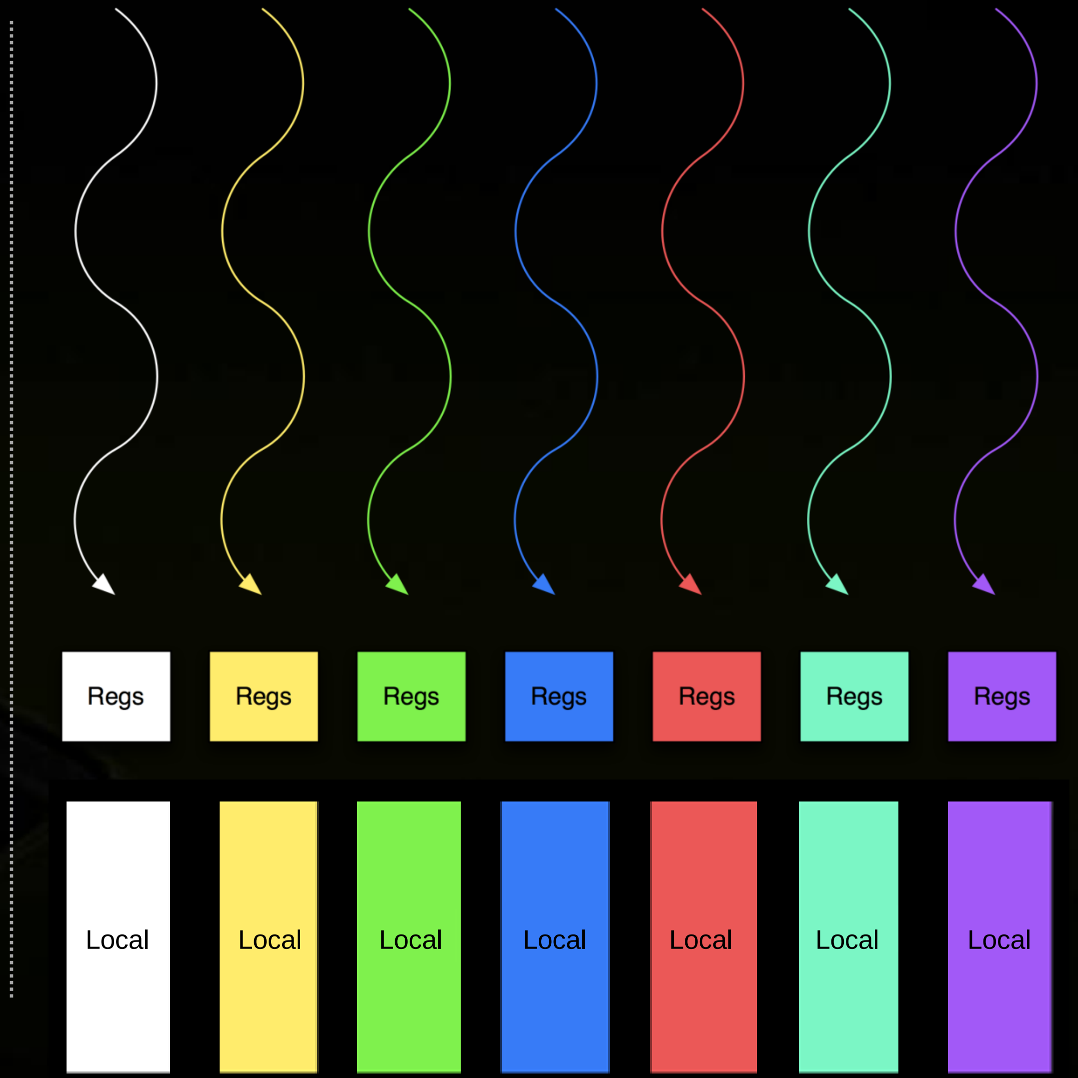
Memory hierarchy

- **Thread:**
    - **Registers**



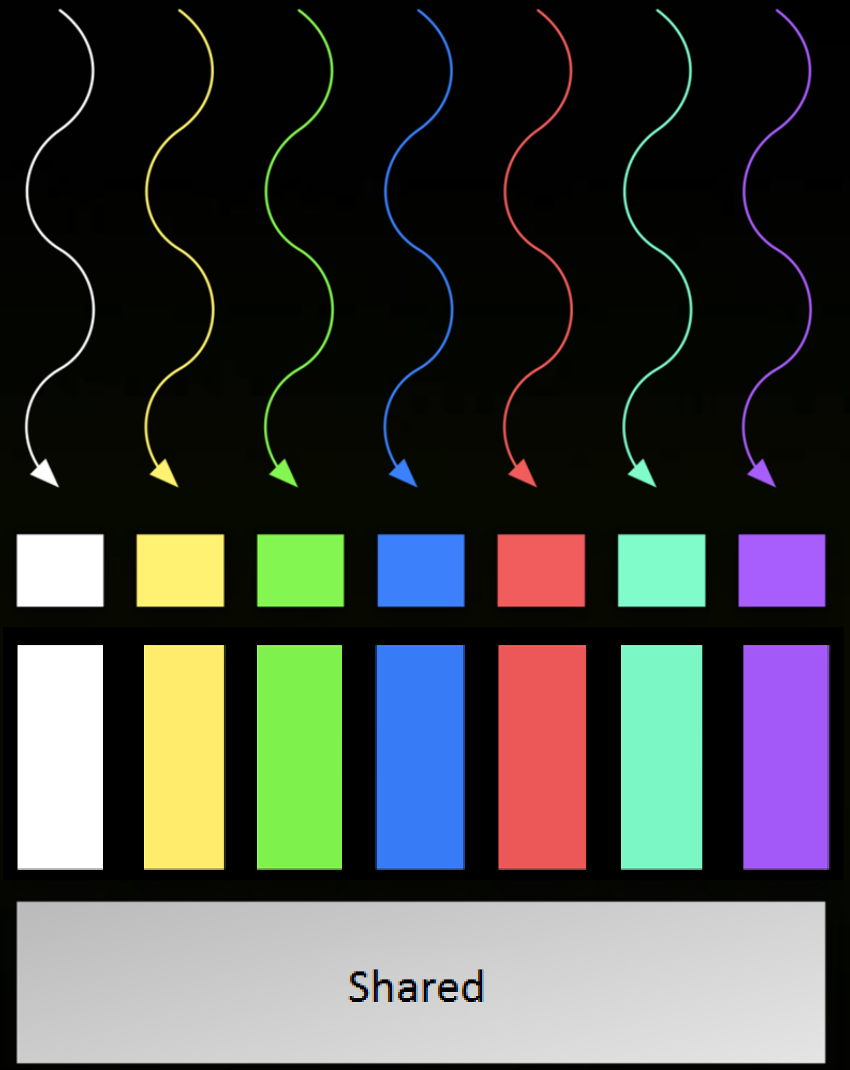
Memory hierarchy

- **Thread:**
  - **Registers**
  - **Local memory**



Memory hierarchy

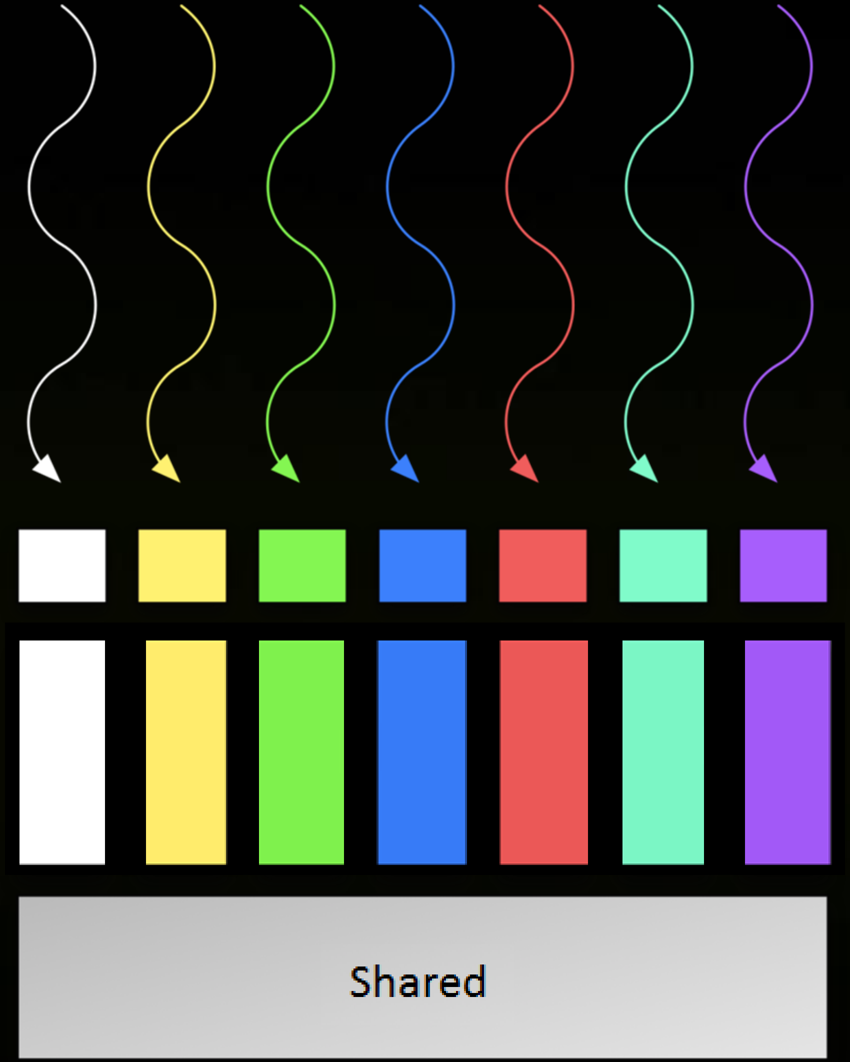
- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory



Memory hierarchy : Shared memory

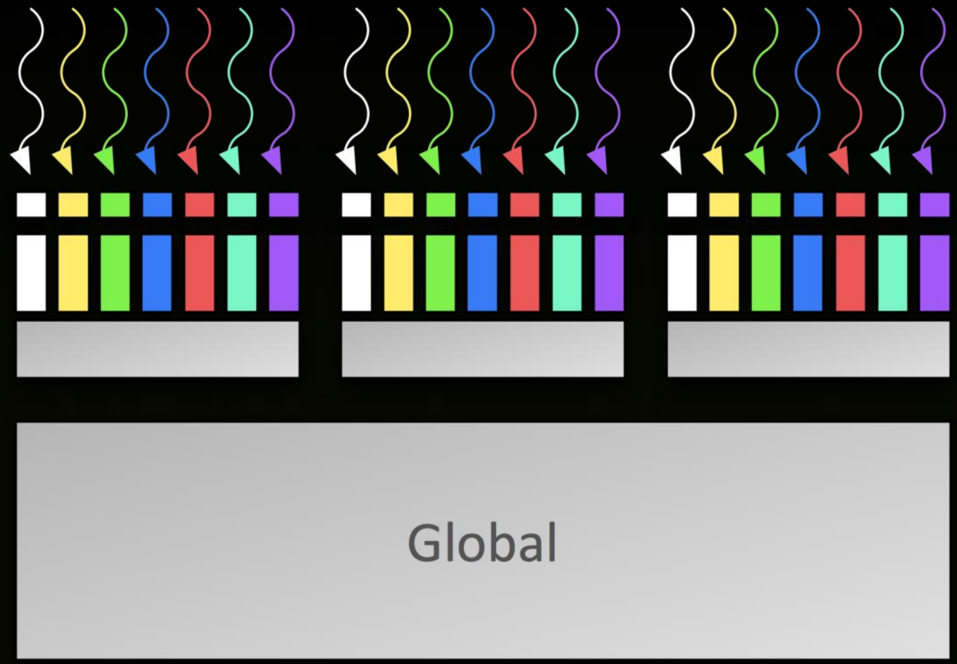
`__shared__ int a[SIZE];`

- Allocated per thread block, same lifetime as the block
- Accessible by any thread in the block
- Latency: a few cycles
- High aggregate bandwidth:
  - 14 * 32 * 4 B * 1.15 GHz / 2 = 1.03 TB/s
- Several uses:
  - Sharing data among threads in a block
  - User-managed cache (reducing gmem accesses)



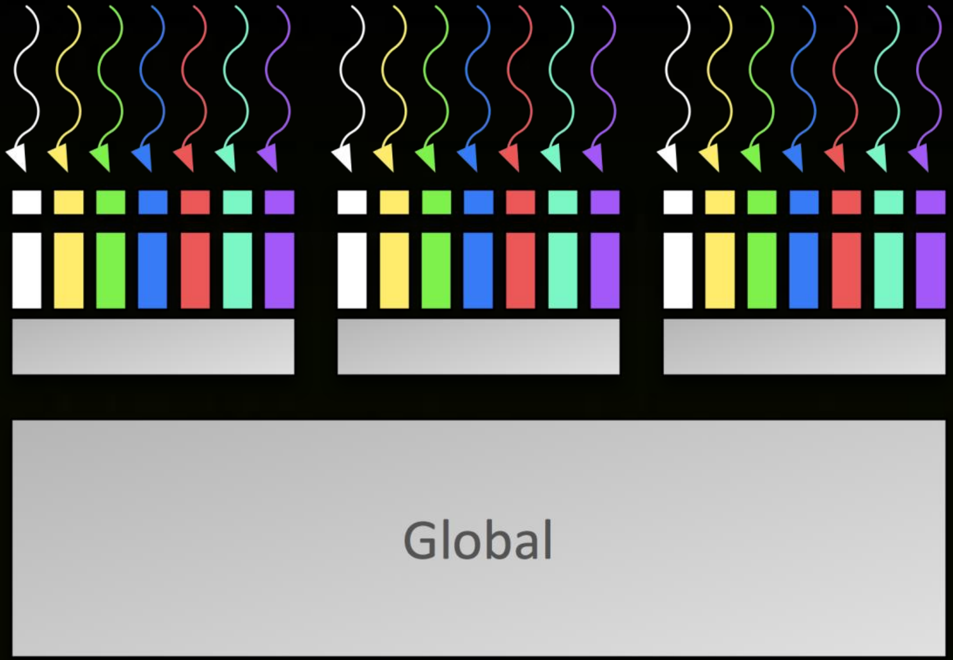
Memory hierarchy

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



Memory hierarchy : Global memory

- Accessible by all threads of any kernel
- Data lifetime: from allocation to deallocation by host code
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)
- Latency: 400-800 cycles
- Bandwidth: 156 GB/s
  - Note: requirement on access pattern to reach peak performance



CUDA DEVELOPMENT RESOURCES

CUDA Programming Resources

- CUDA Toolkit
  - Compiler, libraries, and documentation
  - Free download for Windows, Linux, and MacOS
- GPU Computing SDK
  - Code samples
  - Whitepapers
- Instructional materials on NVIDIA Developer site
  - CUDA introduction & optimization webinar: slides and audio
  - Parallel programming course at University of Illinois UC
  - Tutorials
  - Forums

GPU Tools

- Profiler
  - Available for all supported OSs
  - Command-line or GUI
  - Sampling signals on GPU for:
    - Memory access parameters
    - Execution (serialization, divergence)
- Debugger
  - Linux: cuda-gdb
  - Windows: Parallel Nsight
  - Runs on the GPU

[redacted]

Questions?