Lec 3 CUDA Software Abstraction

Tonghua Su
School of Software
Harbin Institute of Technology

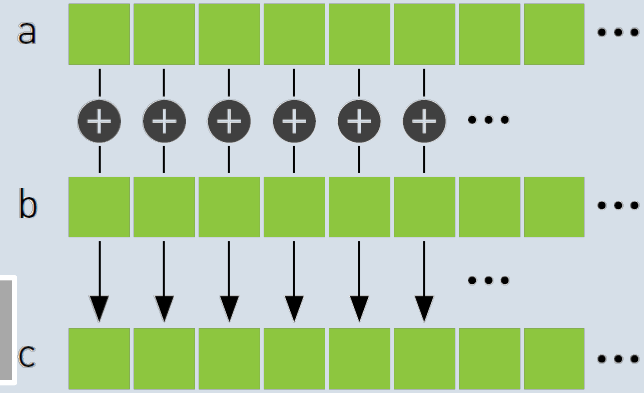
Outline

1. Review Lec1 & 2
2. Multithreading
3. CUDA Abstraction
4. Warp Scheduling
5. Lab 2
6. Software Layers in CUDA

Outline

1. **Review Lec1 & 2**
2. Multithreading
3. CUDA Abstraction
4. Warp Scheduling
5. Lab 2
6. Software Layers in CUDA

Review



Hello CUDA：Vector Sum

```
__global__ addKernel(int * const a, const int * const b, const int * const c)
{
        const unsigned int i = threadIdx.x;
        c[i] = a[i] + b[i];
}
```

线程ID，同时索引数据元素

```
void main(){
        ......
        int *dev_a,*dev_b,*dev_c;
        // Allocate GPU buffers for three vectors (two input, one output)    .
        cudaMalloc((void**)&dev_c, 128* sizeof(int));
        ......
        // Copy input vectors from host memory to GPU buffers.
        cudaMemcpy(dev_a, a, 128* sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, b, 128* sizeof(int), cudaMemcpyHostToDevice);

        // Launch a kernel on the GPU with one thread for each element.
        addKernel<<<1, 128>>>(dev_c, dev_a, dev_b);

        // Copy output vector from GPU buffer to host memory.
        cudaMemcpy(c, dev_c, 128* sizeof(int), cudaMemcpyDeviceToHost);

        cudaFree(dev_c);
        ......
}
```

分配显存

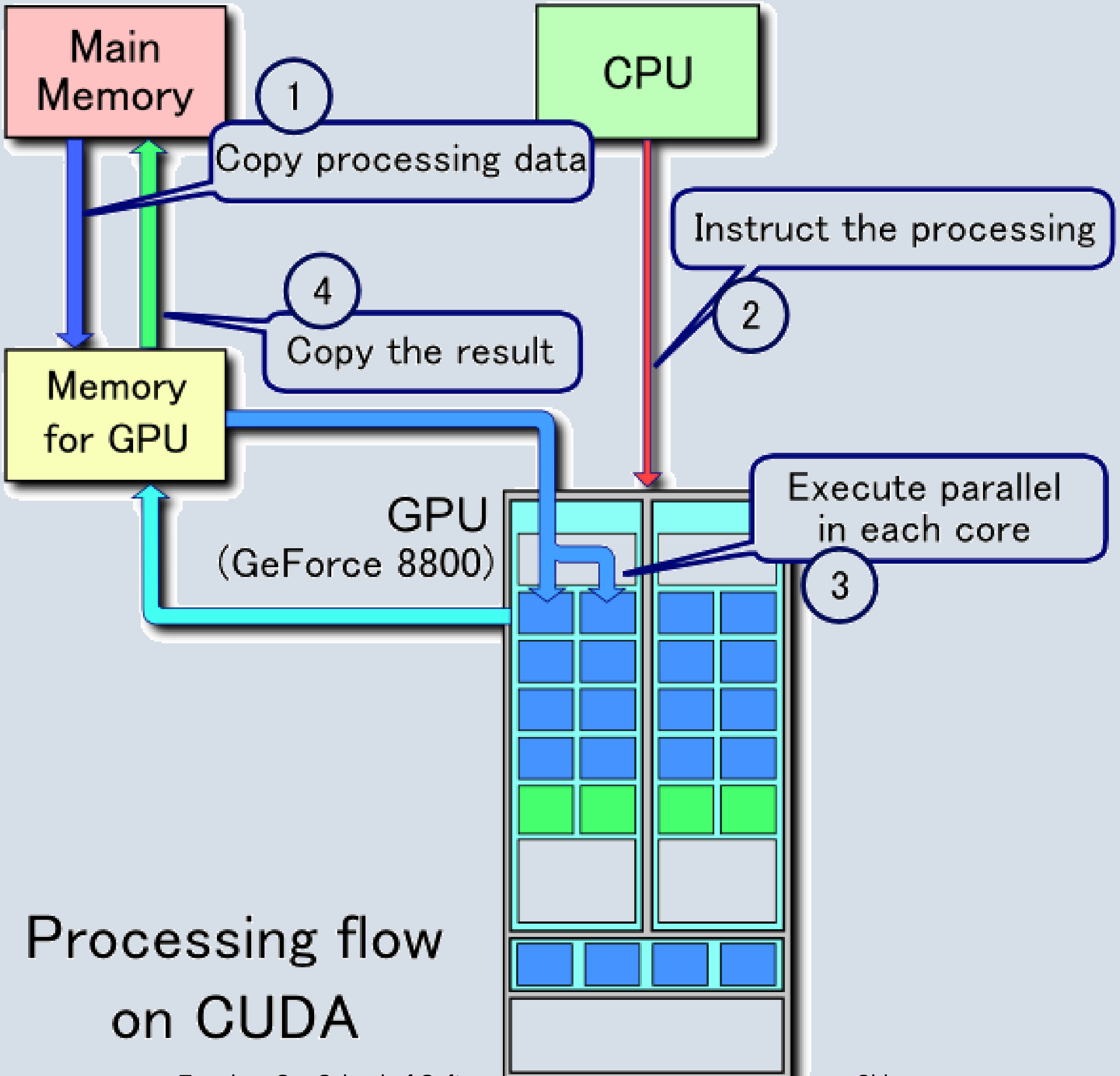
数据从主机复制到GPU

调用内核函数 addKernel

数据从GPU复制回主机

释放显存

Review


Main Memory
CPU
1
Copy processing data
Instruct the processing
2
4
Copy the result
Memory for GPU
GPU (GeForce 8800)
Execute parallel in each core
3
Processing flow on CUDA

Review

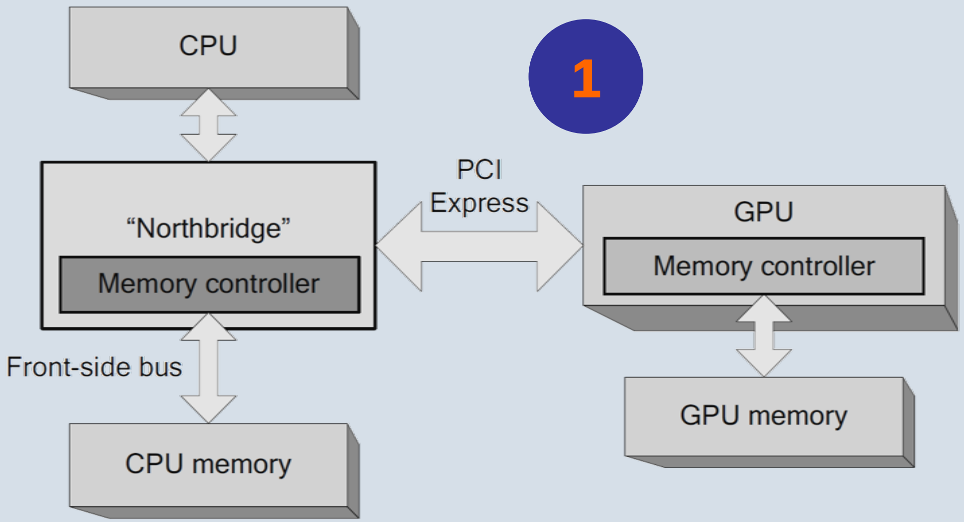
- Amdahl's Law

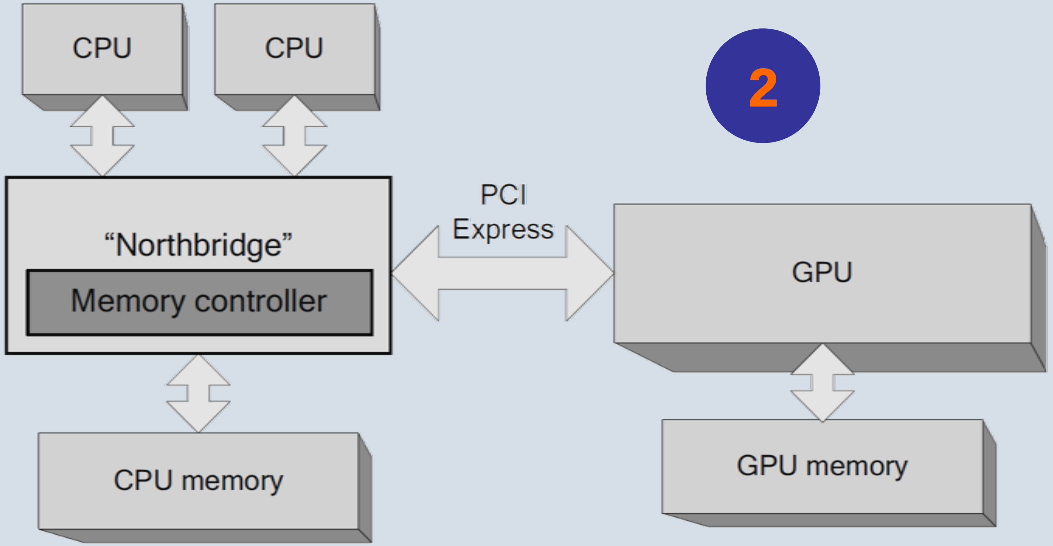
$$Speedup = \frac{1}{r_s + \frac{r_p}{N}}$$

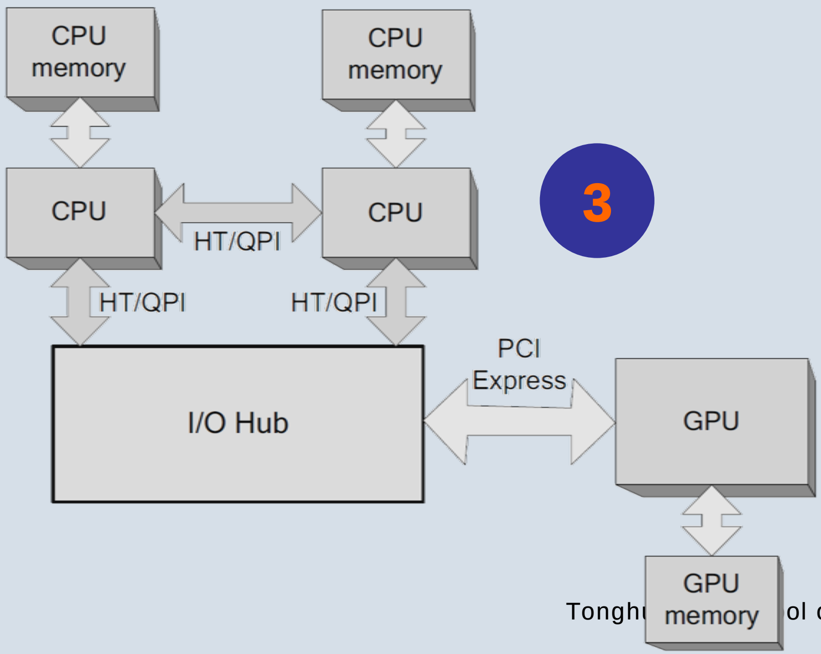
Amdahl, Gene (1967). "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". *AFIPS Conference Proceedings* (30): 483–485.

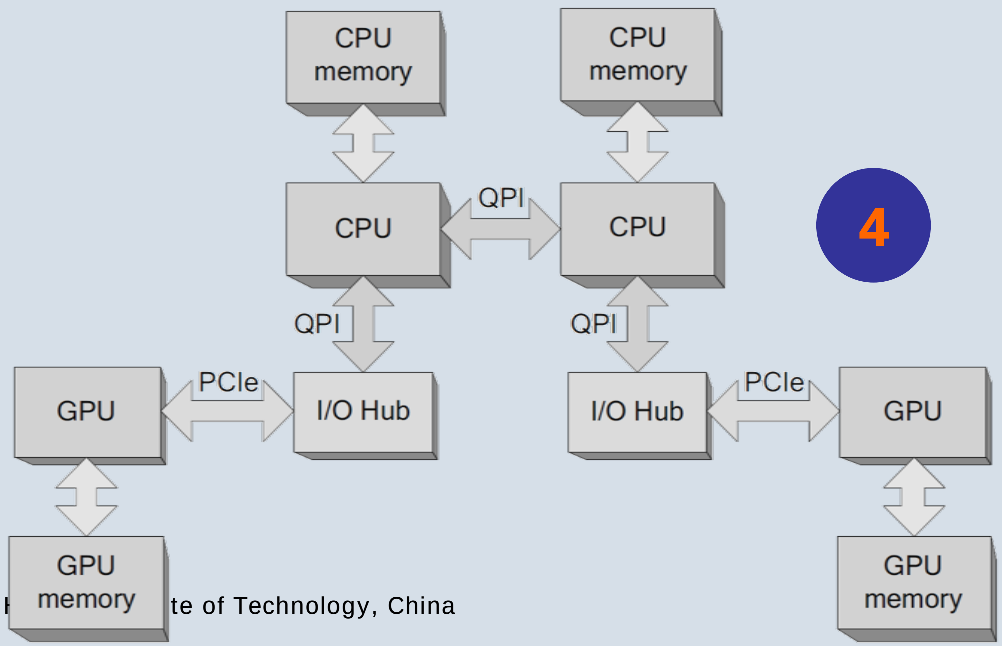
Review

- Single GPU Configuration



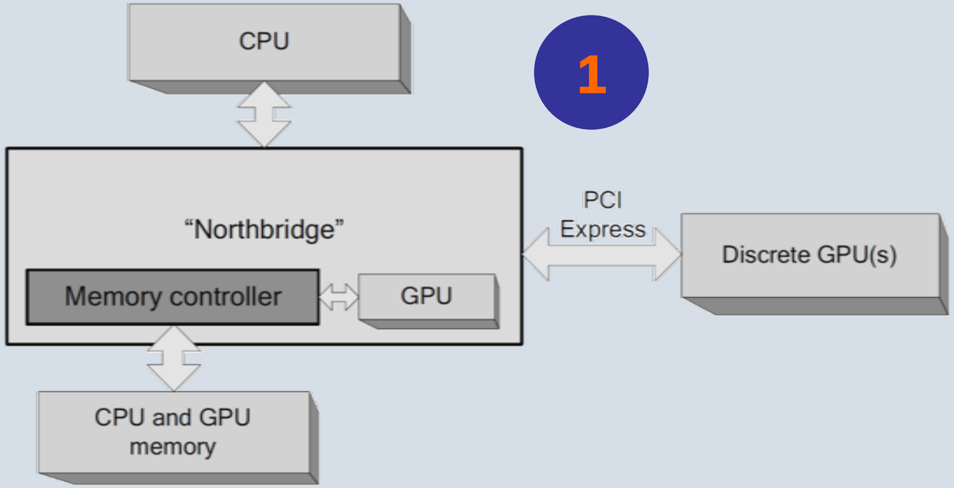


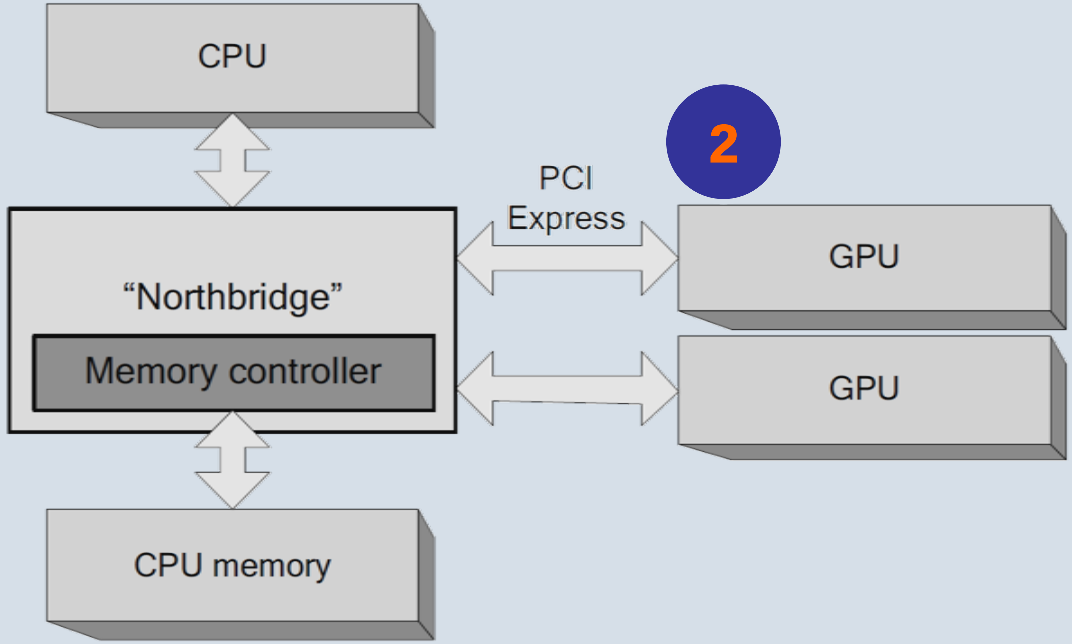


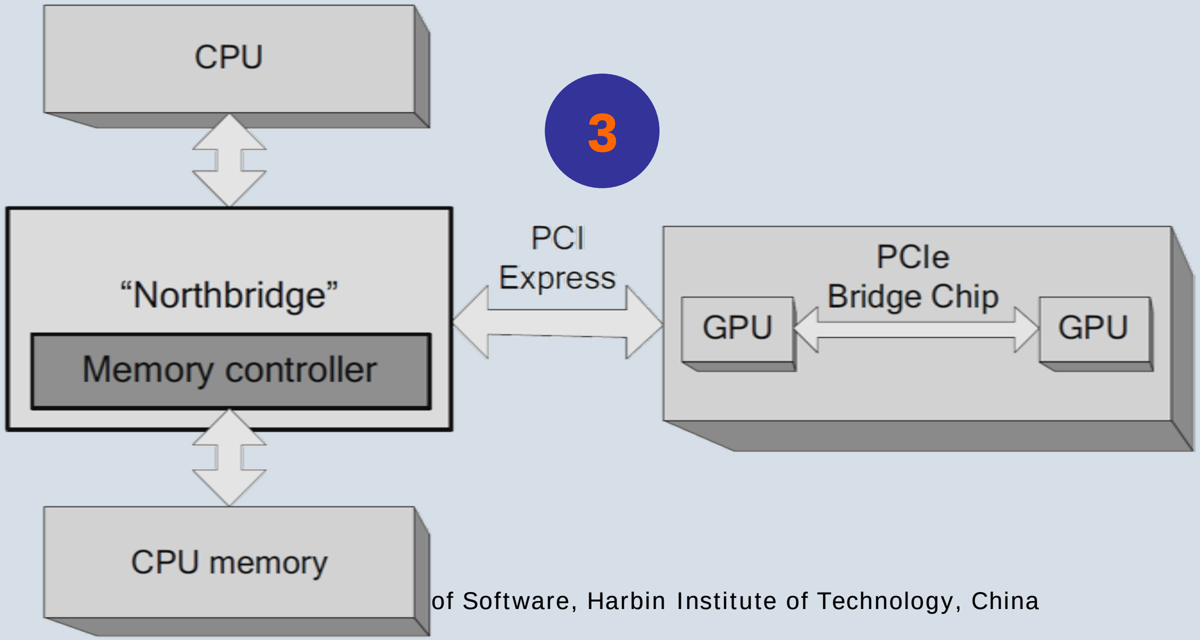


Review

- Multiple GPU Configuration

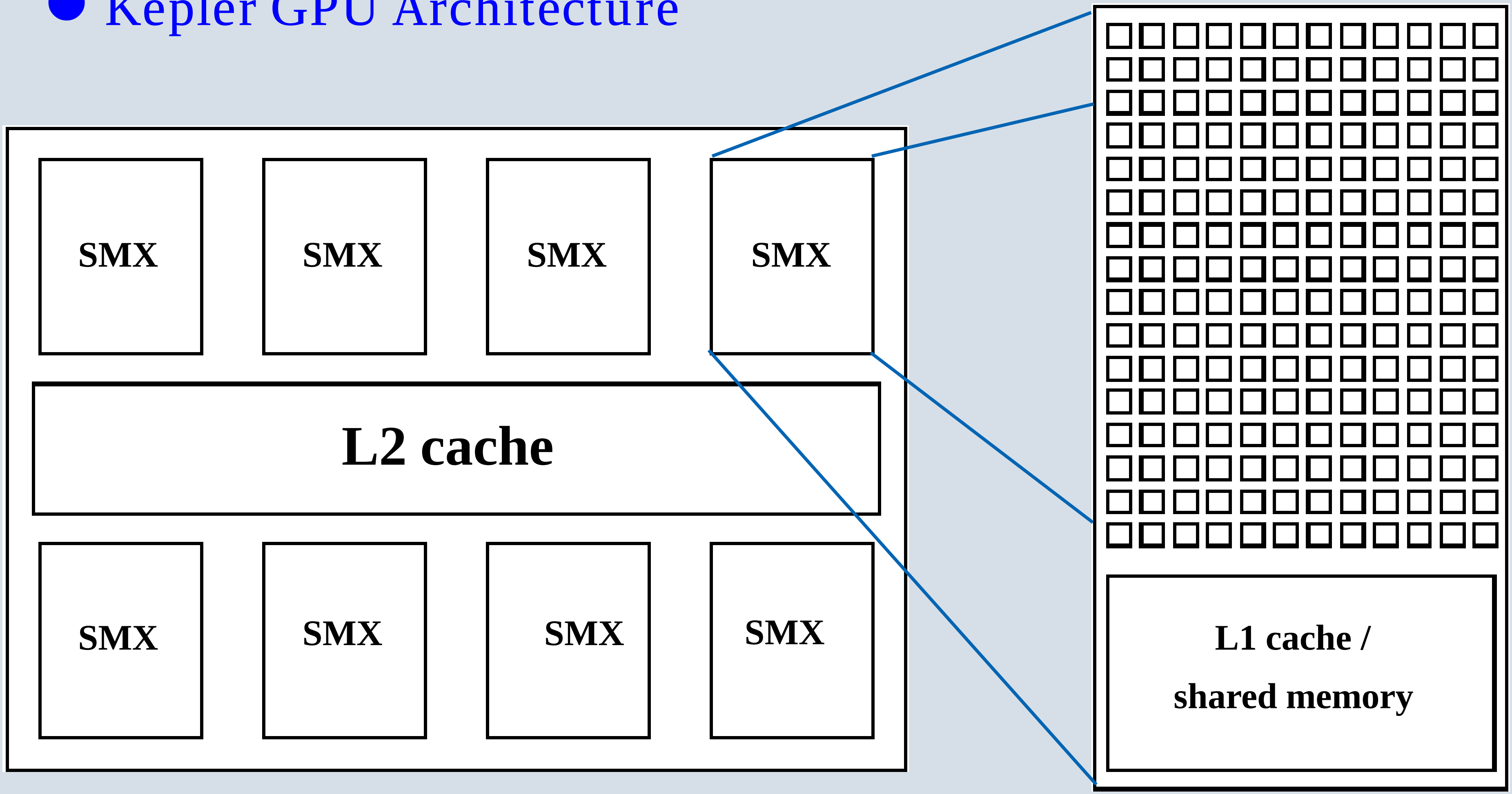






Review

- Kepler GPU Architecture

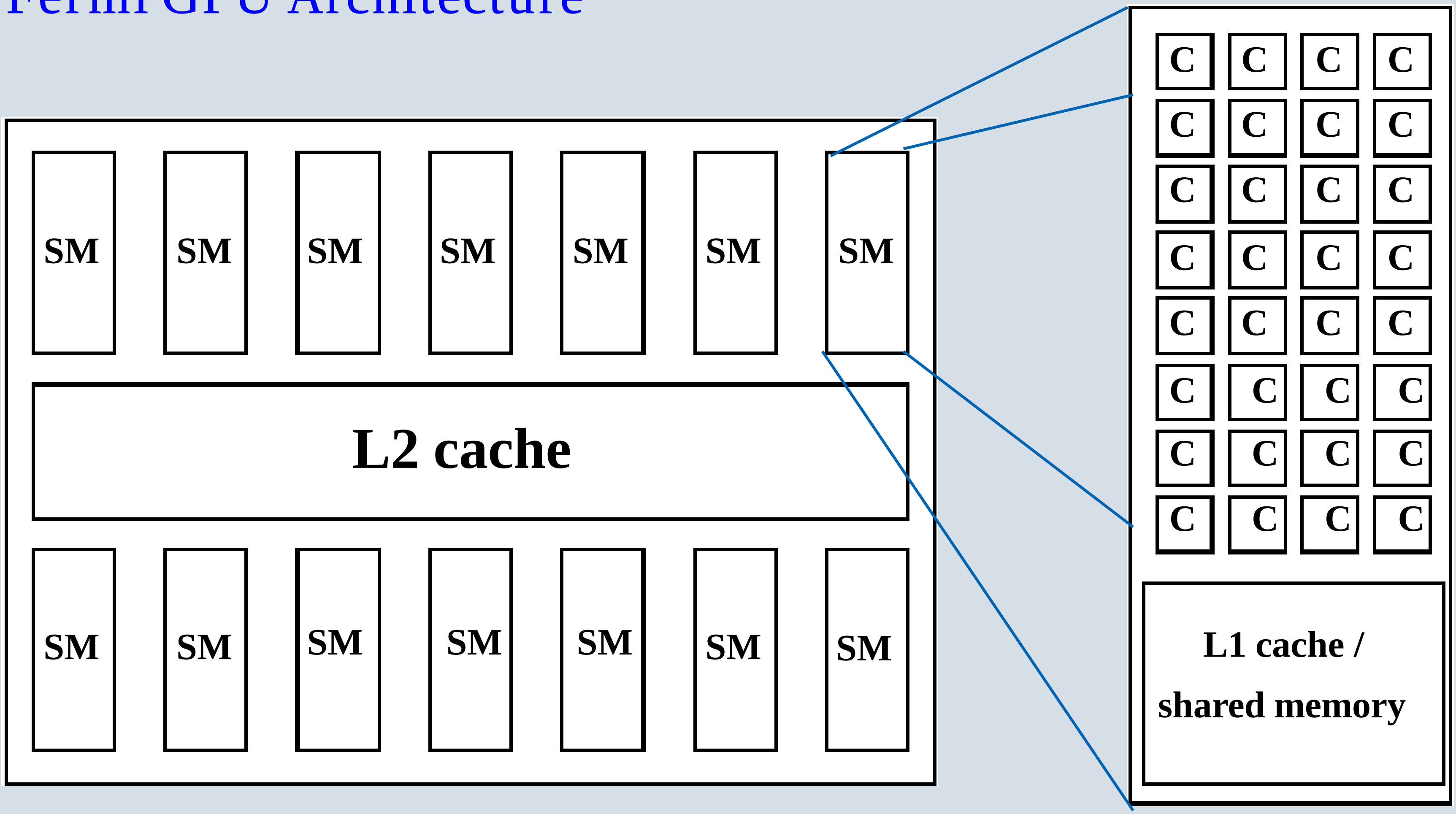


Review

- Kepler GPU Architecture
- building block is a “streaming multiprocessor” (SMX):
  - ✓ 192 cores and 64k registers
  - ✓ 64KB of shared memory / L1 cache
  - ✓ 8KB cache for constants
  - ✓ 48KB texture cache for read-only arrays
  - ✓ up to 2K threads per SMX

Review

- Fermi GPU Architecture



Review

- Fermi GPU Architecture
- older Fermi GPU has SM “streaming multiprocessor”:
  - ✓ 32 cores and 32k registers
  - ✓ 64KB of shared memory / L1 cache
  - ✓ 8KB cache for constants
  - ✓ up to 1536 threads per SM

Outline

Multithreading

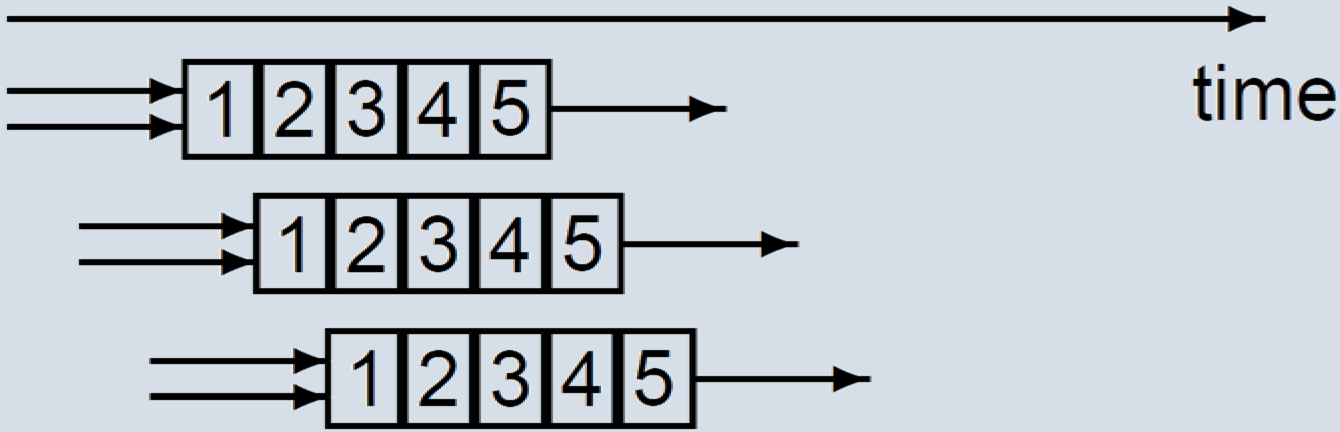
- Key hardware feature is that the cores in an SMX are SIMT (Single Instruction Multiple Threads) cores:
  - ✓ all cores execute the same instructions simultaneously, but with different data
  - ✓ similar to vector computing on CRAY supercomputers
  - ✓ minimum of 32 threads all doing the same thing at (almost) the same time
  - ✓ natural for graphics processing and much scientific computing
  - ✓ SIMT is also a natural choice for many-core chips to simplify each core

Multithreading

- Lots of active threads is the key to high performance:
  - ✓ no “context switching”: each thread has its own registers (which limits the number of active threads)
  - ✓ threads on each SMX execute in groups of 32 called “warps” – execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data

Multithreading

- for each thread, one operation completes long before the next starts – avoids the complexity of pipeline overlaps which can limit the performance of modern processors



- memory access from device memory has a delay of 400-600 cycles; with 40 threads this is equivalent to 10-15 operations, so hopefully there's enough computation to hide the latency

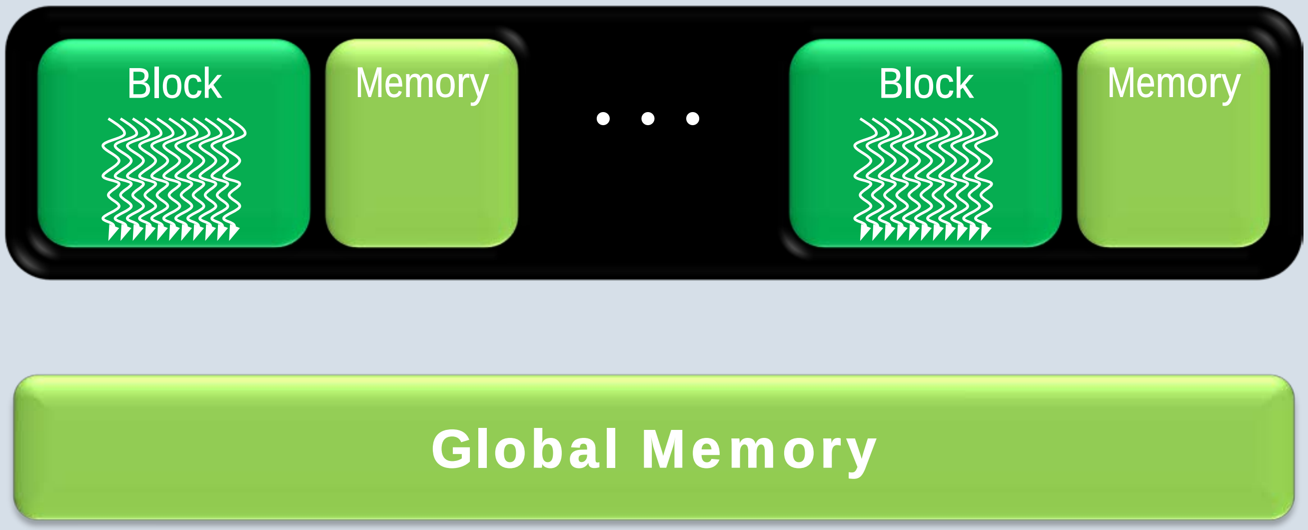
Outline

CUDA

- CUDA (Compute Unified Device Architecture) is NVIDIA's program development environment:
  - ✓ based on C with some extensions
  - ✓ extensive C++ support
  - ✓ FORTRAN support provided by PGI compiles lots of example code and good documentation
  - ✓ 2-4 week learning curve for those with experience of OpenMP and MPI programming
  - ✓ large user community on NVIDIA forums

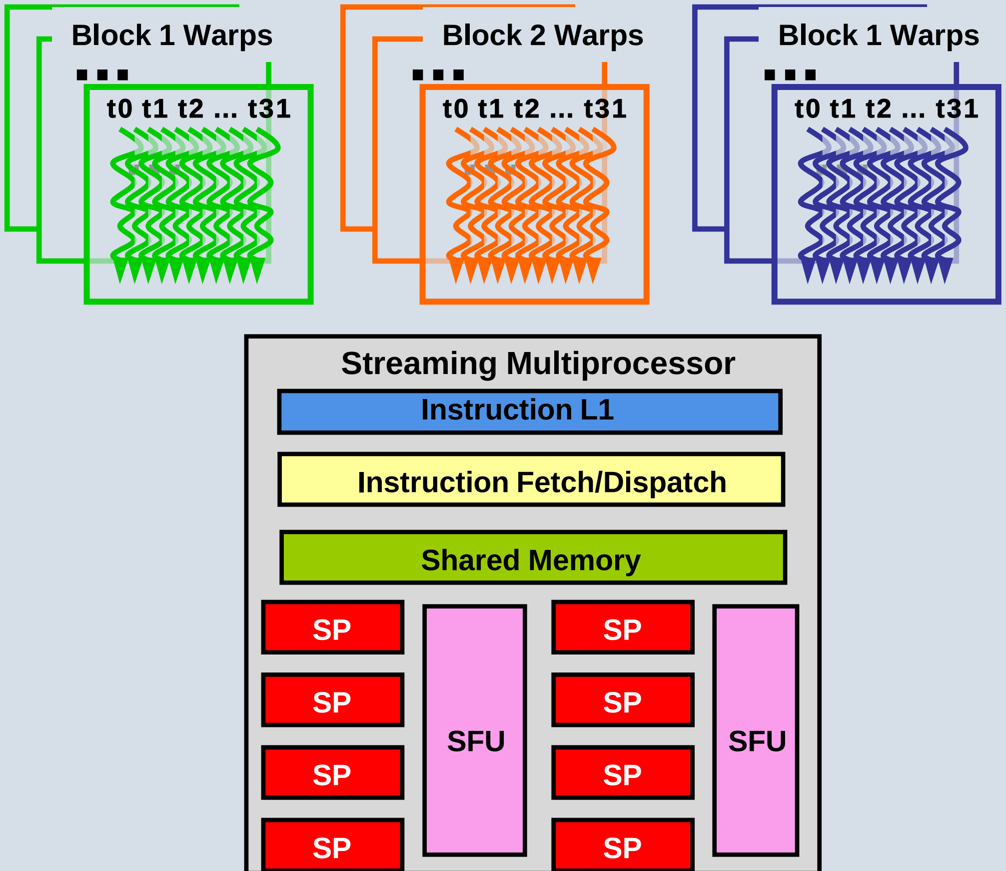
CUDA Abstraction

- CUDA virtualizes the physical hardware
  - ✓ thread is a virtualized scalar processor (registers, PC, state)
  - ✓ block is a virtualized multiprocessor (threads, shared mem.)
- Scheduled onto physical hardware without pre-emption
  - ✓ threads/blocks launch & run to completion/suspension
  - ✓ blocks should be independent



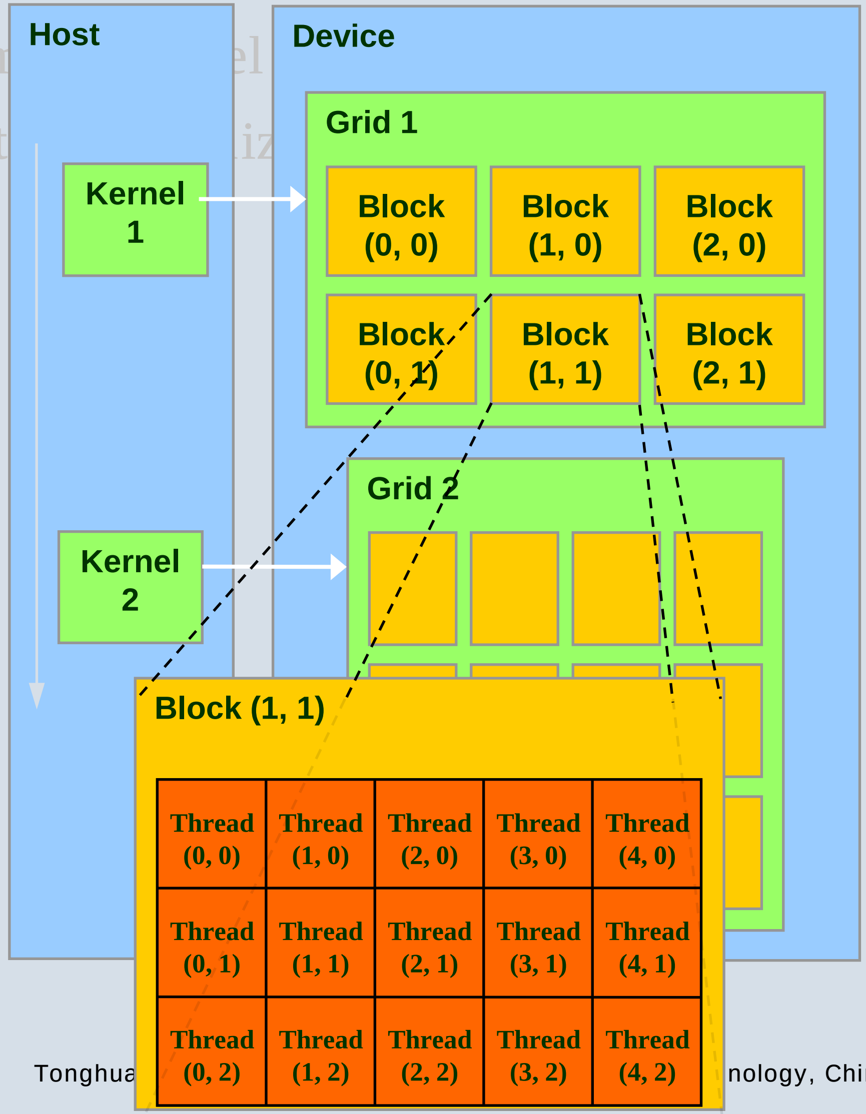
CUDA Abstraction

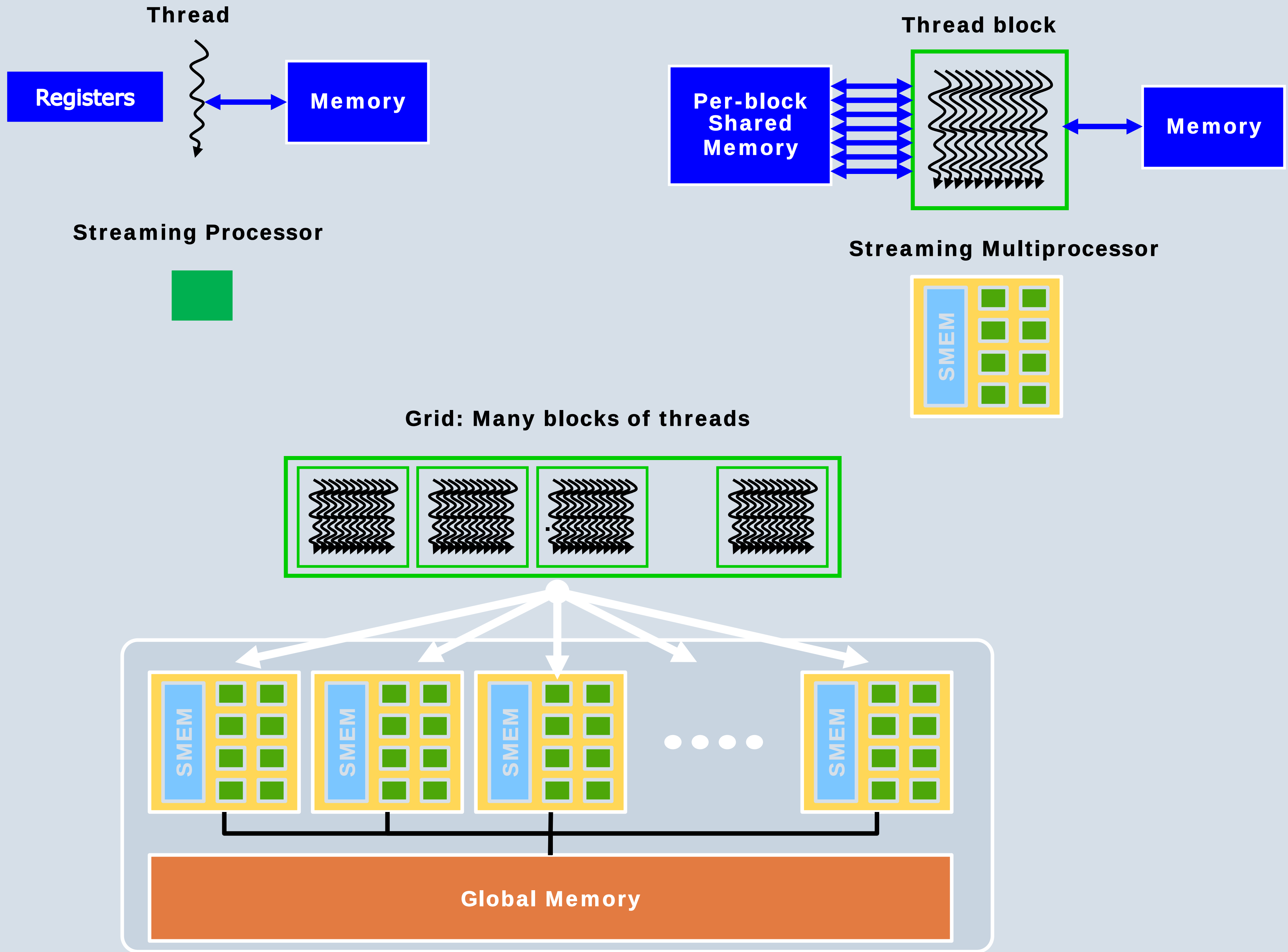
- Key Parallel Abstractions in CUDA
  - ✓ Hierarchy of concurrent threads
  - ✓ Shared memory model for cooperating threads
  - ✓ Lightweight synchronization primitives



CUDA Abstraction

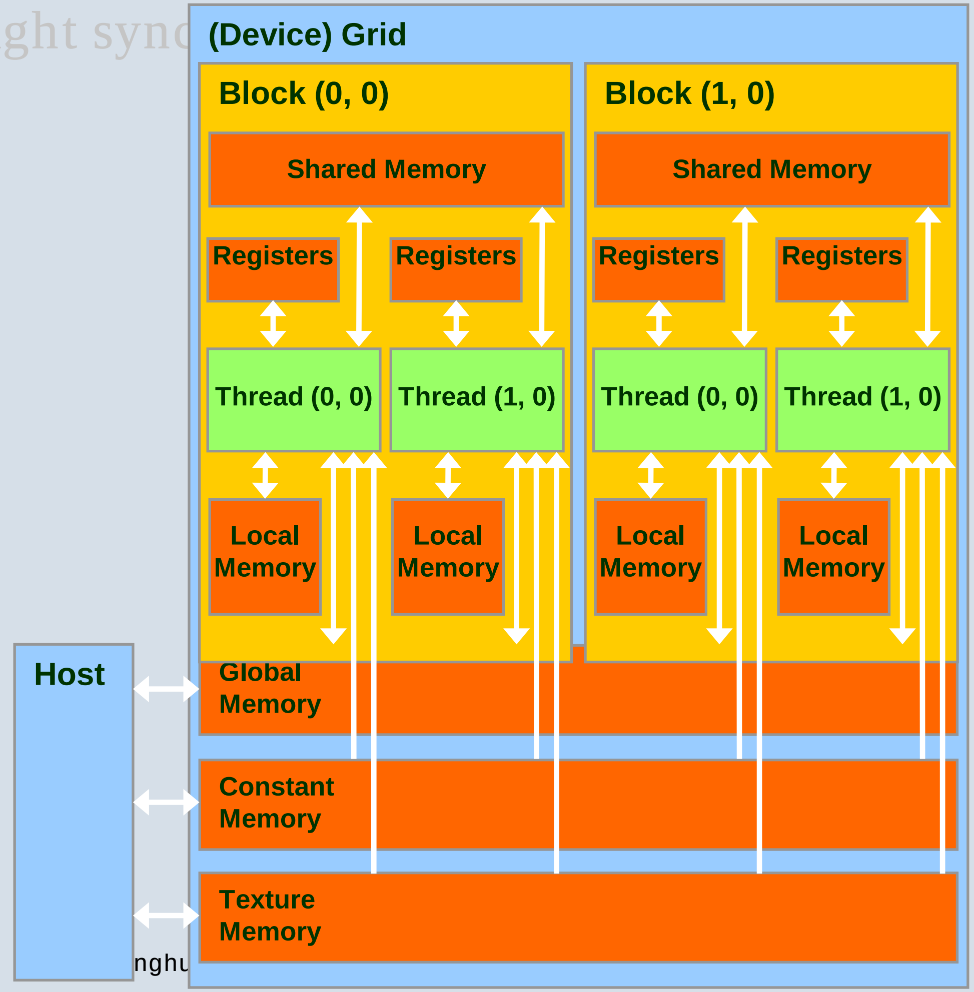
- Key Parallel Abstractions in CUDA
  - ✓ Hierarchy of concurrent threads
  - ✓ Shared mem... el
  - ✓ Lightweight ... iz




Thread
Registers
Memory
Thread block
Per-block
Shared
Memory
Memory
Streaming Processor
Streaming Multiprocessor
SMEM
Grid: Many blocks of threads
SMEM
SMEM
SMEM
SMEM
Global Memory

CUDA Abstraction

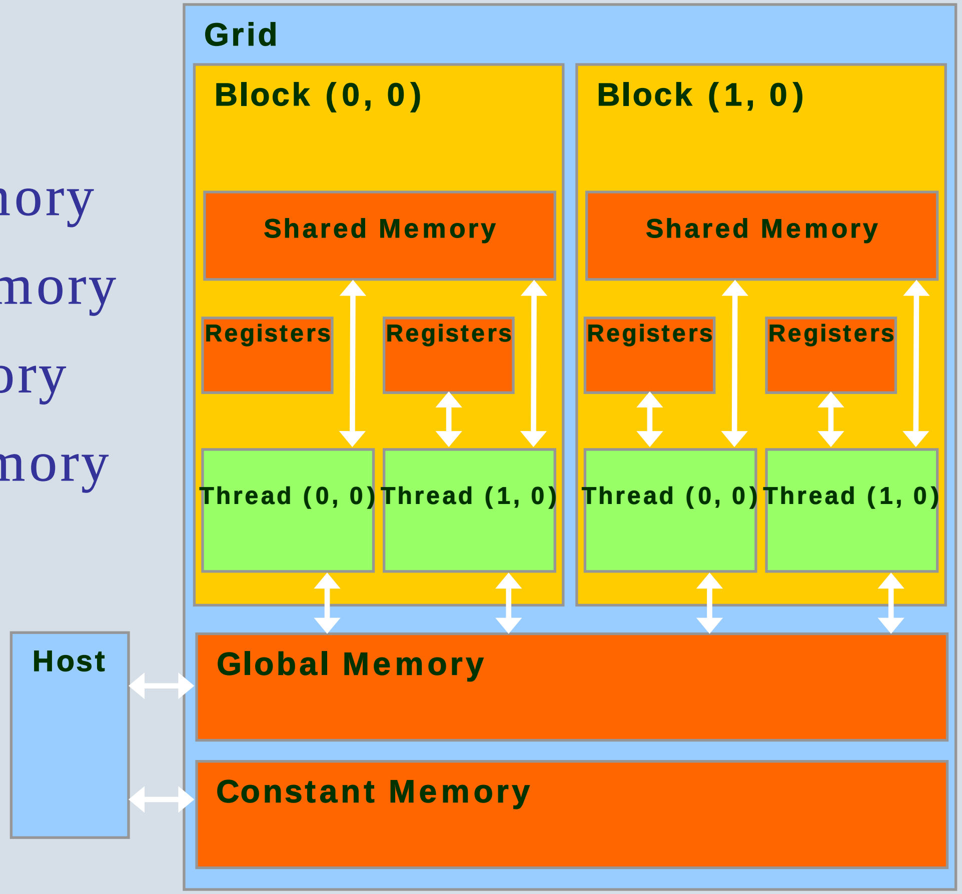
- Key Parallel Abstractions in CUDA
  - ✓ Hierarchy of concurrent threads
  - ✓ Shared memory model for cooperating threads
  - ✓ Lightweight sync



nghu

CUDA Abstraction

- Each thread can:
  - ✓ Read/write per-thread registers
  - ✓ Read/write per-thread local memory
  - ✓ Read/write per-block shared memory
  - ✓ Read/write per-grid global memory
  - ✓ Read/only per-grid constant memory



CUDA Abstraction

- Key Parallel Abstractions in CUDA
  - ✓ Hierarchy of concurrent threads
  - ✓ Shared memory model for cooperating threads
  - ✓ Lightweight synchronization primitives

CUDA Abstraction

- Global Synchronization
  - ✓ Finish a kernel and start a new one
  - ✓ All writes from all threads complete before a kernel finishes

```
step1<<<grid1,blk1>>>(...);
// The system ensures that all writes from step1 complete.
step2<<<grid2,blk2>>>(...);
```

  - ✓ Would need to decompose kernels into before and after parts

CUDA Abstraction

- Threads Synchronization
  - ✓ To ensure the threads visit the shared memory in order
  - ✓ __syncthreads()

```
__global__ void adj_diff(int *result, int *input)
{
  int tx = threadIdx.x;
  // allocate a __shared__ array, one element per thread
  __shared__ int s_data[BLOCK_SIZE];
  // each thread reads one element to s_data
  unsigned int i = blockDim.x * blockIdx.x + tx;
  s_data[tx] = input[i];

  // avoid race condition: ensure all loads complete before continuing
  __syncthreads();
  if(tx > 0)
    result[i] = s_data[tx] – s_data[tx–1];
  else if(i > 0)
  {
    // handle thread block boundary
    result[i] = s_data[tx] – input[i-1];
  }
}
```

CUDA Abstraction

- Race Conditions
  - ✓ What is the value of a in thread 0?
  - ✓ What is the value of a in thread 127?

```
threadId:0                         threadId:127
// vector[0] was equal to 0
vector[0] += 5;                     vector[0] += 1;
...                                  ...
a = vector[0];                     a = vector[0];
```

- ✓ CUDA provides atomic operations to deal with this problem

CUDA Abstraction

- Atomics
  - ✓ An atomic operation guarantees that only a single thread has access to a piece of memory while an operation completes
  - ✓ Different types of atomic instructions:
    - `atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`
  - ✓ Atomics are slower than normal load/store
  - ✓ You can have the whole machine queuing on a single location in memory
  - ✓ More types in Fermi
  - ✓ Atomics unavailable on G80!

CUDA Abstraction

- Atomics

```
// Determine frequency of colors in a picture
// colors have already been converted into ints
// Each thread looks at one pixel and increments
// a counter atomically
__global__ void histogram(int* color,
                          int* buckets)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int c = colors[i];
  atomicAdd(&buckets[c], 1);
}
```

Outline

1. Review Lec1 & 2
2. Multithreading
3. CUDA Abstraction
4. **Warp Scheduling**
5. Lab 2
6. Software Layers in CUDA

Kernel Function

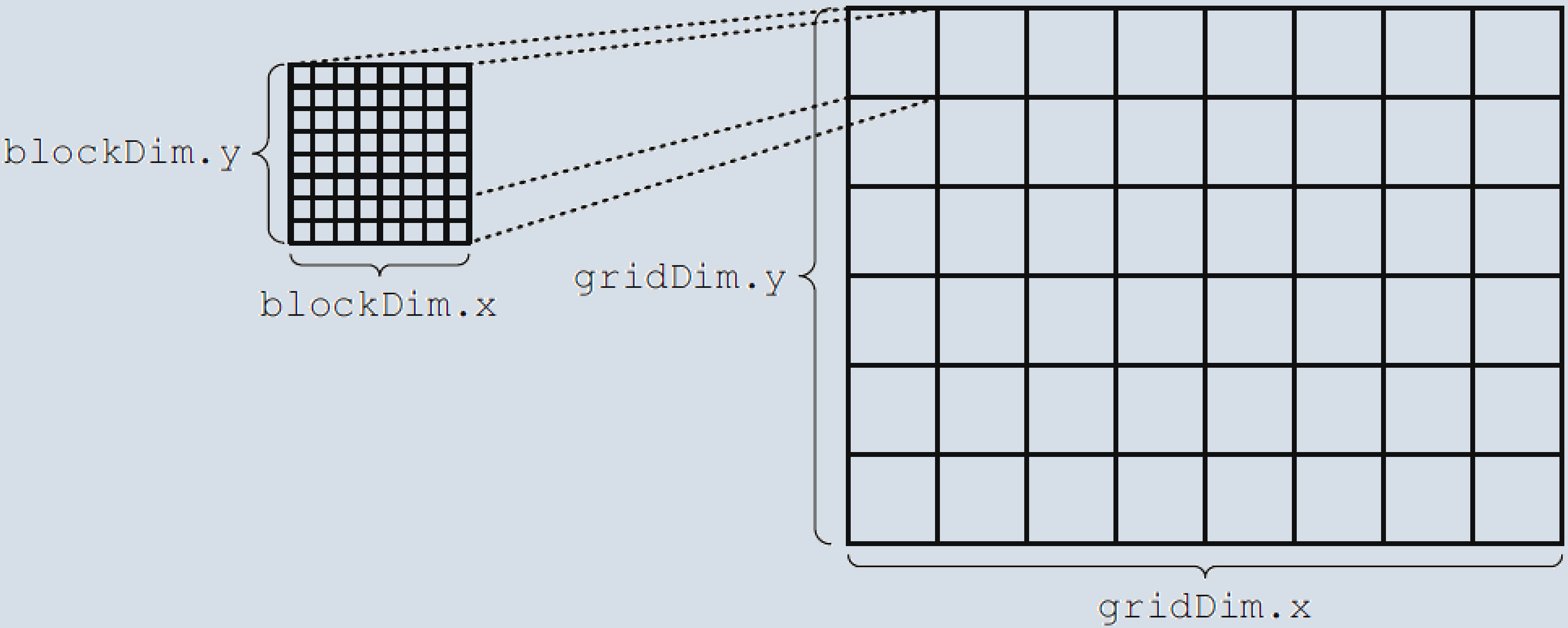
- In its simplest form it looks like:

kernel_routine<<<gridDim, blockDim>>>(args);

- ✓ gridDim is the number of instances of the kernel (the "grid" size)
- ✓ blockDim is the number of threads within each instance (the "block" size)
- ✓ args is a limited number of arguments, usually mainly pointers to arrays in graphics memory, and some constants which get copied by value
- ✓ The more general form allows gridDim and blockDim to be 2D or 3D to simplify application programs

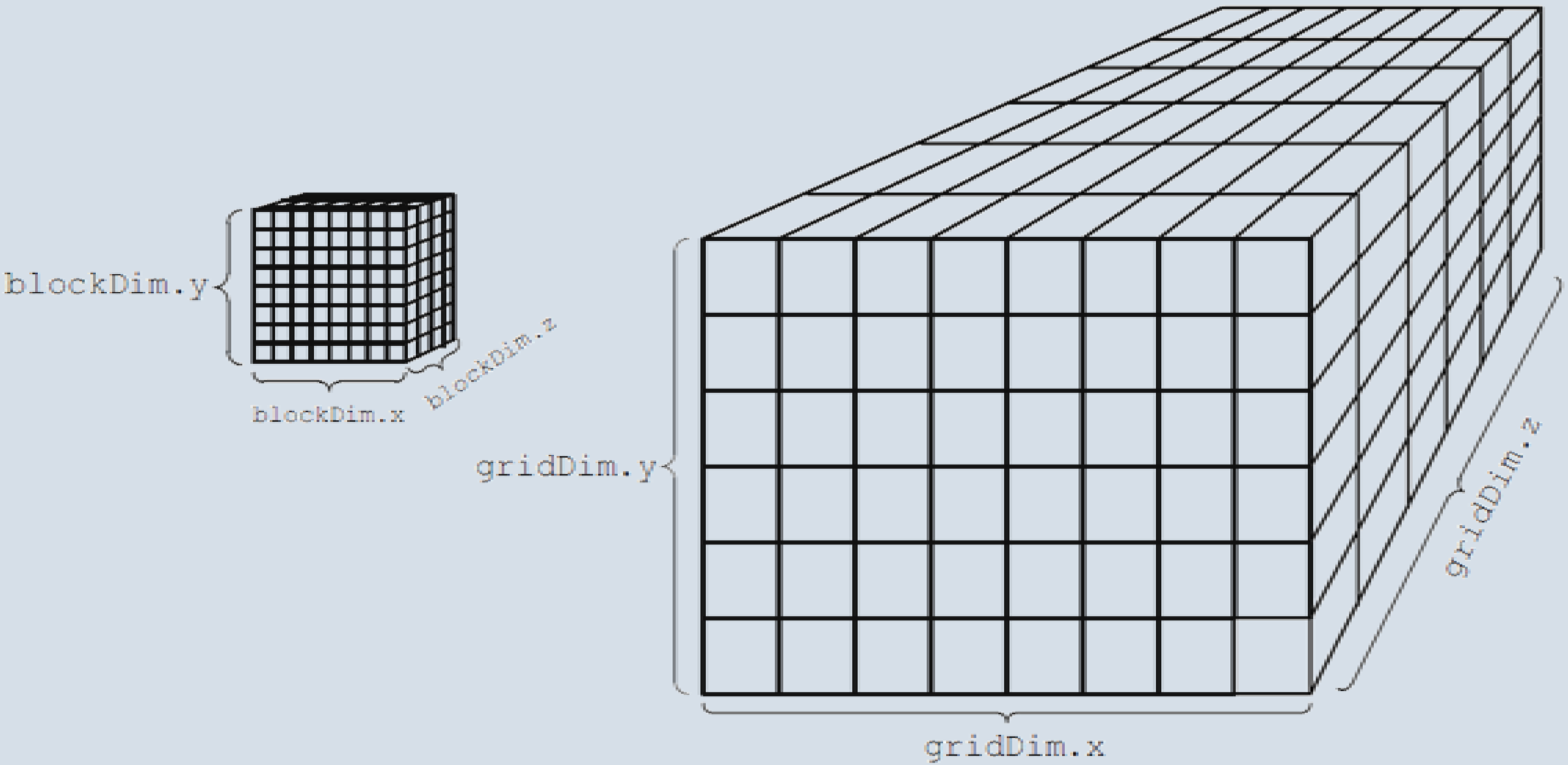
Kernel Function

- 2D block and 2D grid



Kernel Function

- 3D block and 3D grid



Kernel Function

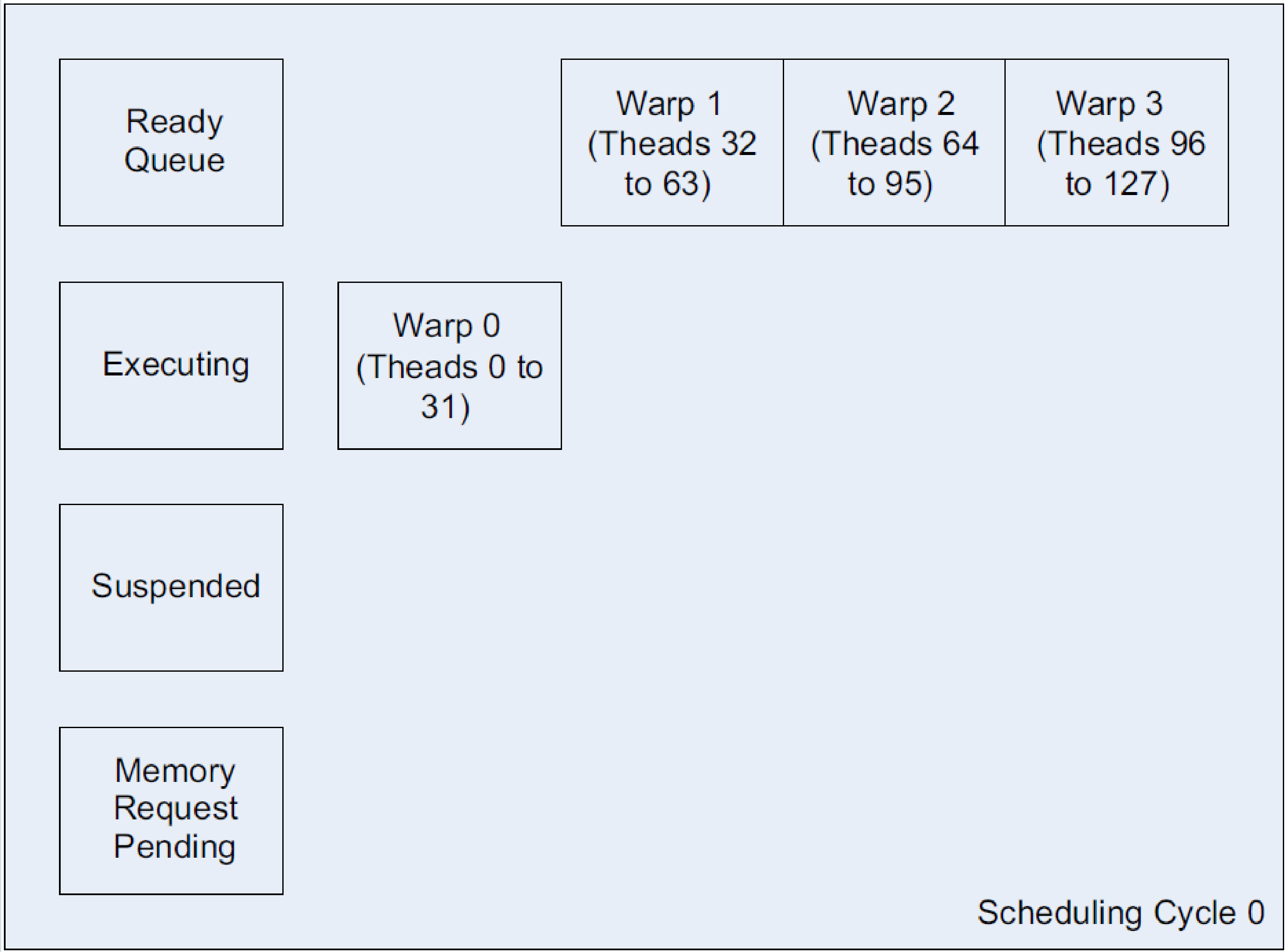
- How to calculate global block ID and thread ID?

CUDA Scheduling

- At a lower level, within the GPU:
  - ✓ each block of the execution kernel executes on an SMX
  - ✓ if the number of blocks exceeds the number of SMXs, then more than one will run at a time on each SMX if there are enough registers and shared memory, and the others will wait in a queue and execute later
  - ✓ all threads within one block can access local shared memory but can't see what the other block are doing (even if they are on the same SMX)
  - ✓ there are no guarantees on the order in which the blocks execute

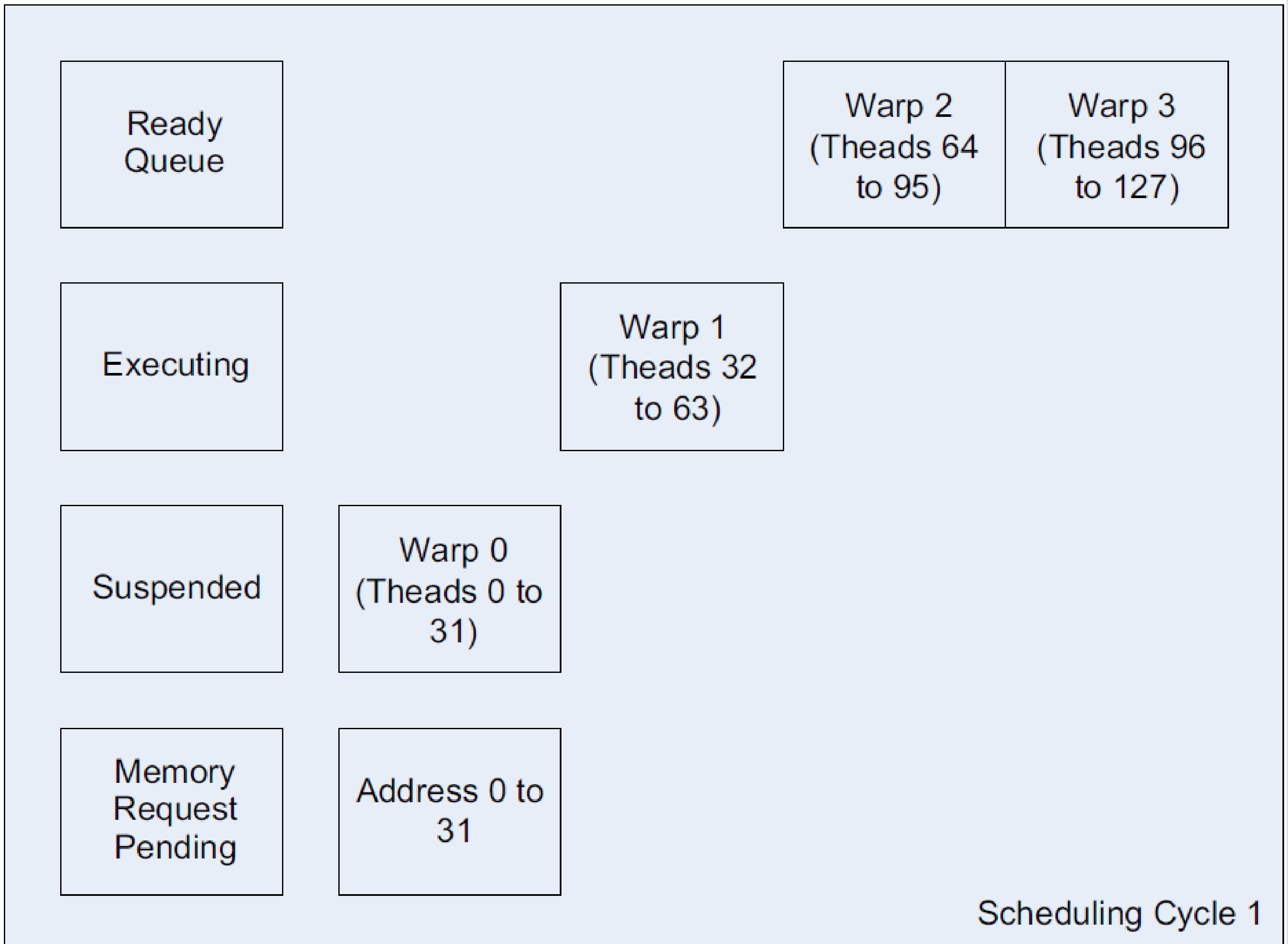
CUDA Scheduling

- Block Scheduling
  - ✓ Execute in warps of 32 threads



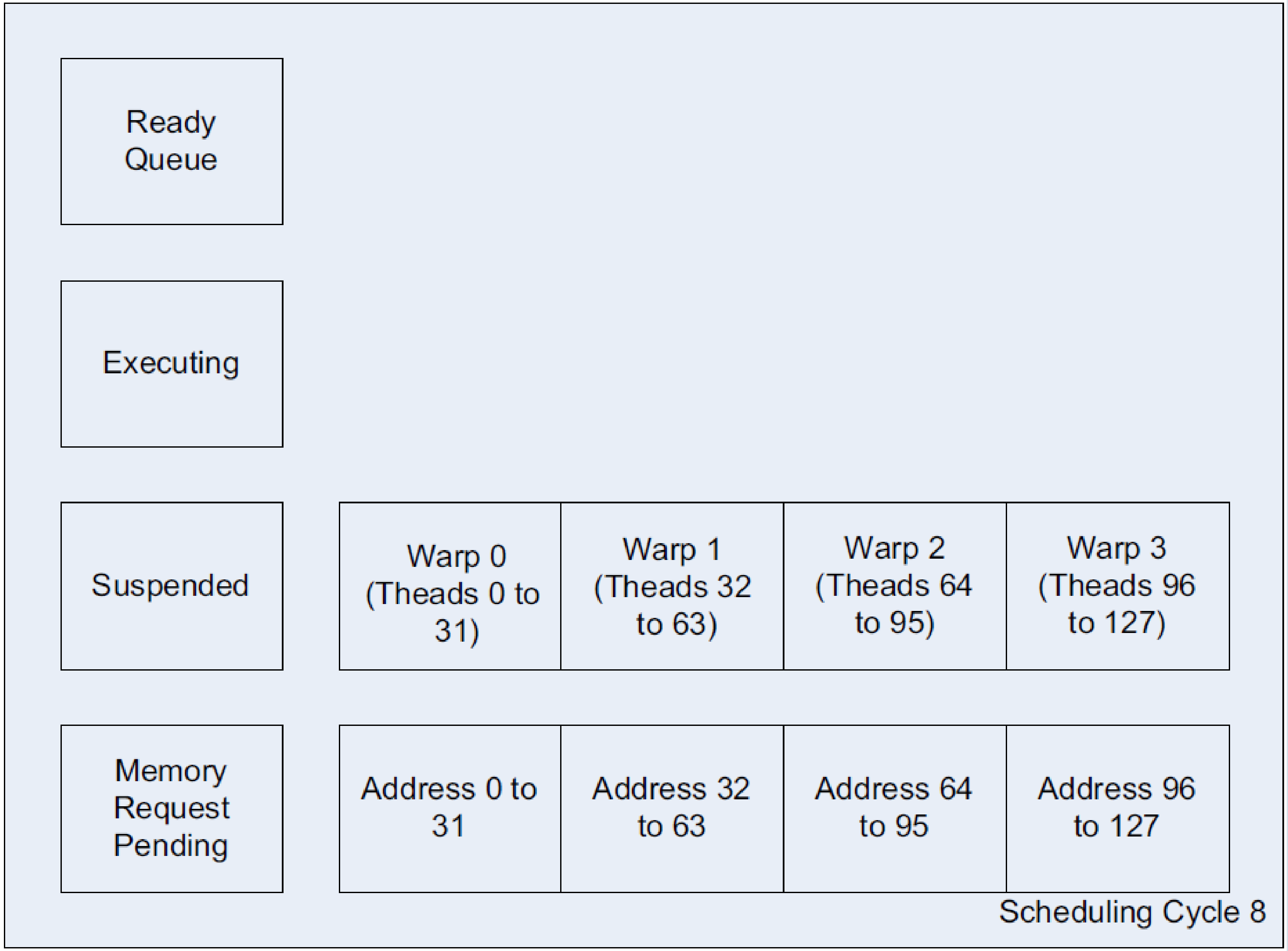
CUDA Scheduling

- Block Scheduling
  - ✓ Execute in warps of 32 threads



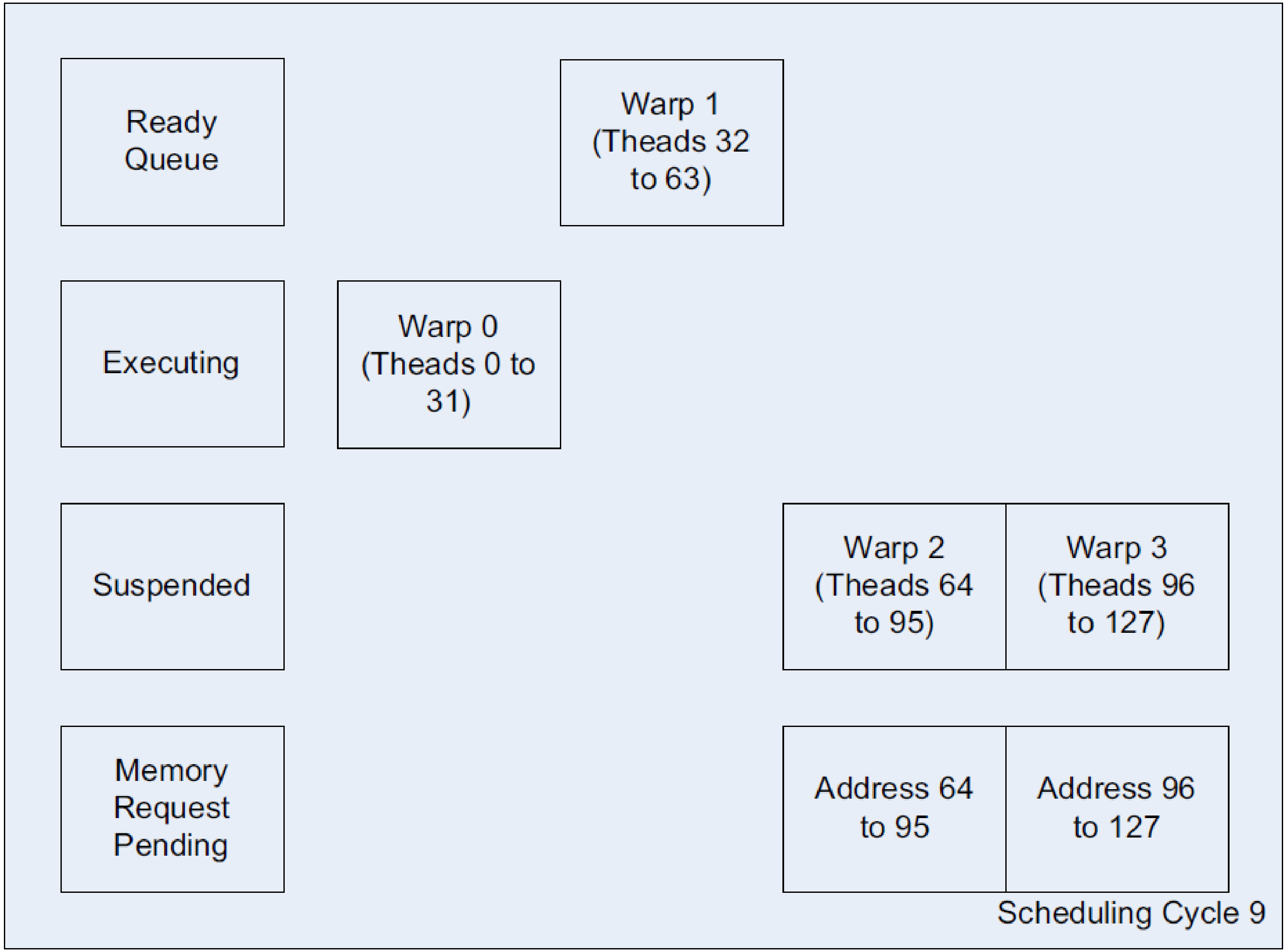
CUDA Scheduling

- Block Scheduling
  - ✓ Execute in warps of 32 threads



CUDA Scheduling

- Block Scheduling
  - ✓ Execute in warps of 32 threads



Outline

1. Review Lec1 & 2
2. Multithreading
3. CUDA Abstraction
4. Warp Scheduling
5. **Lab 2.1**
6. Software Layers in CUDA

Lab 2.1 Warp Scheduling

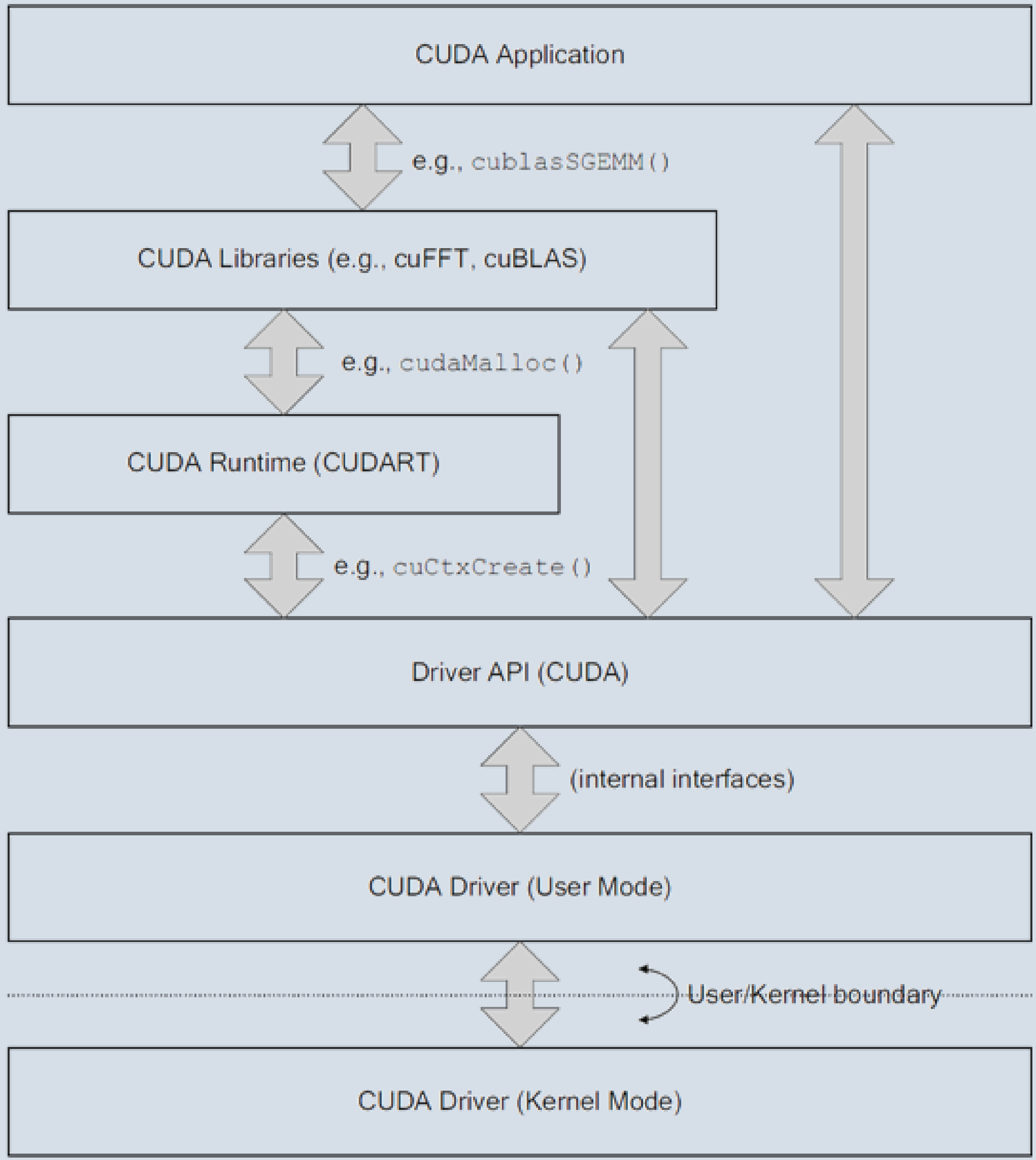
- 理解线程束的调度机制
  - ✓ 验证warp的线程数量
  - ✓ 加入计时功能，对warp的调度时间进行输出，并绘出散点图进行分析
  - ✓ 变大block和grid的大小会如何？
  - ✓ 给出对线程束调度机制的理解
  - ✓ 参见COOK 5.3 和WILT 7.3.3

```
__global__ void what_is_my_id(unsigned int * const block,
              unsigned int * const thread,
              unsigned int * const warp,
              unsigned int * const calc_thread)
{
              /* Thread id is block index * block size t thread offset into the block */
              const unsigned int thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
              block[thread_idx] = blockIdx.x;
              thread[thread_idx] = threadIdx.x;
              /* Calculate warp using built in variable warpSize */
              warp[thread_idx] = threadIdx.x / warpSize;
              calc_thread[thread_idx] = thread_idx;
}
```

Outline

1. Review Lec1 & 2
2. Multithreading
3. CUDA Abstraction
4. Warp Scheduling
5. Lab 2
6. **Software Layers in CUDA**

- Software Layer



- nvcc compiling