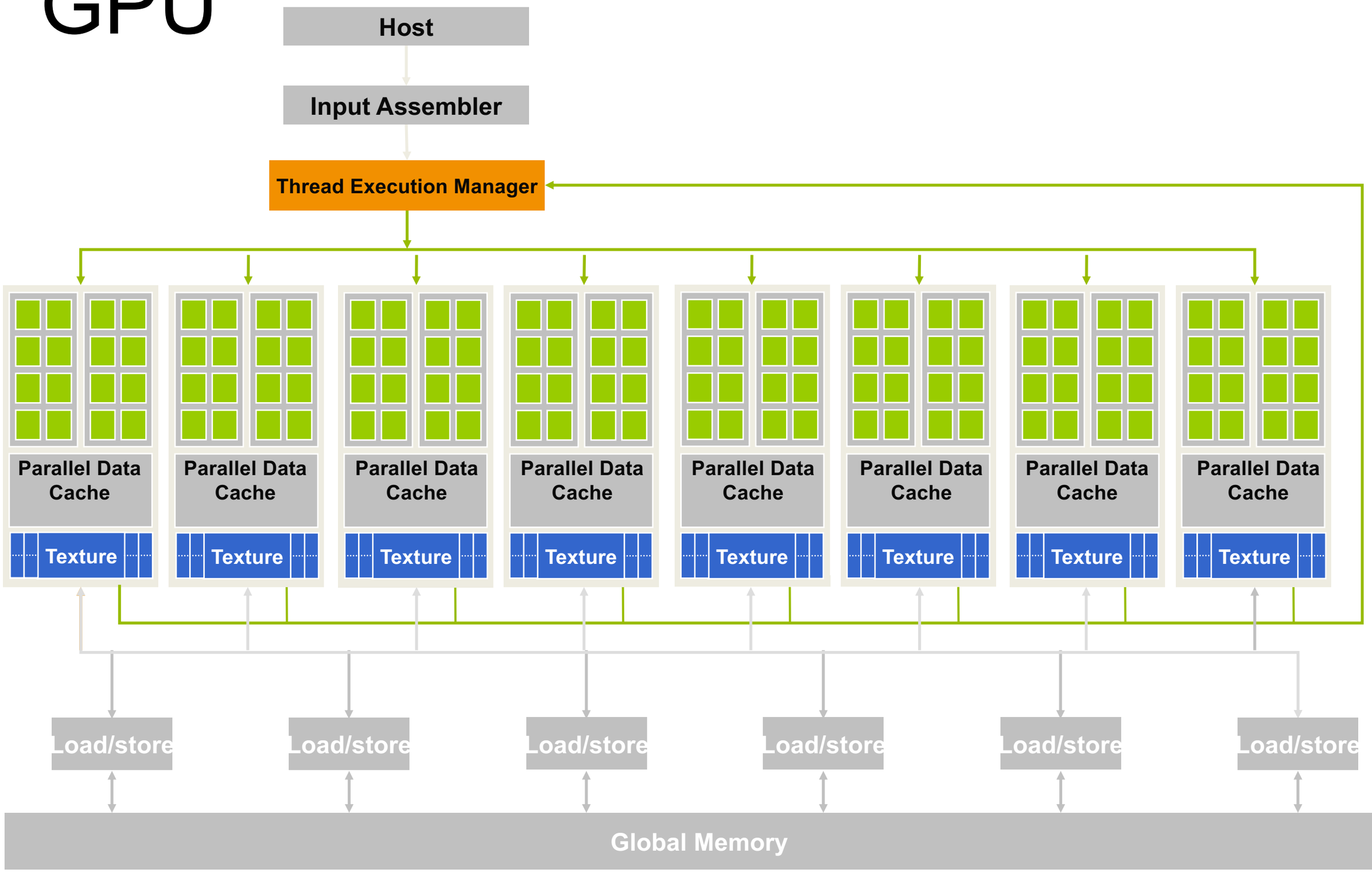
CUDA Thread Basics

CUDA-Intro

- After 5yrs of GeForece 3 sereis
- In November 2006..NVIDIA unveiled Industry's first DIrectx10 GPU..GeForce 8800GTX
- First GPU to built on CUDA Architecture
- This has special new components designed strictly for GPU Computing and alleviated problems towards GPGPU
- ALUs were complied with IEEE standards
- Execution units were allowed to do arbitrary read and write to memory and s/w manged cache or shared memory
- Using CUDA: CUDA C = Stnadard C + Extensions of CUDA
- Tesla—HPC and Super Computing applications
- GeForce-Entertainment
- Quadro-Professional Visualization

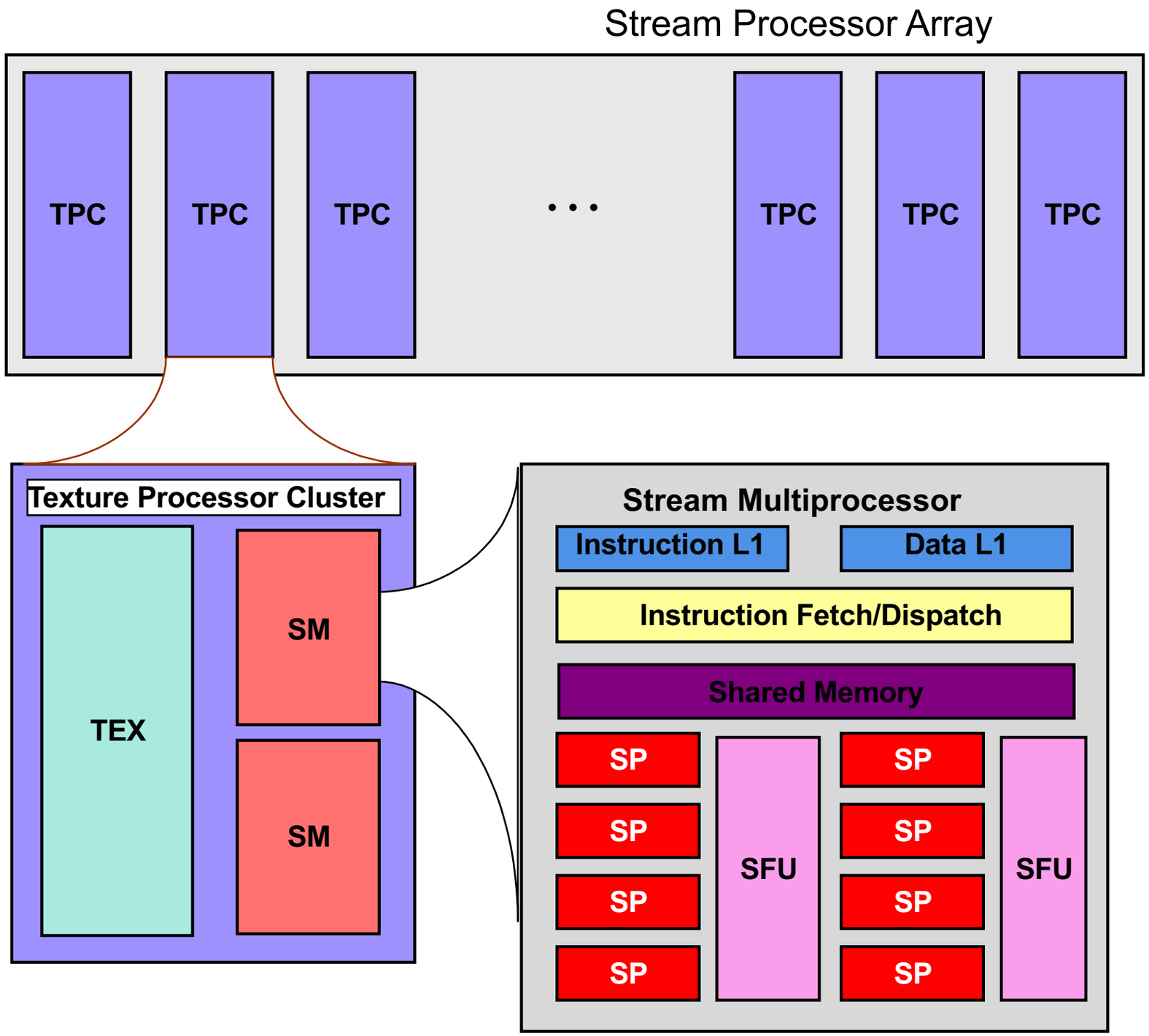
Architecture of a CUDA Capable GPU



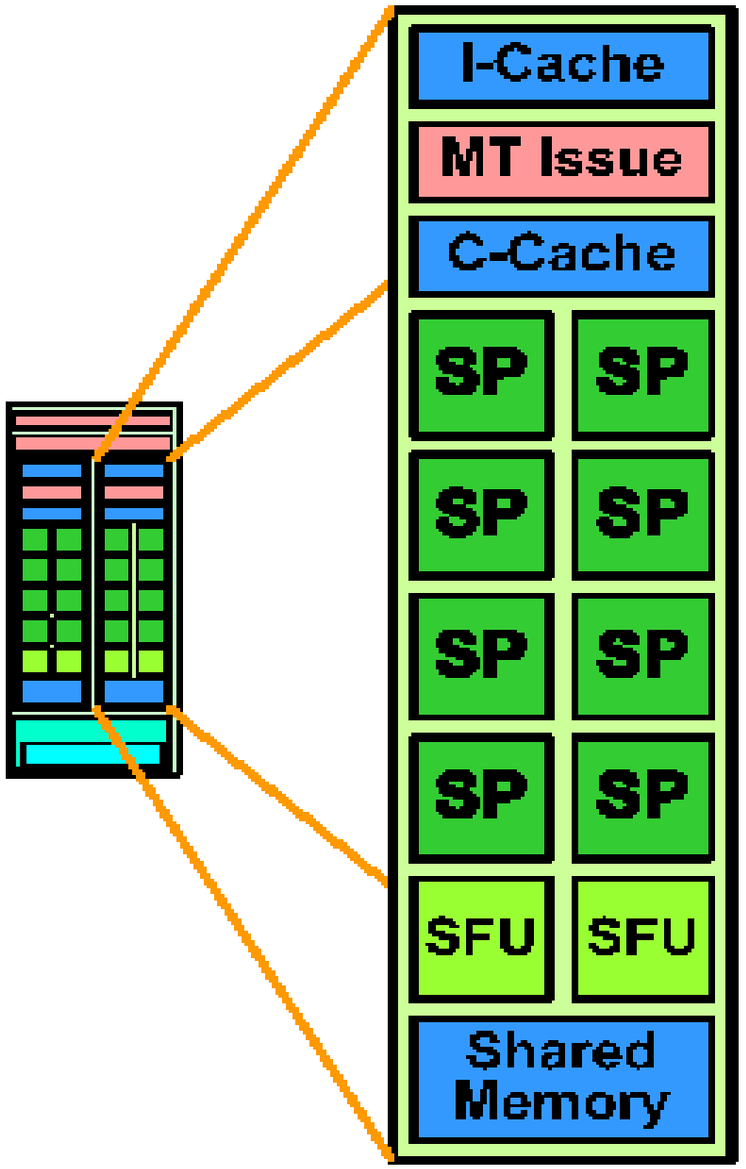
Architecture of GPU

- Organized into CUDA capable Streaming Multiprocessors(SMs)
- 2 SMs form a building block
- Each SM has # SPs (Streaming Processors) that share control logic and instruction cache
- GPU has GDDR-DRAM-global memory
- G80 has 128SPs—16SMs each with 8SPs
- Each SP has MAD-Multiply-add unit with additional multiply unit
- SFUs –to perform floating point operations like SQRT
- GT80-128SPs-500GFLOPs, GT200-240SPs-1TeraFlop
- GT80-768 threads per SM—12000 threads total
- GT80 has 8 blocks that can be launched per SM
- GT200-1024 threads per SM-30000 threads in total

GeForce-8 Series HW Overview



SM Multithreaded Multiprocessor



- SM has 8 SP Thread Processors
  - IEEE 754 32-bit floating point
  - 32-bit and 64-bit integer
  - 8K 32-bit registers
- SM has 2 SFU Special Function Units
- Scalar ISA
  - Memory load/store, texture fetch
  - Branch, call, return
  - Barrier synchronization instruction
- Multithreaded Instruction Unit
  - 768 Threads, hardware multithreaded
  - 24 SIMT warps of 32 threads
  - Independent thread execution
  - Hardware thread scheduling
- 16KB Shared Memory
  - Concurrent threads share data
  - Low latency load/store

Scheduling on the HW

Grid is launched on the SPA

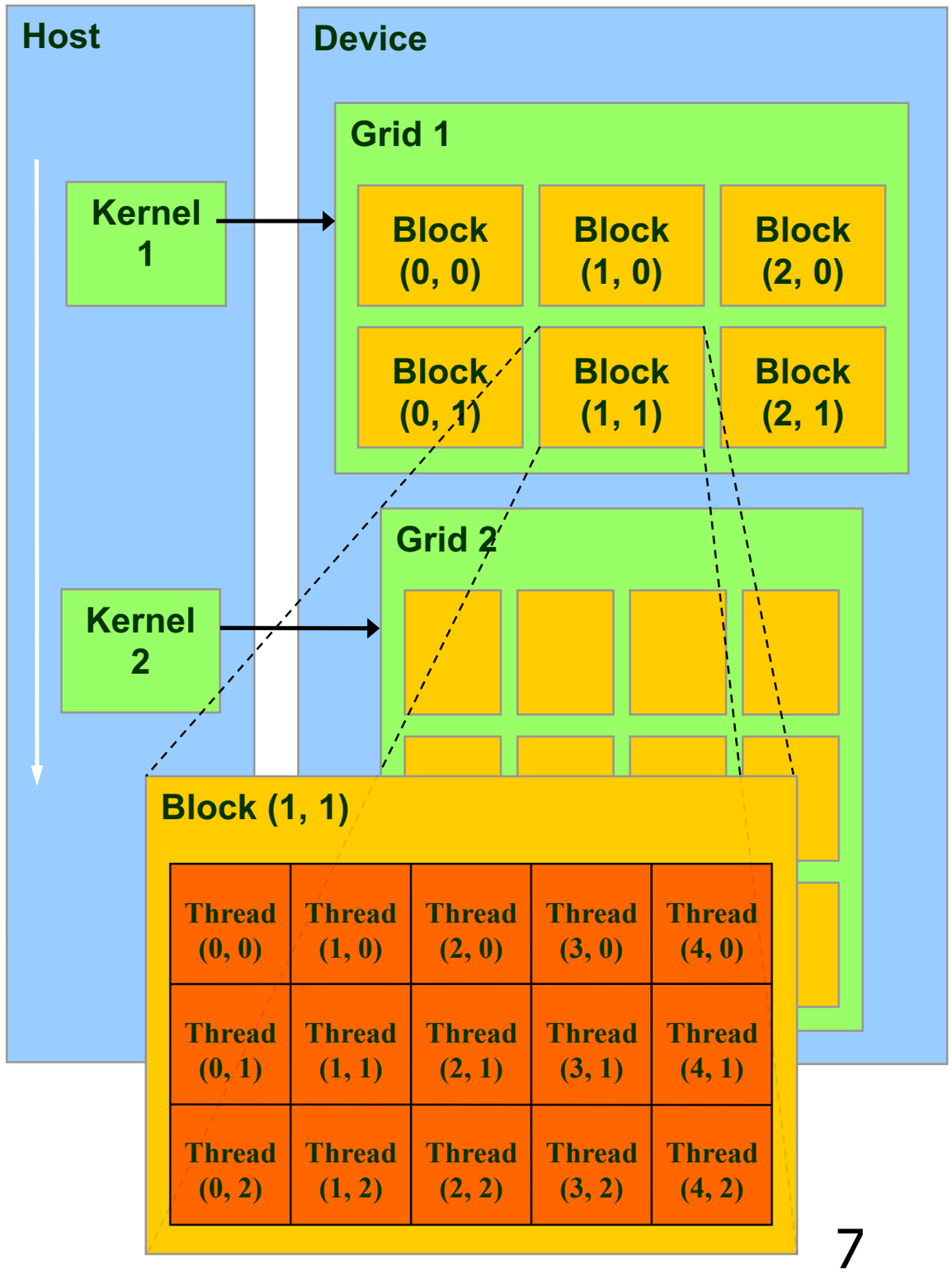
Thread Blocks are serially distributed to all the SMs
- Potentially >1 Thread Block per SM

Each SM launches Warps of Threads

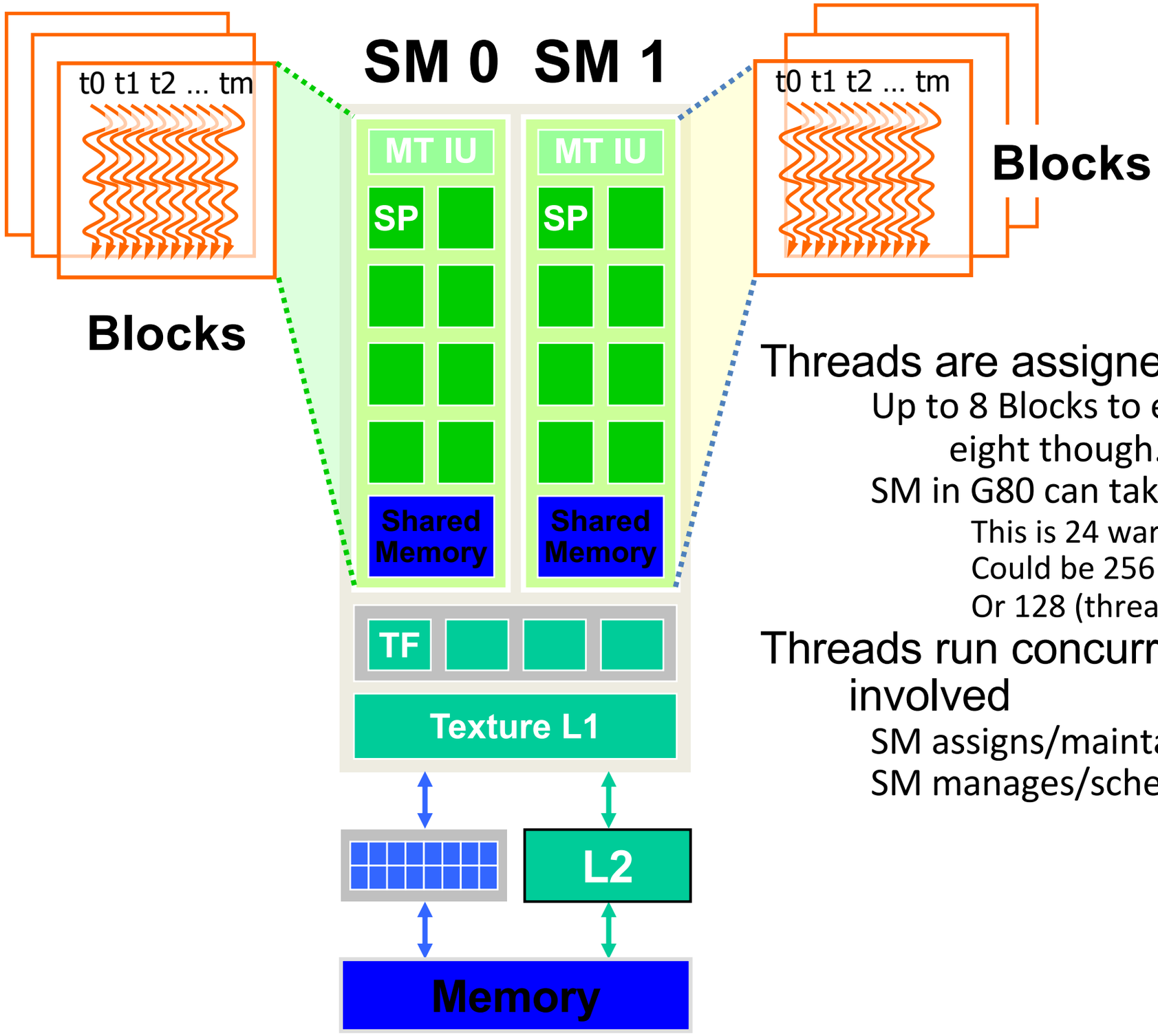
SM schedules and executes Warps that are ready to run

As Warps and Thread Blocks complete, resources are freed
- SPA can launch next Block[s] in line

NOTE: Two levels of scheduling:
- For running [desirably] a large number of blocks on a small number of SMs (16/14/etc.)
- For running up to 24 warps of threads on the 8 SPs available on each SM



SM Executes Blocks

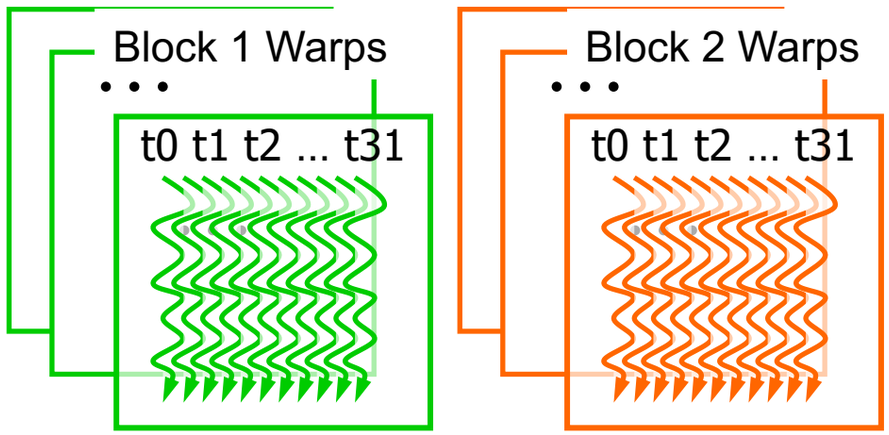


Threads are assigned to SMs in Block granularity
- Up to 8 Blocks to each SM (doesn't mean you'll have eight though…)
- SM in G80 can take up to 768 threads
  - This is 24 warps (occupancy calculator!!)
  - Could be 256 (threads/block) * 3 blocks
  - Or 128 (threads/block) * 6 blocks, etc.

Threads run concurrently but time slicing is involved
- SM assigns/maintains thread id #s
- SM manages/schedules thread execution

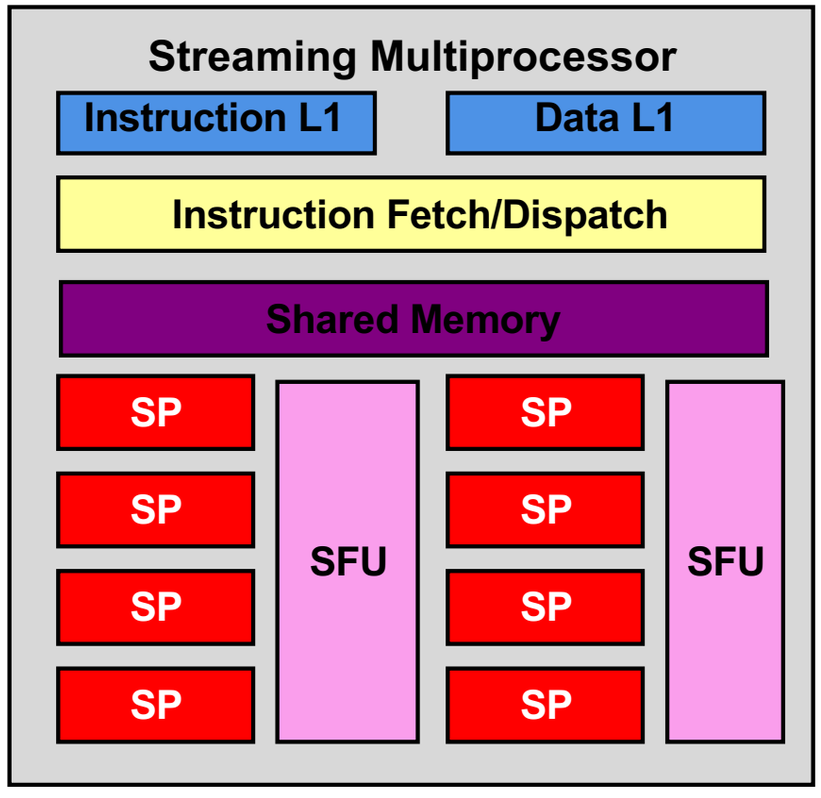
Thread Scheduling/Execution

- Each Thread Block is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model

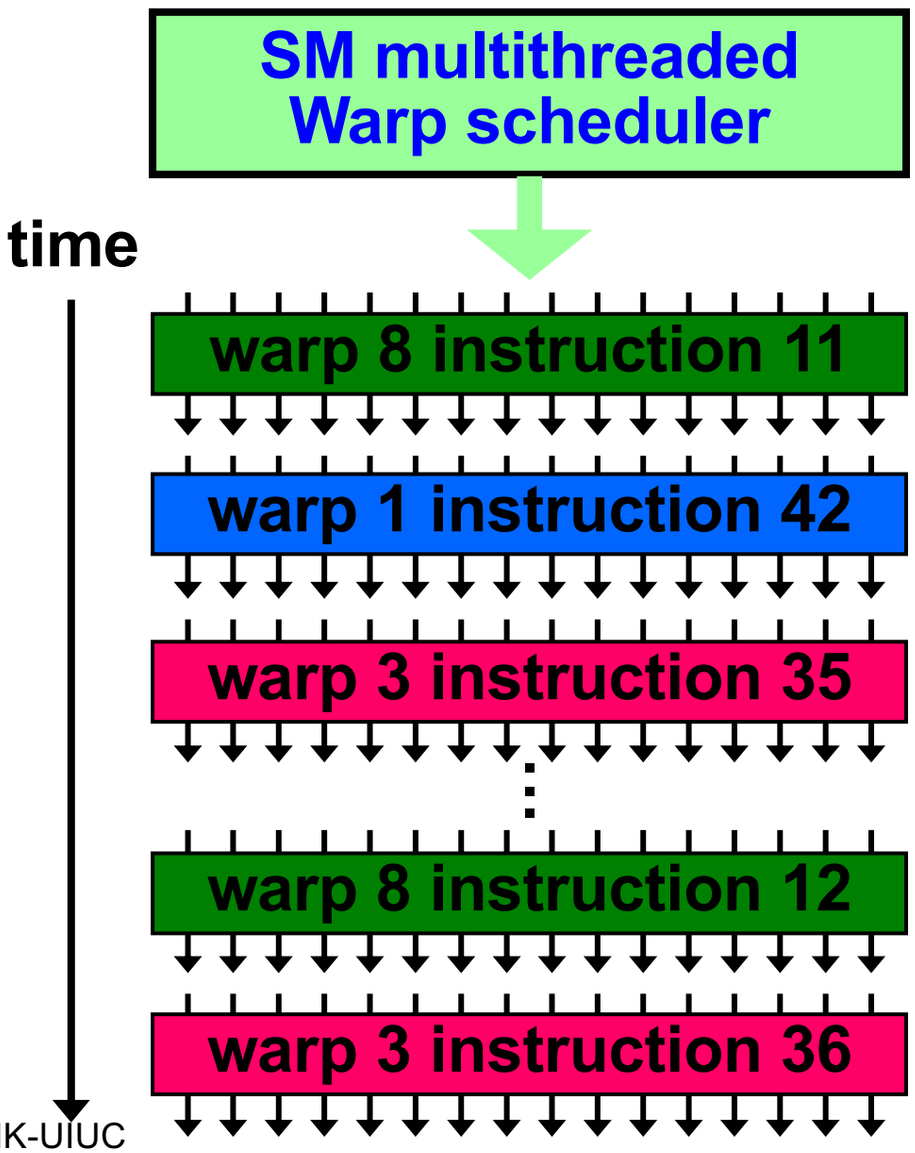


- Warps are the basic scheduling units in SM

- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only *one* of the 24 Warps will be selected for instruction fetch and execution.



SM Warp Scheduling



SM hardware implements zero-overhead Warp scheduling

- Warps whose next instruction has its operands ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy
- All threads in a Warp execute the same instruction when selected

4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80

Side-comment:

- Suppose your code has one global memory access every four instructions
- Then, a minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency

SM SIMT Multithreaded Execution

Single-Instruction Multi-Thread instruction scheduler

warp 8 instruction 11

warp 1 instruction 42

warp 3 instruction 95

warp 8 instruction 12

warp 3 instruction 96

- **Weaving: first parallel thread technology**
- **Warp: the set of 32 parallel threads that execute a SIMT instruction**
- **SIMT: Single-Instruction Multi-Thread**
- **SM hardware implements zero-overhead warp and thread scheduling**
- **Each SM executes up to 768 concurrent threads, as 24 SIMT warps of 32 threads**
- **Threads can execute independently**
- **SIMT warp diverges and converges when threads branch independently**
- **Best efficiency and performance when threads of a warp execute together**
- **SIMT across threads (not just SIMD data) provides easy single-thread scalar programming with SIMD efficiency**

Language Extensions: Built-in Variables

dim3 gridDim;

Dimensions of the grid in blocks (**gridDim.z** unused)

dim3 blockDim;

Dimensions of the block in threads

dim3 blockIdx;

Block index within the grid

dim3 threadIdx;

Thread index within the block

CUDA Thread Organization

- All threads in grid execute same kernel funtion
- They distinguish themselves with blockIdx and threadIdx
- gridDim and blockDim provide the dimensionality of grid and block
- Grid with N thread blocks—blockIdx: 0 to N-1
- Each block is m threads: blockIdx: 0 to M-1
- Each grid has a total of N*M threads
- Thread ID= blockIdx.x*blockDim.x + threadIdx.x
- Thread 3 of block 0 has thread ID= 0*M+3=3
- Thread 3 of block 5 has thread ID=5*M+3
- Grid is organized as a 2D array of blocks
- Block is organized as 3D array of threads

CUDA Thread Organization

- Organization of the grid is determined by execution configuation given by kernel launch
- First parameter in launch spcifies dimension of the grid int erms of number of threads
- Each such parameter is a dim3 type which is essentially a C struct with unsigned x,y,z
- But grids are 2D..third should be set to 1 for clarity
- Dim3 dimGrid(128, 1, 1);
- Dim3 dimBlock(32, 1,1);
- kernelFunction<<<dimGrid, dimBlock>>>(..);
- gridDim.x and gridDim.y range from 1 to 65535
- blockIdx.x ranges between 0 to gridDim.x-1
- blockIdx.y ranges from 0 to grid Dim.y-1
- Total threads in a block is limited to 512
- (512,1,1), (8,16,2), (16,16,2) allowed bu not (32,32,1)

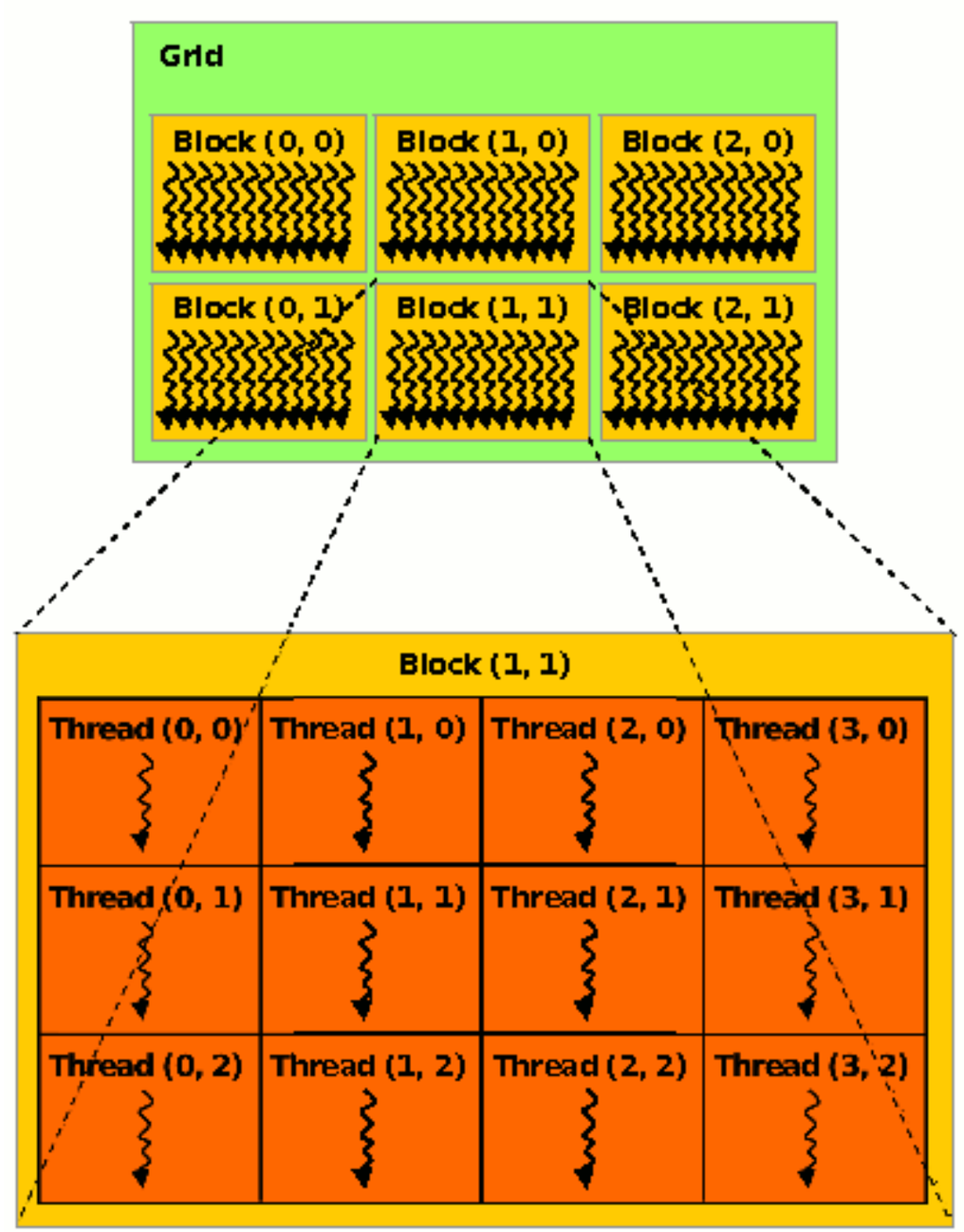
Hello World v.4.0: Vector Addition

```
#define N 256
#include <stdio.h>

__global void vecAdd (int *a, int *b, int *c);

int main() {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;

 // initialize a and b with real values (NOT SHOWN)

 size = N * sizeof(int);

 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);

 cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

 vecAdd<<<1,N>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);

 cudaFree(dev_a);
 cudaFree(dev_b);
 cudaFree(dev_c);

 exit (0);
}

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

- CUDA gives each thread a unique ThreadID to distinguish between each other even though the kernel instructions are the same.
- In our example, in the kernel call the memory arguments specify 1 block and N threads.

NVIDIA GPU Memory Hierarchy

- Grids map to GPUs
- Blocks map to the MultiProcessors (MP)
- Threads map to Stream Processors (SP)
- Warps are groups of (32) threads that execute simultaneously



NVIDIA GPU Memory Architecture

- In aNVIDIA GTX 480:
  - Maximum number of threads per block: 1024
  - Maximum sizes of x-, y-, and z- dimensions of thread block: 1024 x 1024 x 64
  - Maximum size of each dimension of grid of thread blocks: 65535 x 65535 x 65535

Defining Grid/Block Structure

- Need to provide each kernel call with values for two key structures:
  - Number of blocks in each dimension
  - Threads per block in each dimension

- myKernel<<< B,T >>>(arg1, …);

- B – a structure that defines the number of blocks in grid in each dimension (1D or 2D).
- T – a structure that defines the number of threads in a block in each dimension (1D, 2D, or 3D).

1D Grids and/or 1D Blocks

- If want a 1-D structure, can use a integer for B and T in:
- myKernel<<< B, T >>>(arg1, …);
- B – An integer would define a 1D grid of that size
- T – An integer would define a 1D block of that size

- Example: myKernel<<< 1, 100 >>>(arg1, ...);

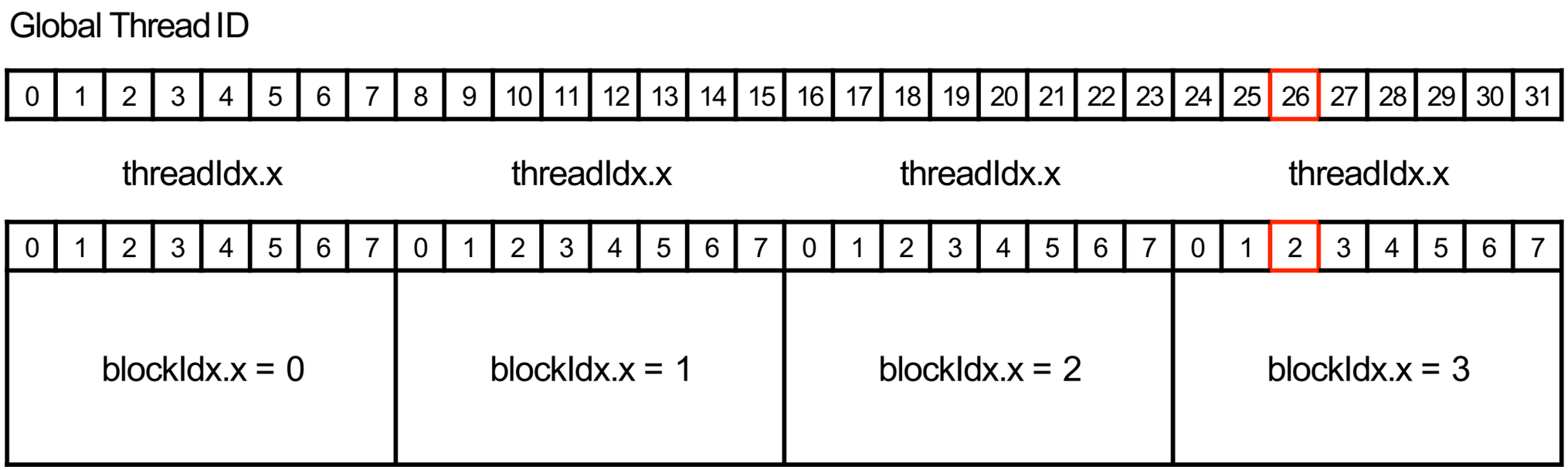
CUDA Built-In Variables

- **`blockIdx.x`**, **`blockIdx.y`**, **`blockIdx.z`** are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.
- **`threadIdx.x`**, **`threadIdx.y`**, **`threadIdx.z`** are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.
- **`blockDim.x`**, **`blockDim.y`**, **`blockDim.z`** are built-in variables that return the "block dimension" (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis).

- So, you can express your collection of blocks, and your collection of threads within a block, as a 1D array, a 2D array or a 3D array.
- These can be helpful when thinking of your data as 2D or 3D.

- The full global thread ID in x dimension can be computed by:
  - x = blockIdx.x * blockDim.x + threadIdx.x;

Thread Identification Example: x-direction



- Assume a hypothetical 1D grid and 1D block architecture: 4 blocks, each with 8 threads.

- For Global Thread ID 26:
  - gridDim.x = 4 x 1
  - blockDim.x = 8 x 1
  - Global Thread ID = blockIdx.x * blockDim.x + threadIdx.x
  - = 3 x 8 + 2 = 26

Vector Addition Revisited

```
#define N 1618
#define T 1024 // max threads per block
#include <stdio.h>

__global void vecAdd (int *a, int *b, int *c);

int main() {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;

 // initialize a and b with real values (NOT SHOWN)

 size = N * sizeof(int);


 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);

 cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

 vecAdd<<<(int)ceil(N/T),T>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);

 cudaFree(dev_a);
 cudaFree(dev_b);
 cudaFree(dev_c);

 exit (0);
}


__global void vecAdd (int *a, int *b, int *c) {
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 if (i < N) {
  c[i] = a[i] + b[i];
 }
}
```

- Since the maximum number of threads per dimension in a block is 1024, if you must use more than one block to access more threads.
- Divide the work between different blocks.
- Notice that each block is reserved completely; in this example, two blocks are reserved even though most of the second block is not utilized.
- WARNING: CUDA does not issue warnings or errors if your thread bookkeeping is incorrect -- Use small test cases to verify that everything is okay.

Higher Dimensional Grids/Blocks

- 1D grids/blocks are suitable for 1D data, but higher dimensional grids/blocks are necessary for:
  - higher dimensional data.
  - data set larger than the hardware dimensional limitations of blocks.

- CUDA has built-in variables and structures to define the number of blocks in a grid in each dimension and the number of threads in a block in each dimension.

CUDA Built-In Vector Types and Structures

- uint3 and dim3 are CUDA-defined structures of unsigned integers: x, y, and z.
  - struct uint3 {x; y; z;};
  - struct dim3 {x; y; z;};
- The unsigned structure components are automatically initialized to 1.
- These vector types are mostly used to define grid of blocks and threads.
- There are other CUDA vector types (discussed later).

CUDA Built-In Variables for Grid/Block Sizes

- dim3 gridDim -- Grid dimensions, x and y (z not used).
- Number of blocks in grid =
  gridDim.x * gridDim.y

- dim3 blockDim -- Size of block dimensions x, y, and z.
- Number of threads in a block =
  blockDim.x * blockDim.y * blockDim.z

Example Initializing Values

- To set dimensions:

```
dim3 grid(16,16);        // grid = 16 x 16 blocks
dim3 block(32,32);       // block = 32 x 32 threads
myKernel<<<grid, block>>>(...);
```

- which sets:

```
grid.x = 16;
grid.y = 16;
block.x = 32;
block.y = 32;
block.z = 1;
```

CUDA Built-In Variables for Grid/Block Indices

- uint3 blockIdx -- block index within grid:
  - blockIdx.x, blockIdx.y (z not used)
- uint3 threadIdx -- thread index within block:
  - threadIdx.x, threadIdx.y, threadIdx.z

- Full global thread ID in x and y dimensions can be computed by:
  - x = blockIdx.x * blockDim.x + threadIdx.x;
  - y = blockIdx.y * blockDim.y + threadIdx.y;