



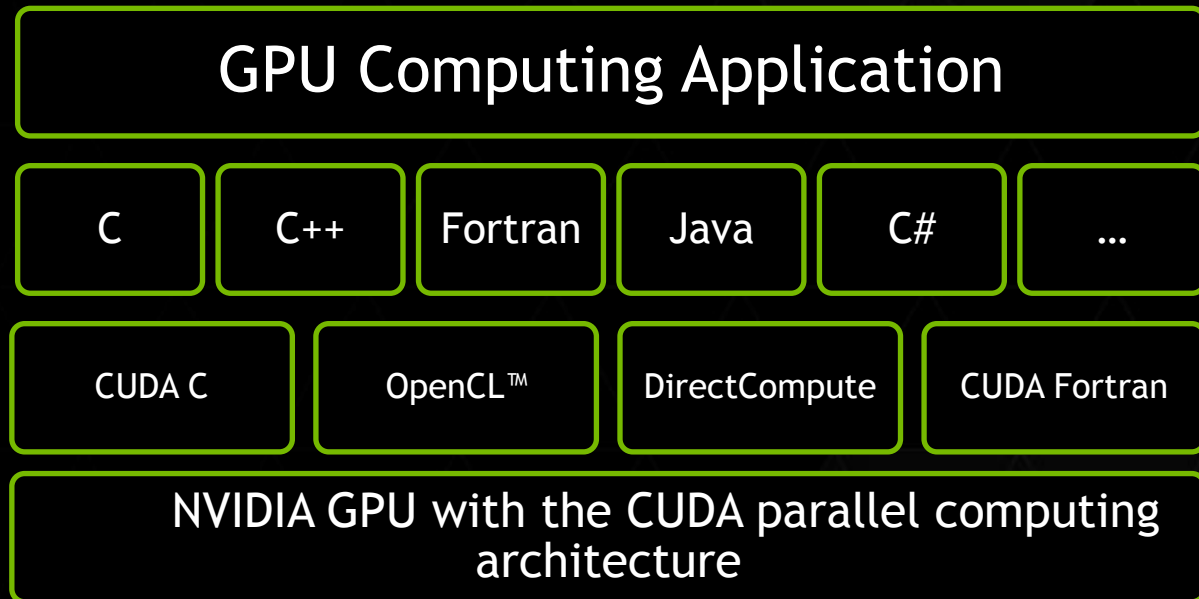
CUDA PROGRAMMING MODEL

Carlo Nardone

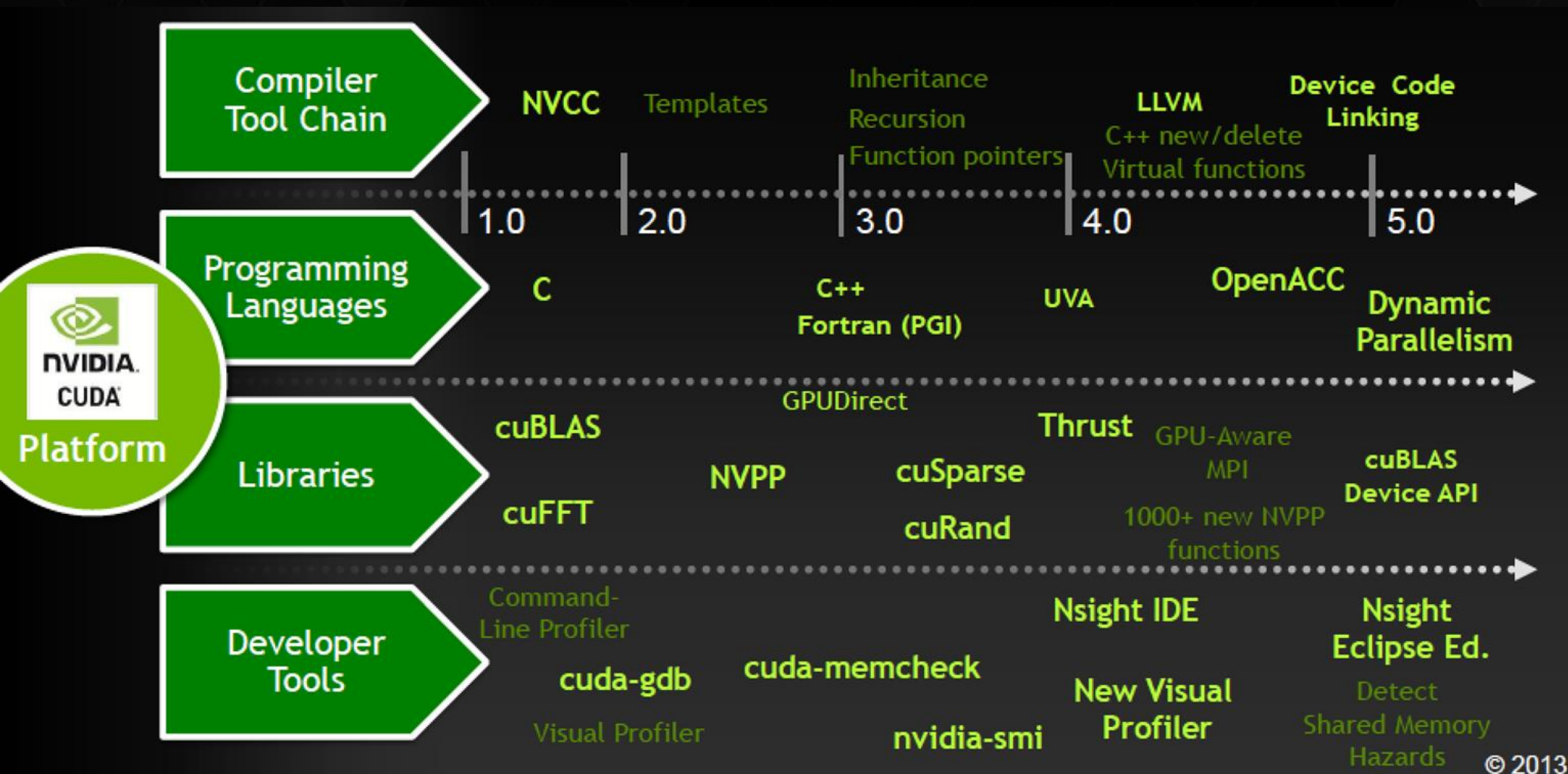
Sr. Solution Architect, NVIDIA EMEA

CUDA: COMMON UNIFIED DEVICE ARCHITECTURE

- Parallel computing architecture and programming model
- Includes a CUDA C compiler, support for OpenCL and DirectCompute
- Architected to natively support multiple computational interfaces (standard languages and APIs)



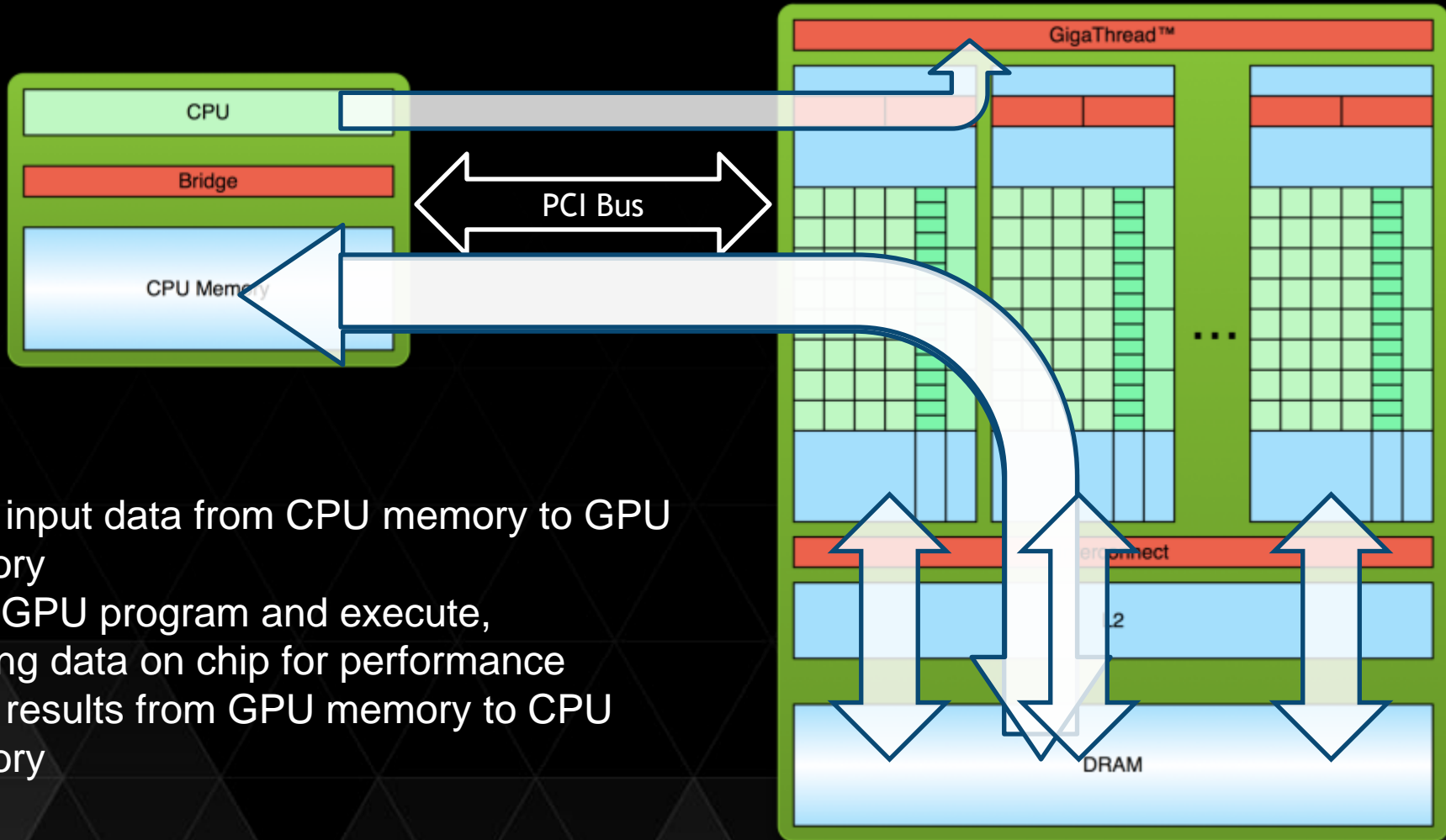
NVIDIA CUDA EVOLUTION



The background features a green grid pattern that is distorted by black wavy shapes, creating a sense of depth and movement. The grid lines are thin and closely spaced, while the black shapes are thick and fluid, resembling liquid or smoke. The overall effect is a dynamic and modern aesthetic.

Computing Model

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

CUDA Kernels



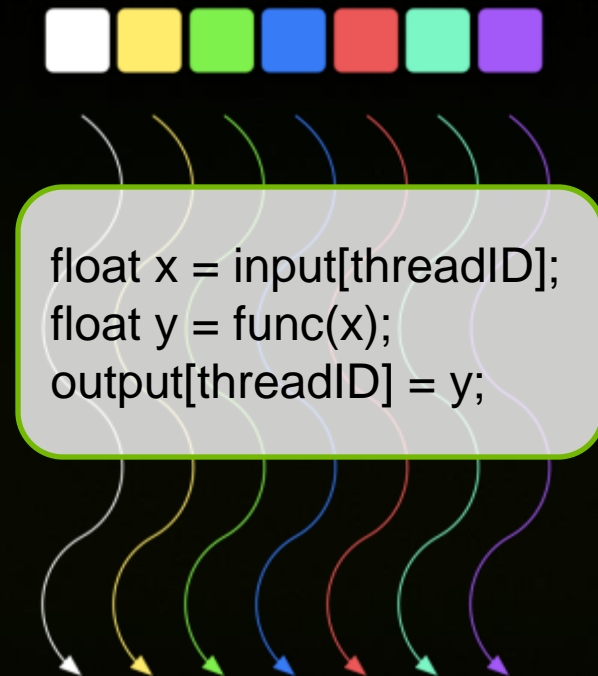
- **Parallel portion of application: execute as a **kernel****
 - Entire GPU executes kernel, many threads
- **CUDA threads:**
 - Lightweight
 - Fast switching
 - 1000s execute simultaneously

CPU	Host	Executes functions
GPU	Device	Executes kernels

CUDA Kernels: Parallel Threads



- A **kernel** is an array of threads, executed in parallel
- All threads execute the same code
- Each thread has an ID
 - Select input/output data
 - Control decisions

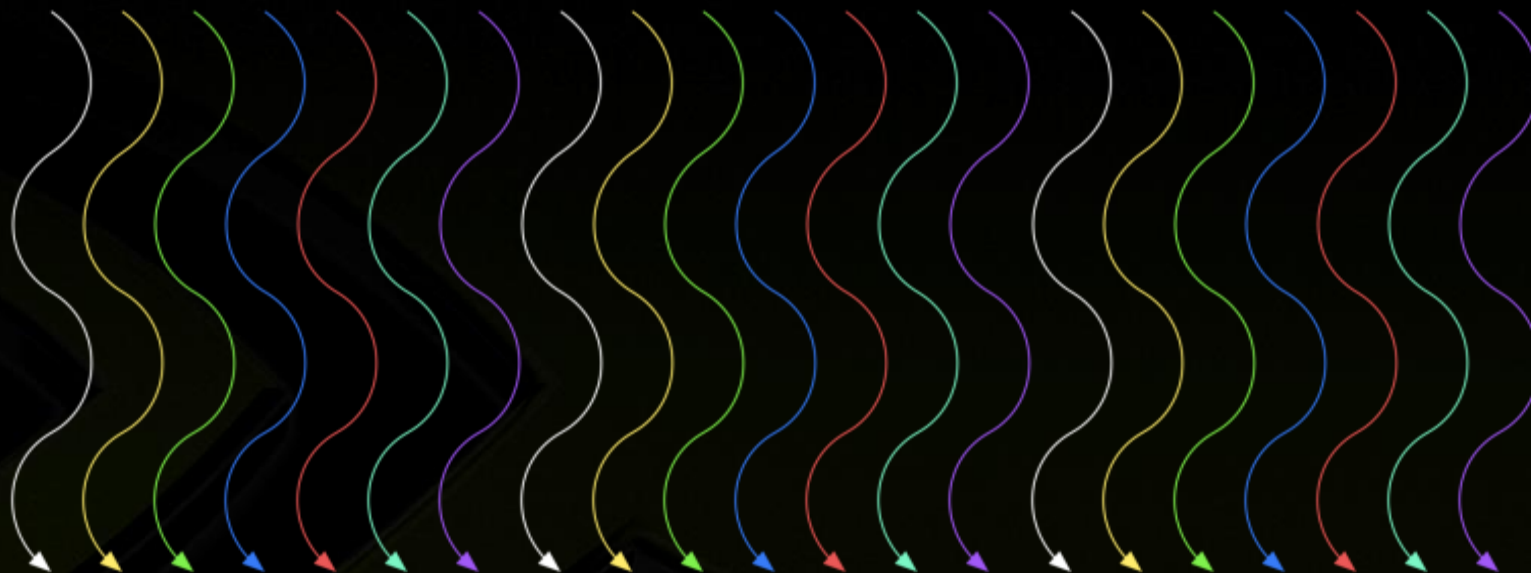


Key Idea of CUDA



- **Write a single-threaded program parameterized in terms of the thread ID.**
- **Use the thread ID to select a subset of the data for processing, and to make control flow decisions.**
- **Launch a number of threads, such that the ensemble of threads processes the whole data set.**

CUDA Kernels: Subdivide into Blocks

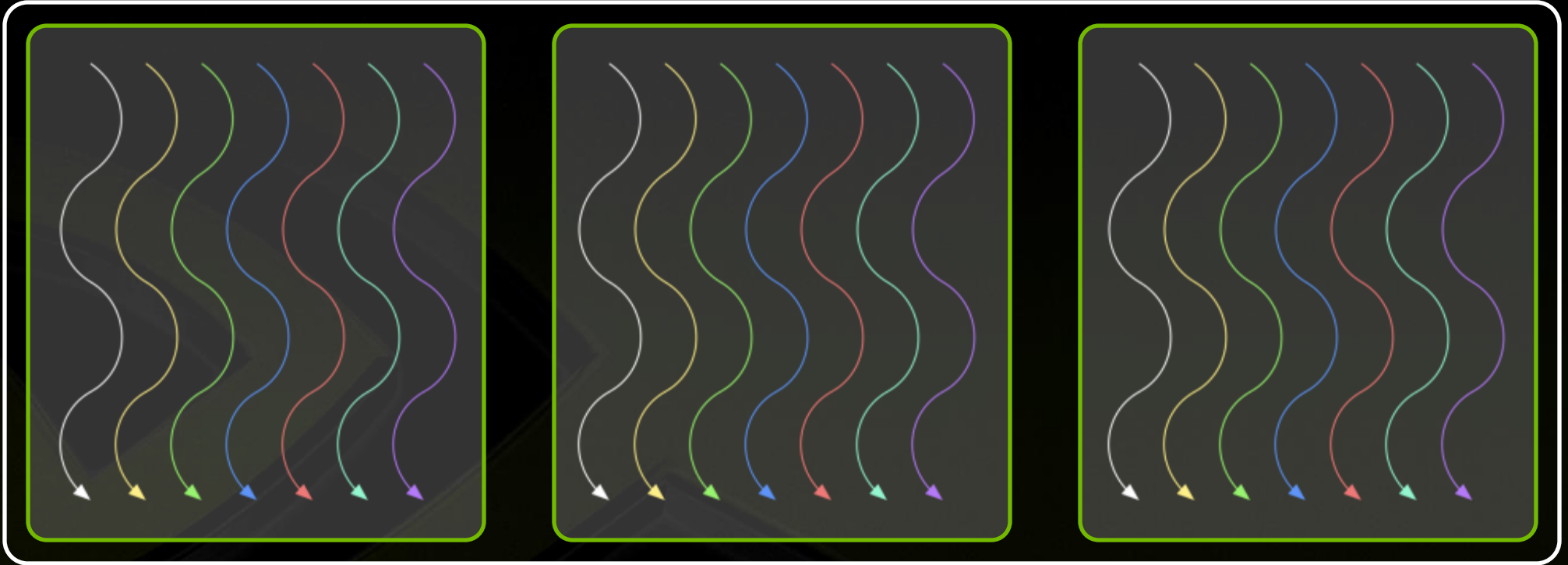


CUDA Kernels: Subdivide into Blocks



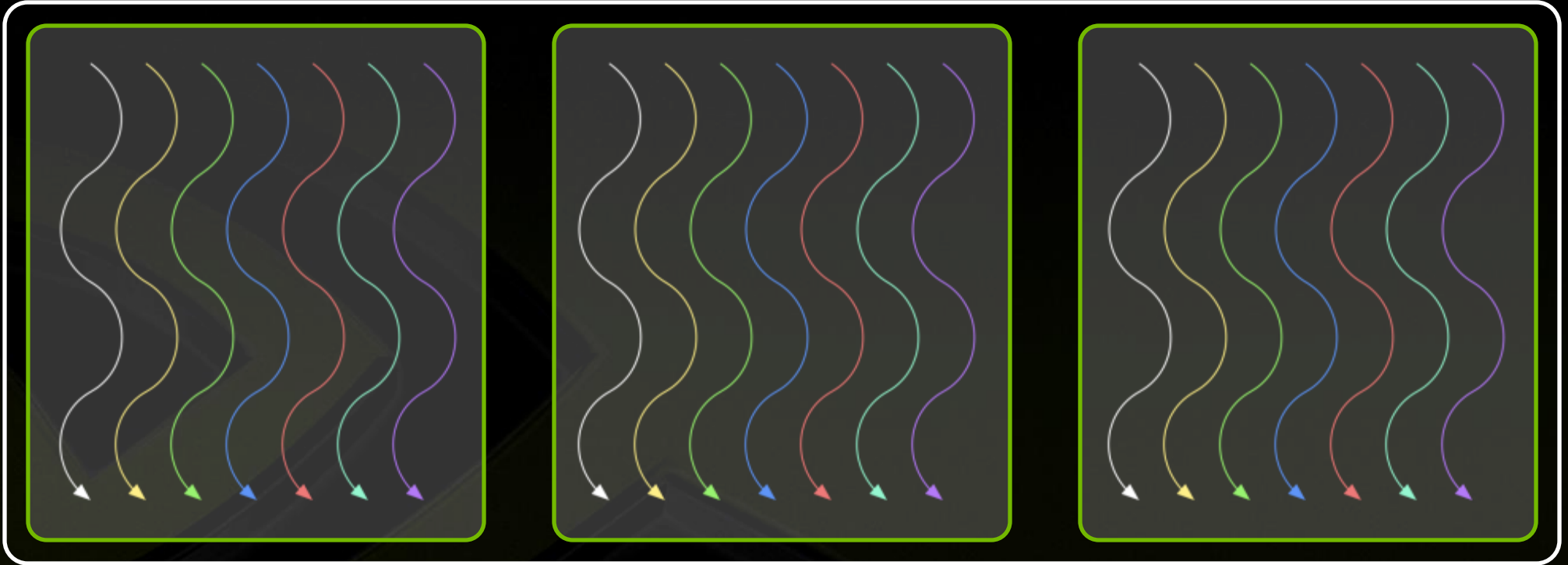
- Threads are grouped into **blocks**

CUDA Kernels: Subdivide into Blocks



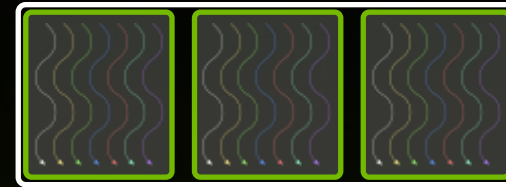
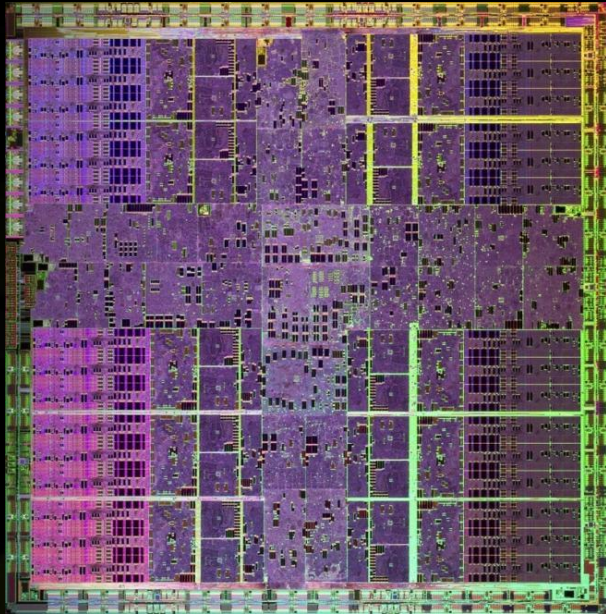
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**

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**

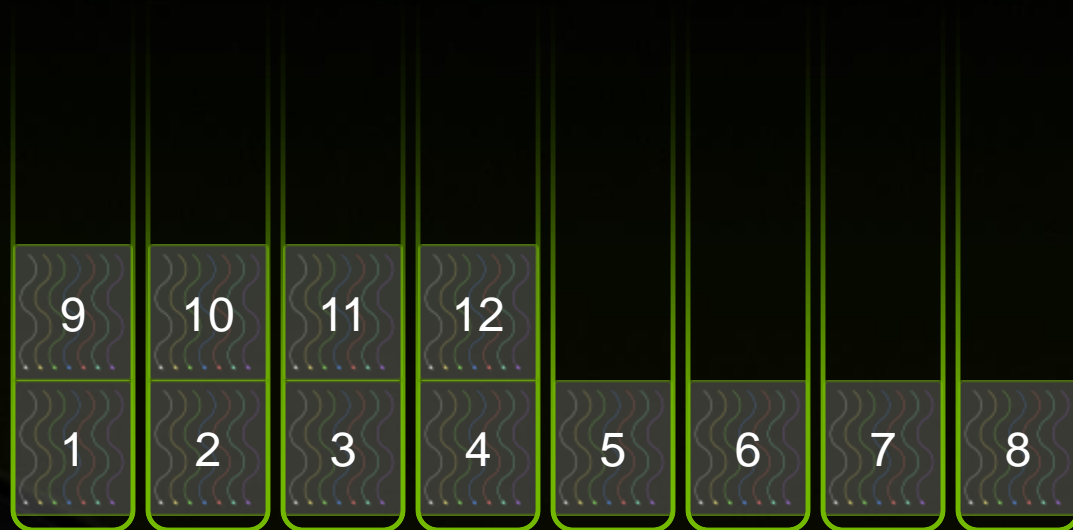
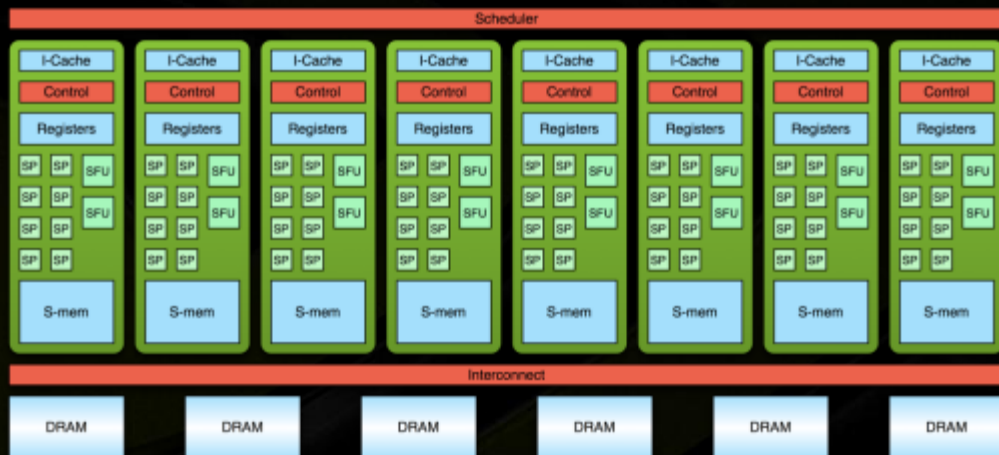
Communication Within a Block

- **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

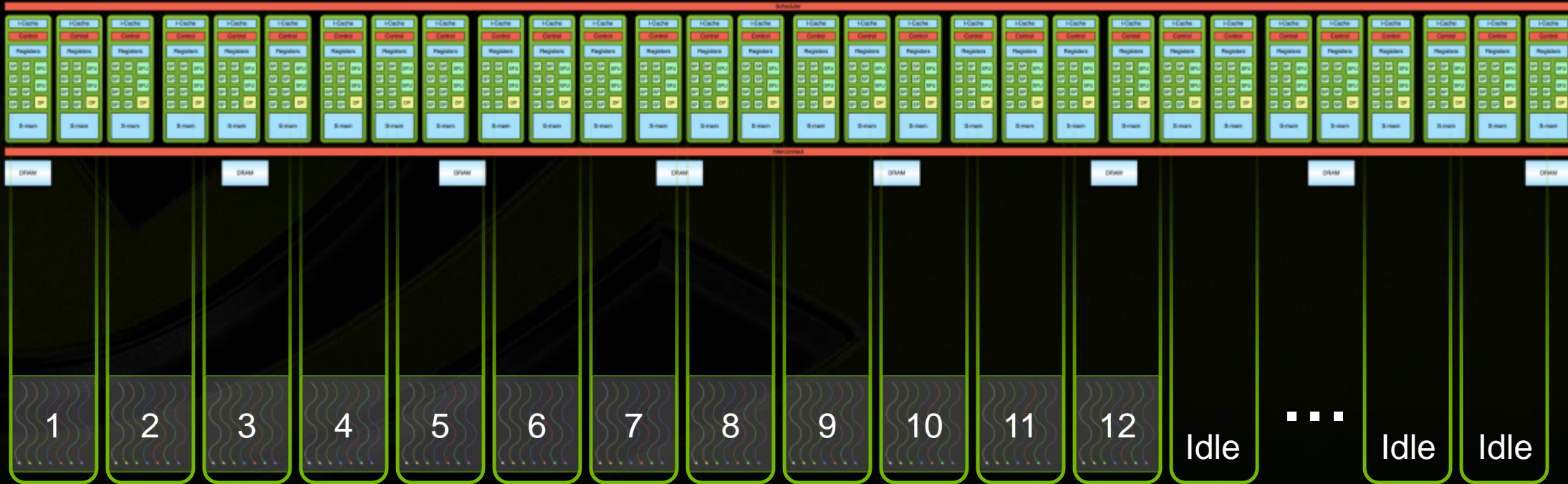
Transparent Scalability – G84



Transparent Scalability – G80



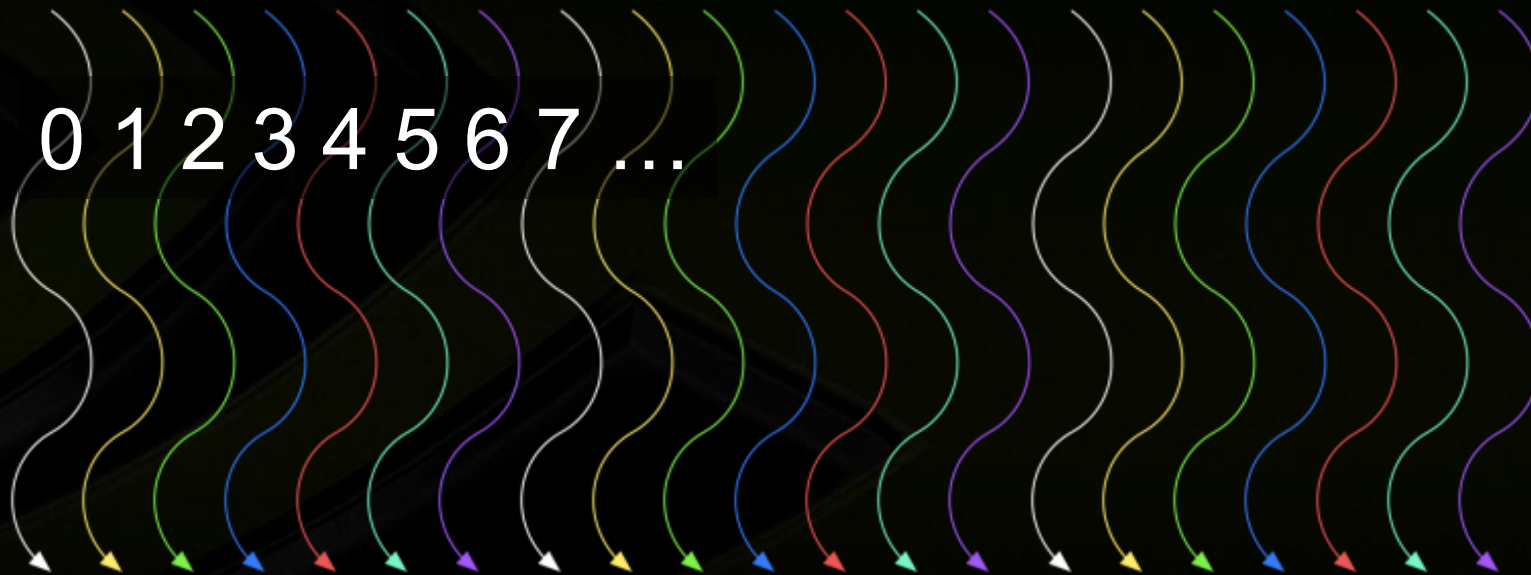
Transparent Scalability – GT200



Numbering of Threads



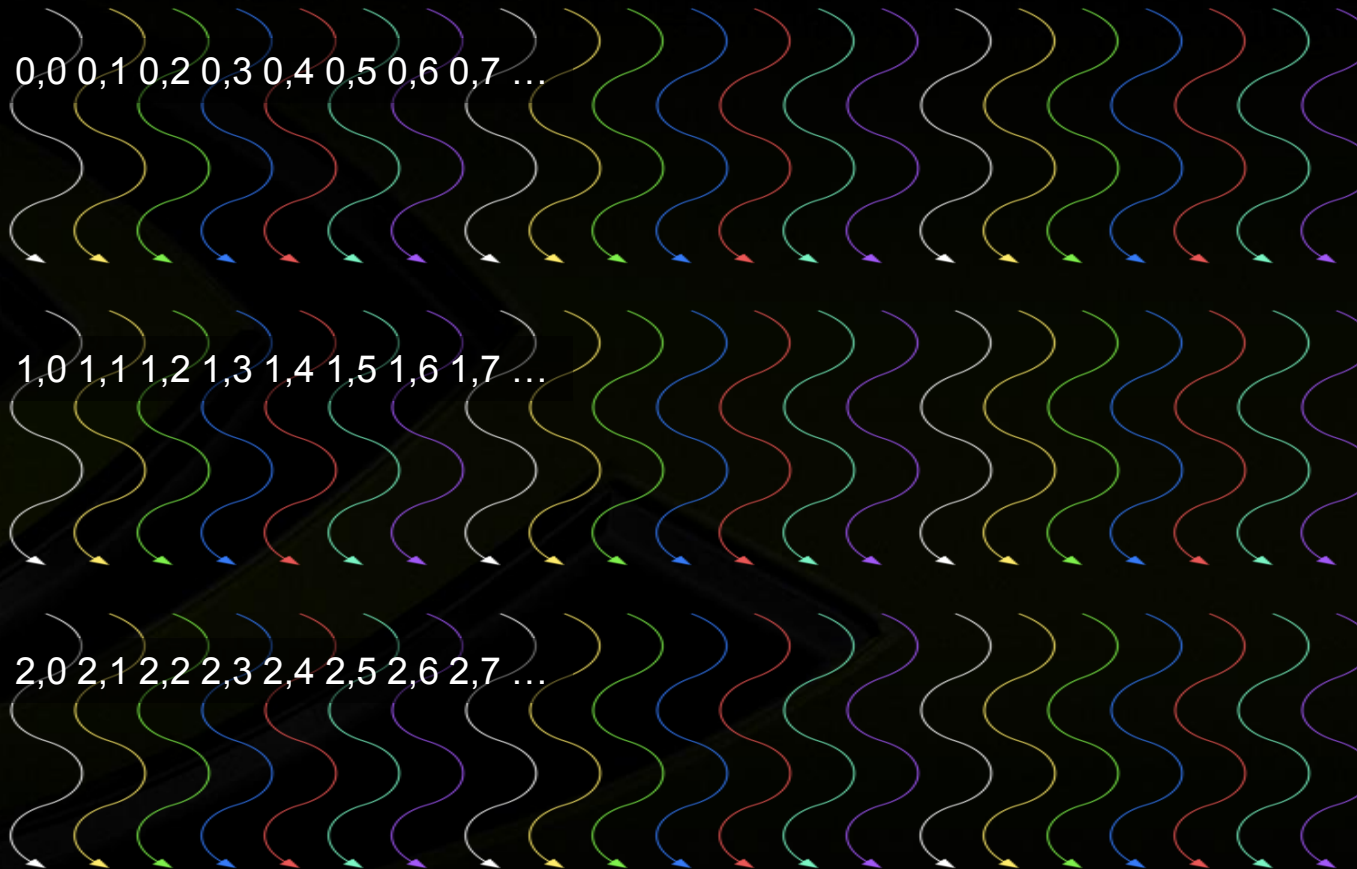
1-dimensional indexing



Numbering of Threads



2-dimensional indexing



Numbering of Threads

Or 3-dimensional indexing

0,0,0 0,0,1 0,0,2 0,0,3 0,0,4 0,0,5 0,0,6 0,0,7 ...
0,1,0 0,1,1 0,1,2 0,1,3 0,1,4 0,1,5 0,1,6 0,1,7 ...

■ ■ ■

1,0,0 1,0,1 1,0,2 1,0,3 1,0,4 1,0,5 1,0,6 1,0,7 ...
1,1,0 1,1,1 1,1,2 1,1,3 1,1,4 1,1,5 1,1,6 1,1,7 ...

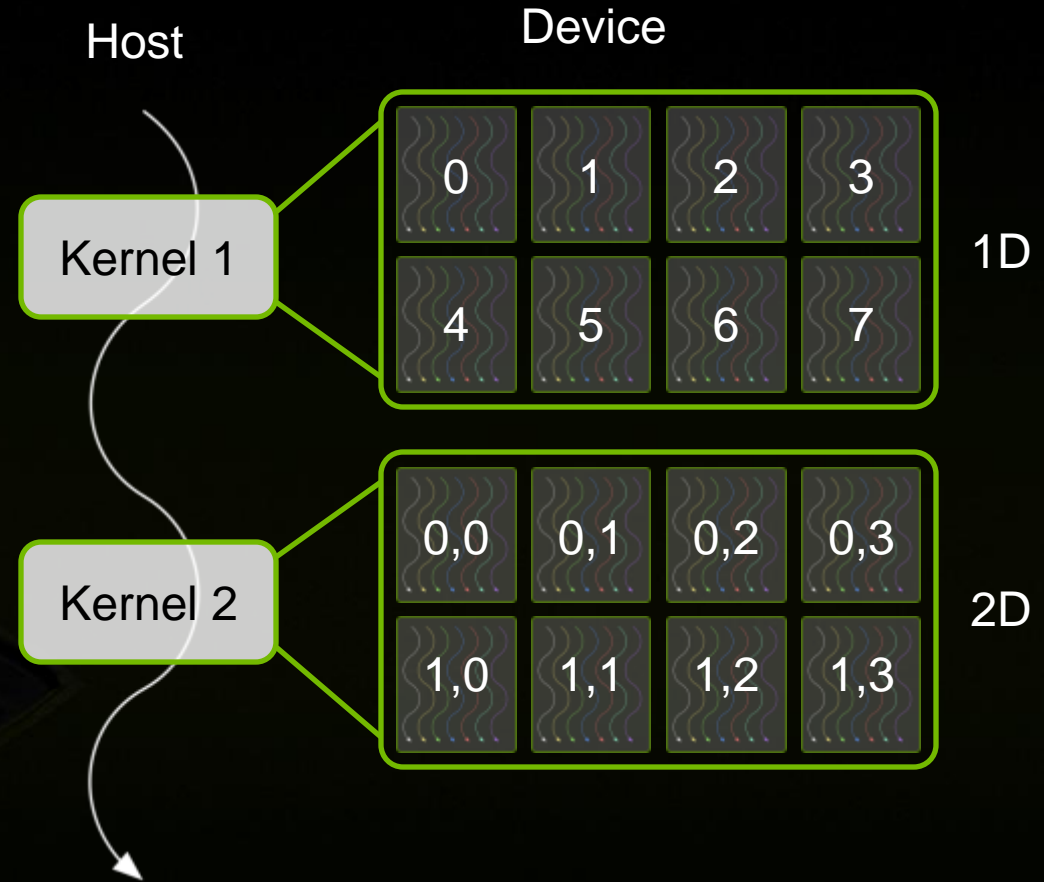
Numbering of Blocks



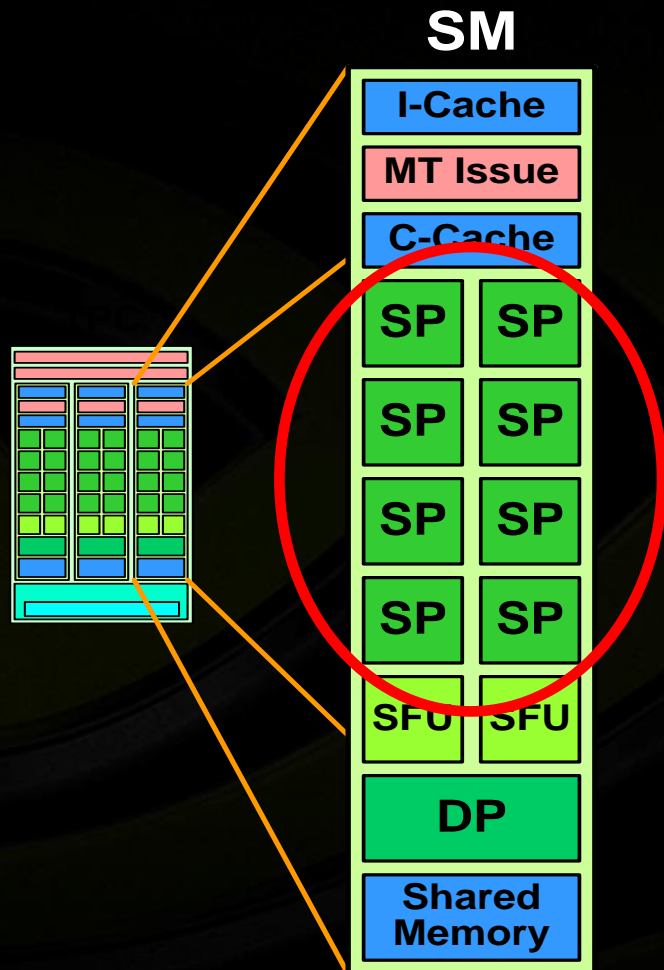
CUDA Programming Model - Summary



- 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



Single-Instruction, Multiple-Thread Execution

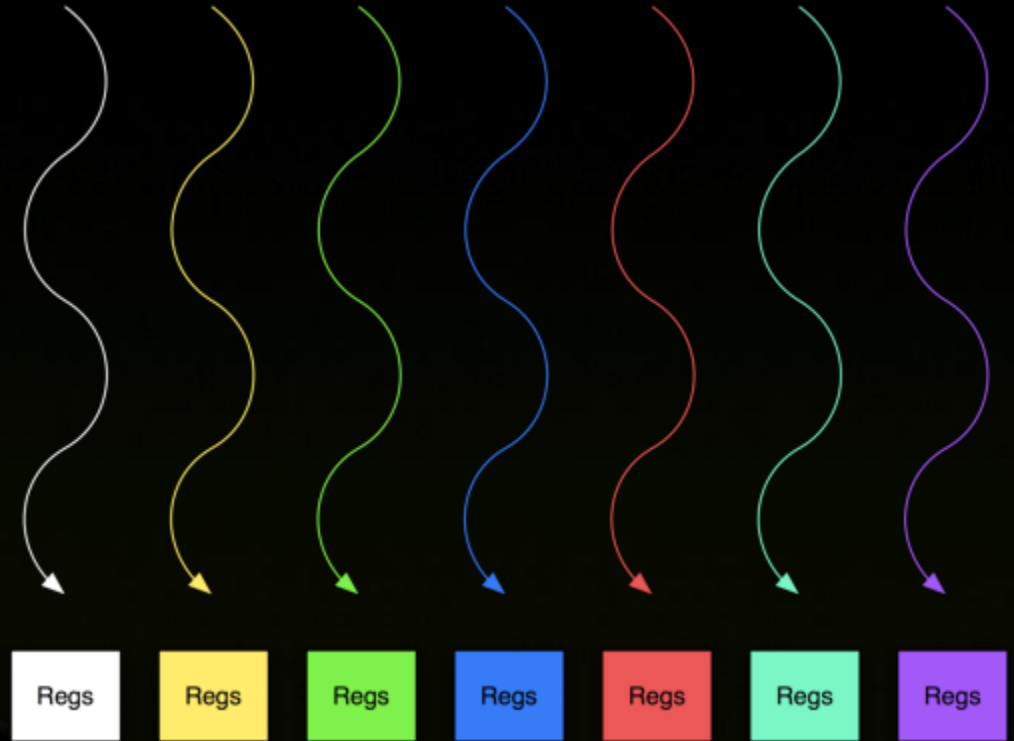


- Warp: set of 32 parallel threads that execute together in single-instruction, multiple-thread mode (SIMT) on a streaming multiprocessor (SM)
- SM hardware implements zero-overhead warp and thread scheduling
- Threads can execute independently
- SIMT warp diverges and converges when threads branch independently
- Best efficiency and performance when threads of a warp execute together, so no penalty if all threads in a warp take same path of execution
- Each SM executes up to 1024 concurrent threads, as 32 SIMT warps of 32 threads

Memory Model

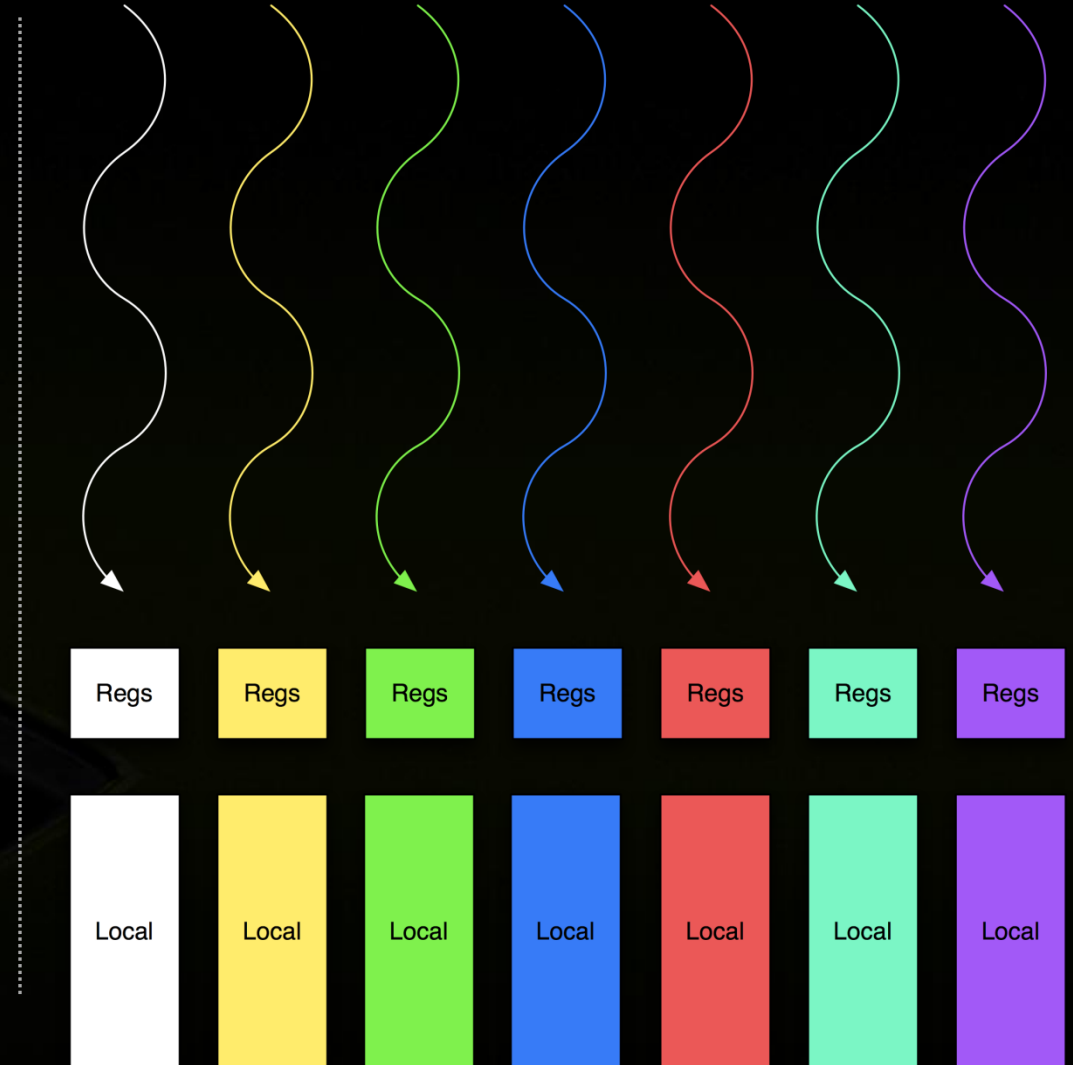
Memory hierarchy

- Thread:
 - Registers



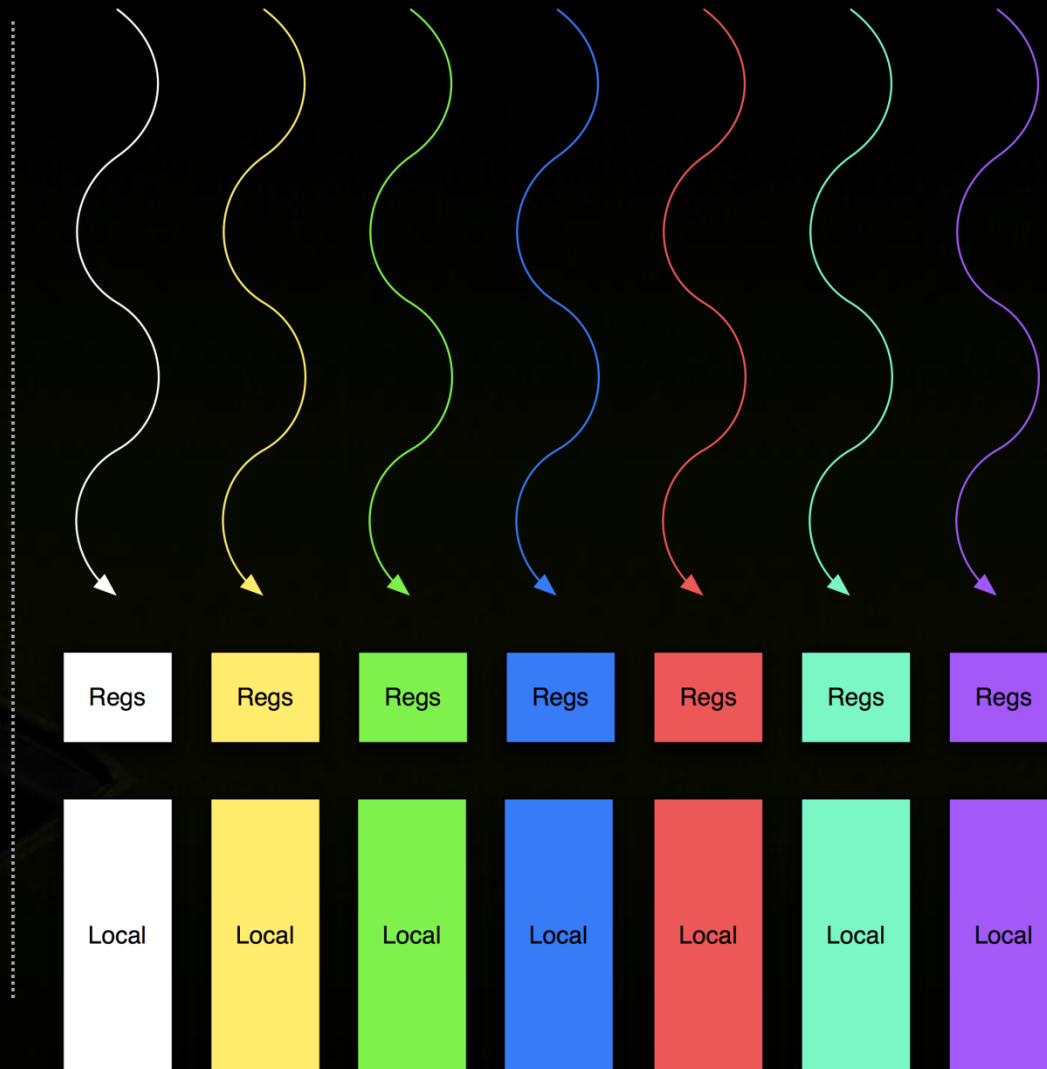
Memory hierarchy

- Thread:
 - Registers
- Thread:
 - Local memory



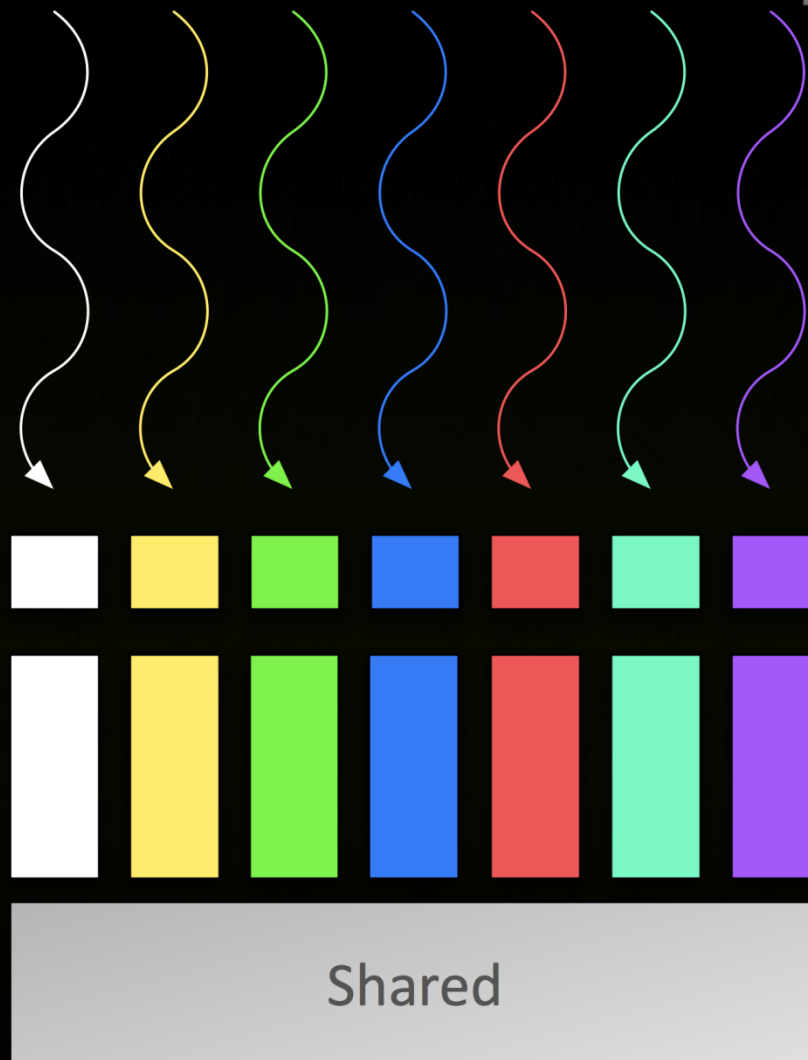
Memory hierarchy

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



Memory hierarchy

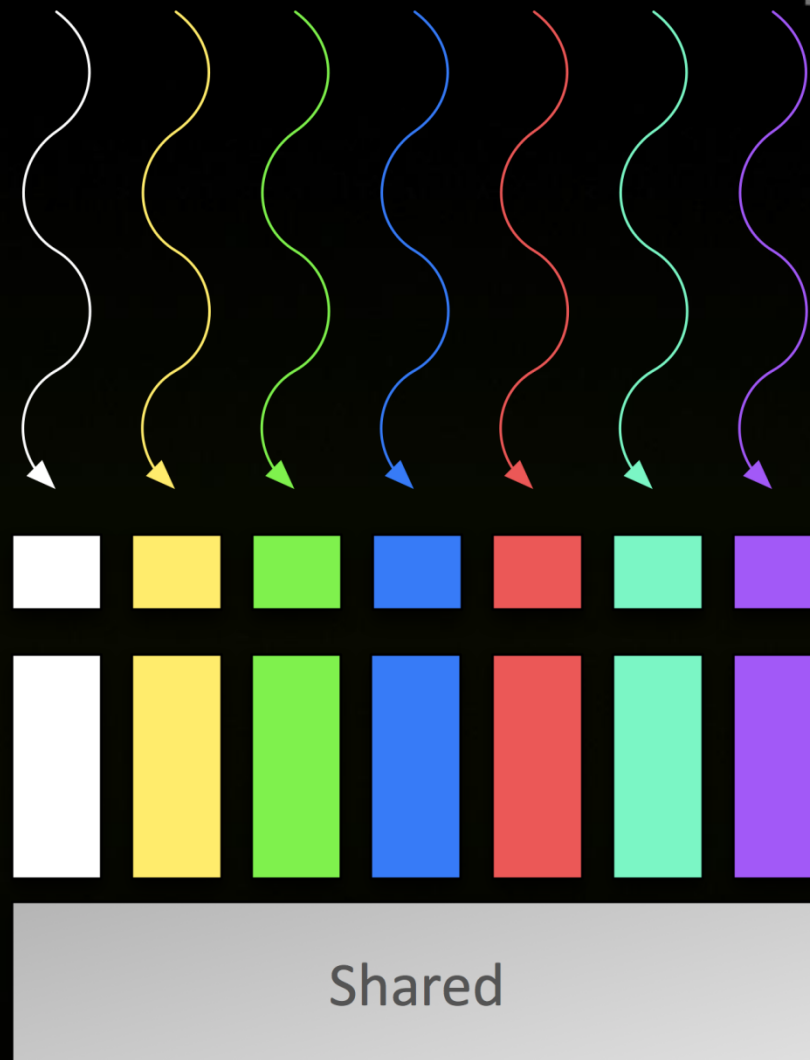
- Thread:
 - Registers
- Thread:
 - Local memory
- Block of threads:
 - Shared memory



Memory hierarchy



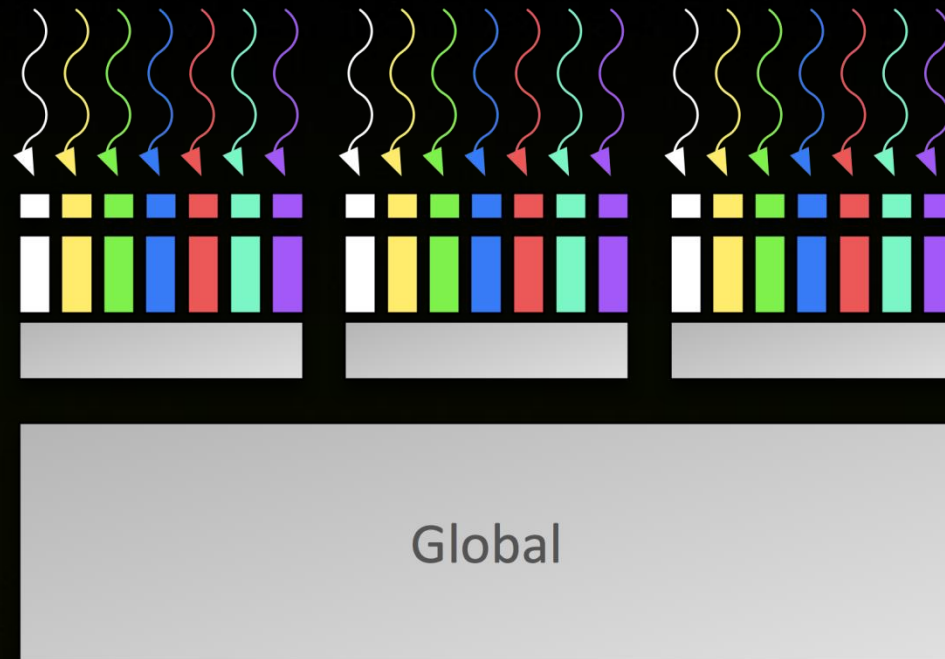
- Thread:
 - Registers
- Thread:
 - Local memory
- Block of threads:
 - Shared memory
- All blocks:
 - Global memory



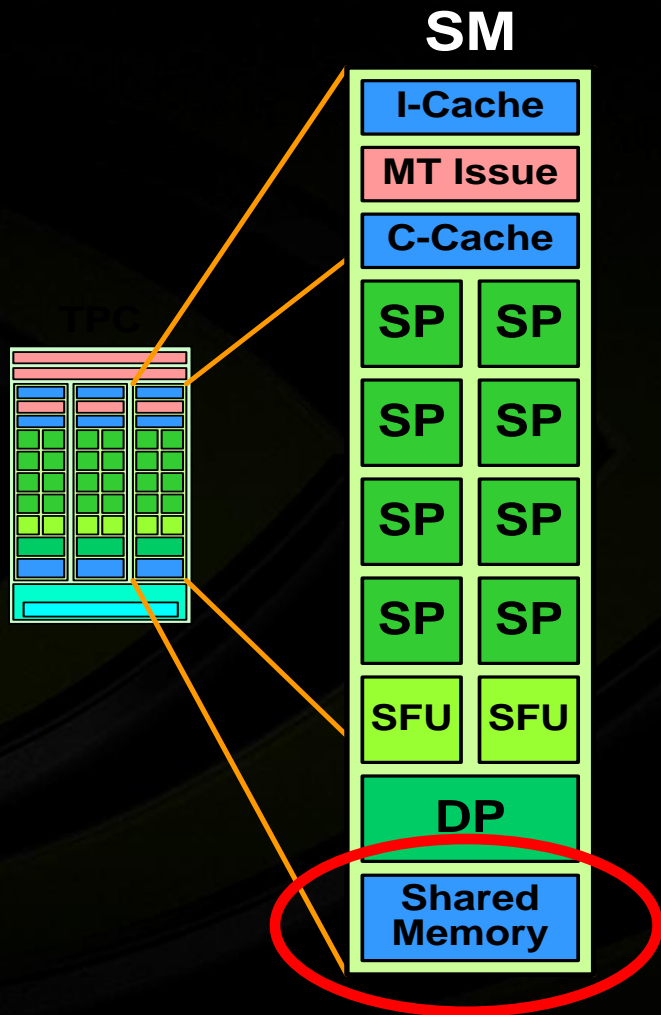
Memory hierarchy



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

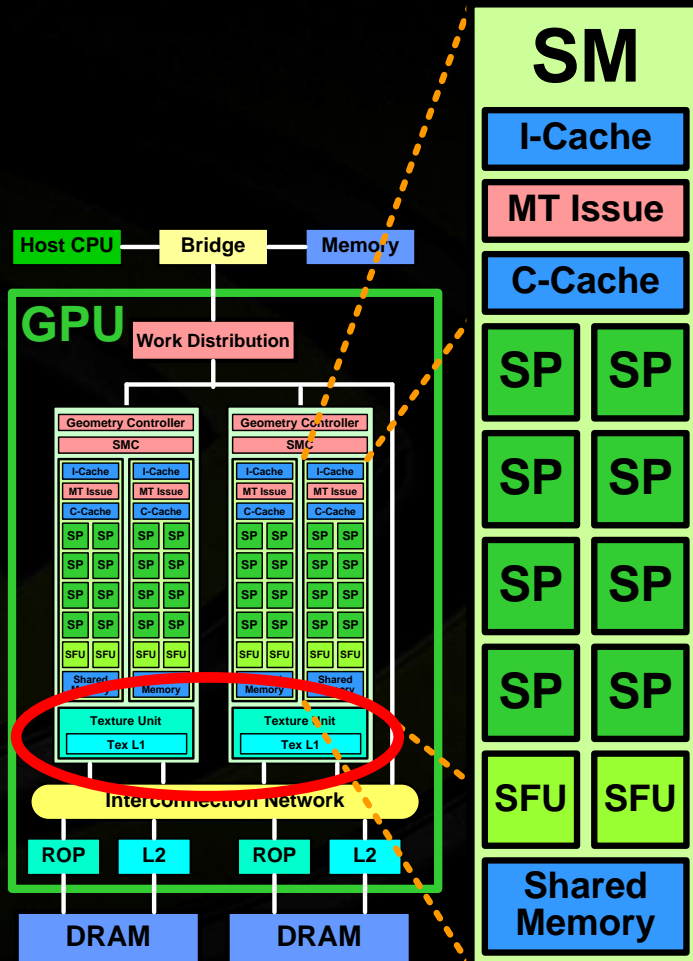


Shared Memory



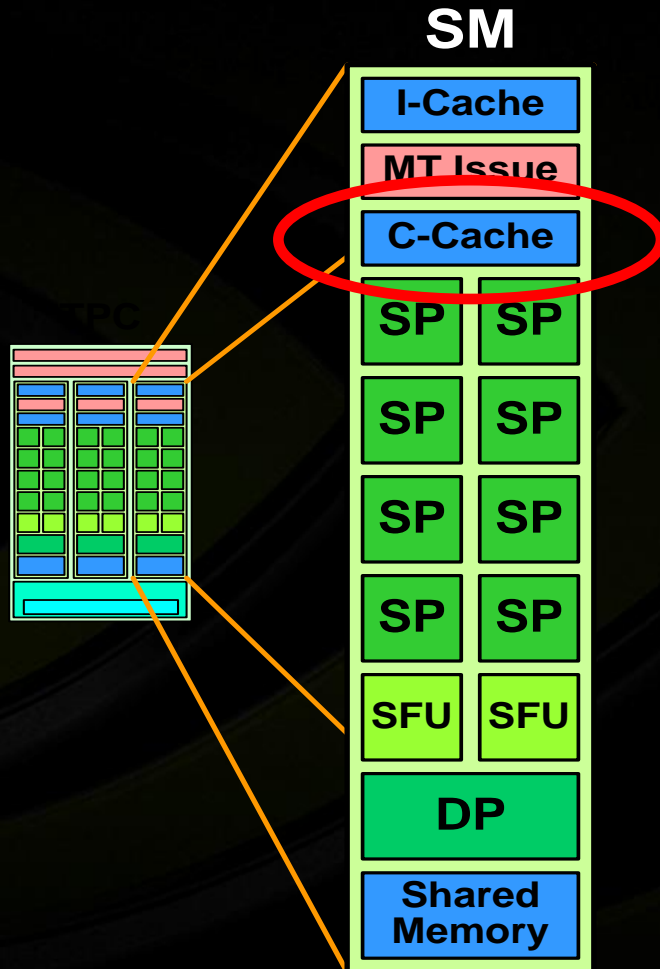
- More than 1 Tbyte/sec aggregate memory bandwidth
- Use it
 - As a cache
 - To reorganize global memory accesses into coalesced pattern
 - To share data between threads
- 16 kbytes per SM

Texture Memory



- Texture is an object for reading data
- Data is cached
- Host actions
 - Allocate memory on GPU
 - Create a texture memory reference object
 - Bind the texture object to memory
 - Clean up after use
- GPU actions
 - Fetch using texture references `tex1Dfetch()`, `tex1D()`, `tex2D()`, `tex3D()`

Constant Memory



- Write by host, read by GPU
- Data is cached
- Useful for tables of constants

Memory Spaces



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

The background features a green grid pattern that is distorted by black wavy shapes, creating a sense of depth and movement. The grid lines are thin and closely spaced, while the black shapes are thick and fluid, resembling liquid or smoke. The overall effect is a dynamic and modern aesthetic.

CUDA C

CUDA C — C with Language Extensions



- **Function qualifiers**

```
__global__ void MyKernel() {}           // call from host, execute on GPU
__device__ float MyDeviceFunc() {}      // call from GPU, execute on GPU
__host__ int HostFunc() {}              // call from host, execute on host
```

- **Variable qualifiers**

```
__device__ float MyGPUArray[32];        // in GPU memory space
__constant__ float MyConstArray[32];    // write by host; read by GPU
__shared__ float MySharedArray[32];     // shared within thread block
```

- **Built-in vector types**

```
int1, int2, int3, int4
float1, float2, float3, float4
double1, double2
etc.
```

CUDA C — C with Language Extensions



- **Execution configuration**

```
dim3 dimGrid(100, 50);           // 5000 thread blocks
dim3 dimBlock(4, 8, 8);          // 256 threads per block
MyKernel <<< dimGrid, dimBlock >>> (...); // Launch kernel
```

- **Built-in variables and functions valid in device code:**

```
dim3 gridDim;           // Grid dimension
dim3 blockDim;          // Block dimension
dim3 blockIdx;          // Block index
dim3 threadIdx;         // Thread index
void __syncthreads();    // Thread synchronization
```

CUDA C — C with Runtime Extensions

- **Device management:**
`cudaGetDeviceCount()`, `cudaGetDeviceProperties()`
- **Device memory management:**
`cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- **Texture management:**
`cudaBindTexture()`, `cudaBindTextureToArray()`
- **Graphics interoperability:**
`cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`

SAXPY: Device Code

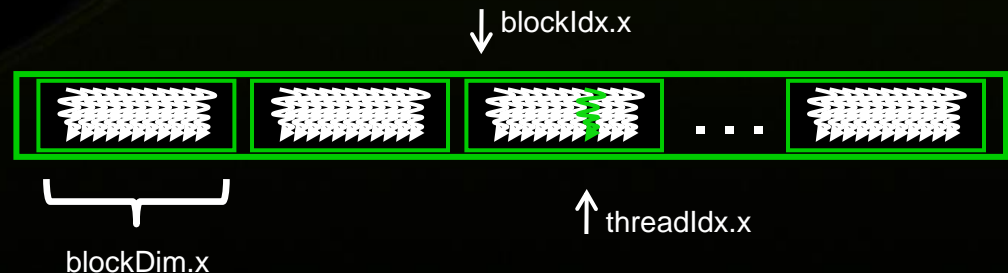


```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

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];
}
```

Parallel C Code



SAXPY: Host Code

```
// Allocate two N-vectors h_x and h_y
int size = N * sizeof(float);
float* h_x = (float*)malloc(size);
float* h_y = (float*)malloc(size);

// Initialize them...

// Allocate device memory
float* d_x; float* d_y;
cudaMalloc((void**)&d_x, size);
cudaMalloc((void**)&d_y, size);

// Copy host memory to device memory
cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice);

// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (N + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(N, 2.0, d_x, d_y);

// Copy result back from device memory to host memory
cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost);
```


Launching a Kernel

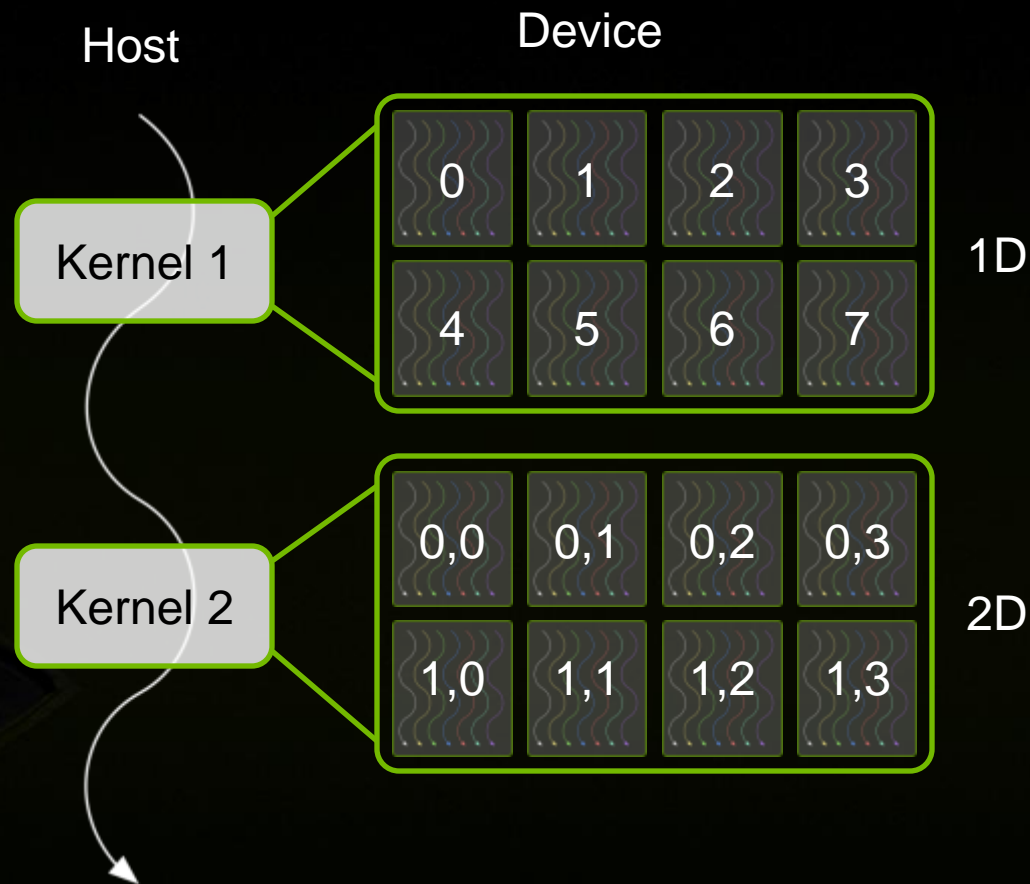


- **Call a kernel with**

```
Func <<<Dg,Db,Ns,S>>> (params);  
dim3 Dg(mx,my,1); // grid spec  
dim3 Db(nx,ny,nz); // block spec  
size_t Ns; // shared memory  
cudaStream_t S; // CUDA stream
```
- **Execution configuration is passed to kernel with built-in variables**

```
dim3 gridDim, blockDim, blockIdx,  
threadIdx;
```
- **Extract components with**

```
threadIdx.x, threadIdx.y,  
threadIdx.z, etc.
```




Execution Configuration



```
vectorAdd <<< BLOCKS, THREADS_PER_BLOCK >>> (N, 2.0, d_x, d_y);
```

- How many blocks?
 - At least one block per SM to keep every SM occupied
 - At least two blocks per SM so something can run if block is waiting for a synchronization to complete
 - Many blocks for scalability to larger and future GPUs
- How many threads?
 - At least 192 threads per SM to hide read after write latency of 11 cycles (not necessarily in same block)
 - Use many threads to hide global memory latency
 - Too many threads exhausts registers and shared memory
 - Thread count a multiple of warp size
 - Typically, between 64 and 256 threads per block



```
x = y + 5;  
z = x + 3;
```

Expensive Operations

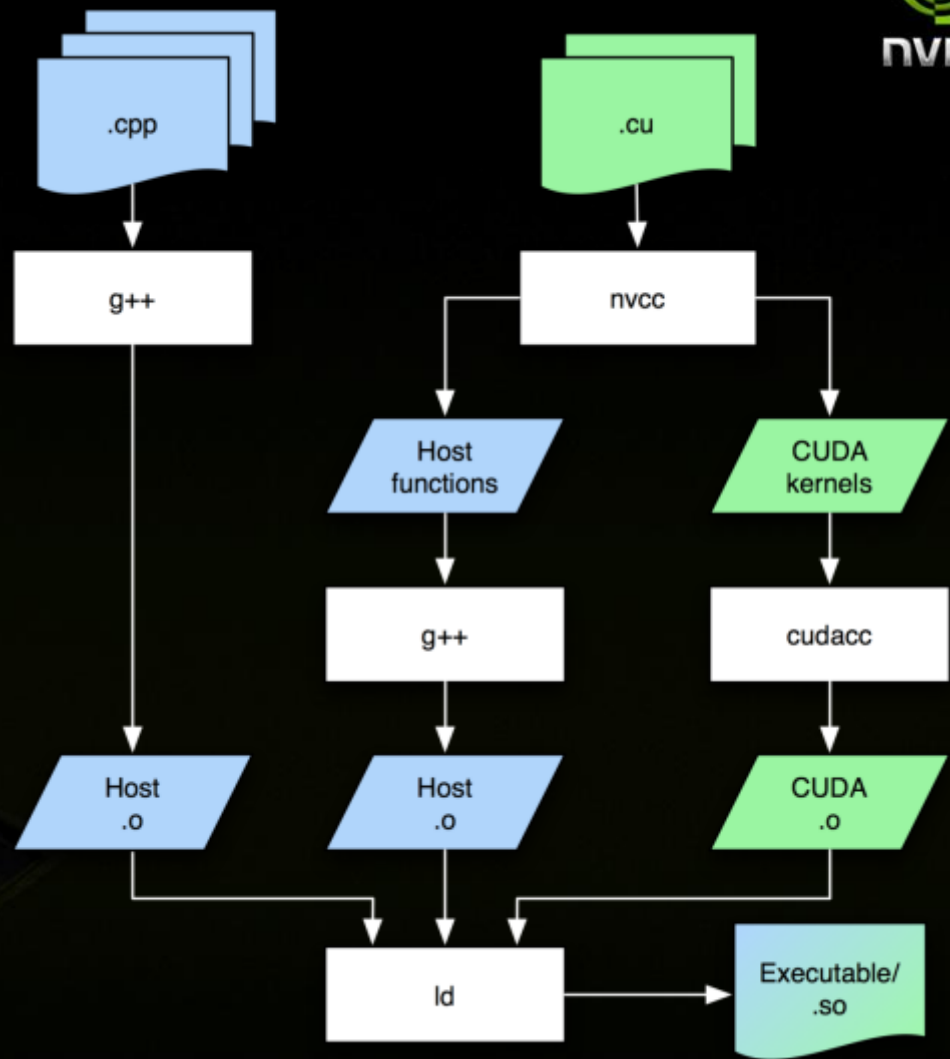


- `sin()`, `exp()` etc.; faster, less accurate versions are `__sin()`, `__exp()` etc.
- Integer division and modulo; avoid if possible; replace with bit shift operations for powers of 2
- Branching where threads of warp take differing paths of control flow

Compilation

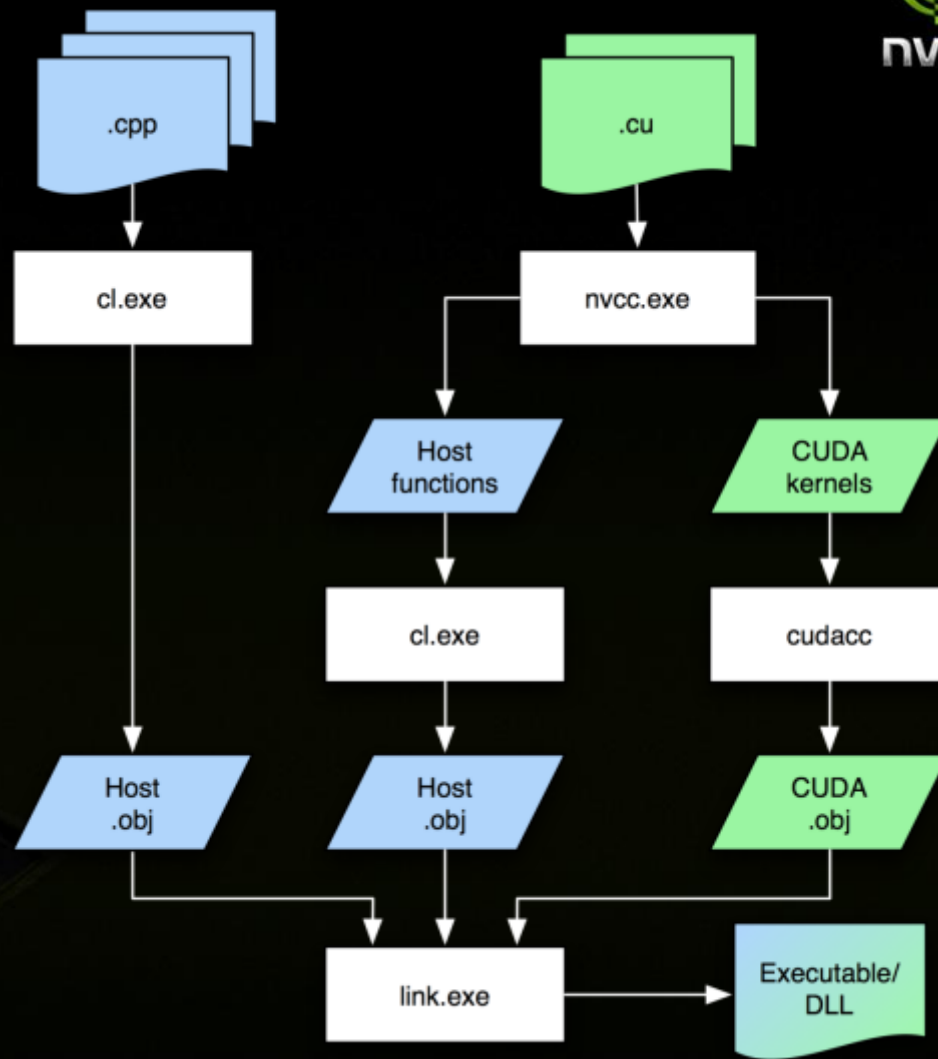
Linux

- **Separate file types**
 - .c/.cpp for host code
 - .cu for device/mixed code
- **Typically makefile driven**
- **cuda-gdb, Allinea DDT, TotalView for debugging**
- **CUDA Visual Profiler**



Visual Studio

- **Separate file types**
 - .c/.cpp for host code
 - .cu for device/mixed code
- **Compilation rules: cuda.rules**
 - Syntax highlighting
 - Intellisense
- **Integrated debugger and profiler: Nsight**



Compilation Commands

- **nvcc <filename>.cu [-o <executable>]**
 - Builds release code
- **nvcc -g <filename>.cu**
 - Builds debug CPU code
- **nvcc -G <filename>.cu**
 - Builds debug GPU code
- **nvcc -O <level> <filename>.cu**
 - Builds optimised GPU code



THANKS!

Carlo Nardone

+39 335 5828197

cnardone@nvidia.com

