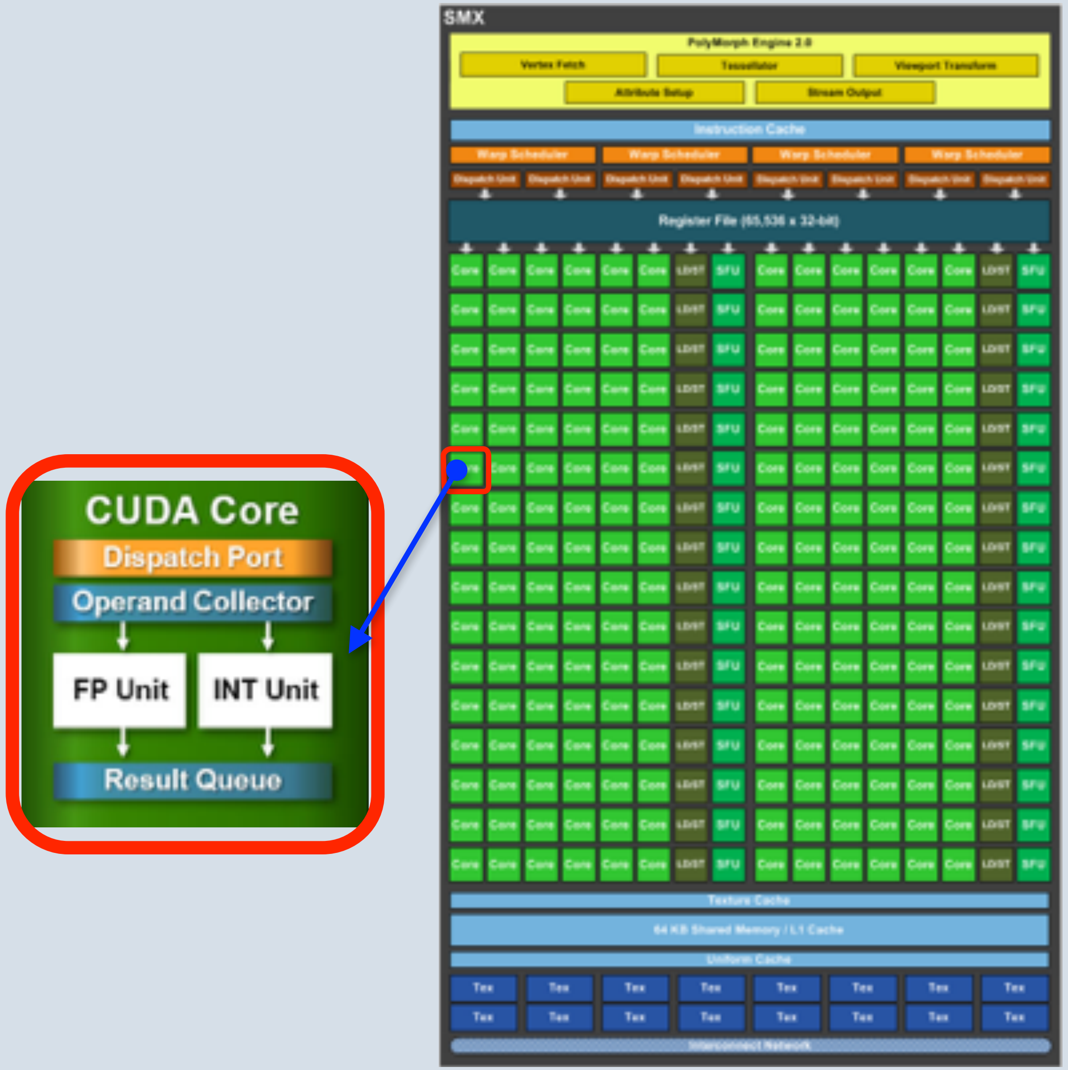
Review on Lec 1 & Lec 2

Hello CUDA

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

```
void main(){
        ......
        int *dev_a,*dev_b,*dev_c;
        cudaMalloc((void**)&dev_c, 128* sizeof(int));
        ......
        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);

        cudaMemcpy(c, dev_c, 128* sizeof(int), cudaMemcpyDeviceToHost);

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

Kepler SMX



Lec 3 CUDA Software Abstraction

Dong Li, Tonghua Su

School of Software

Harbin Institute of Technology

Outline

1. Multithreading
2. CUDA Abstraction
3. Kernel Execution
4. Warp Scheduling
5. CUDA Toolchain

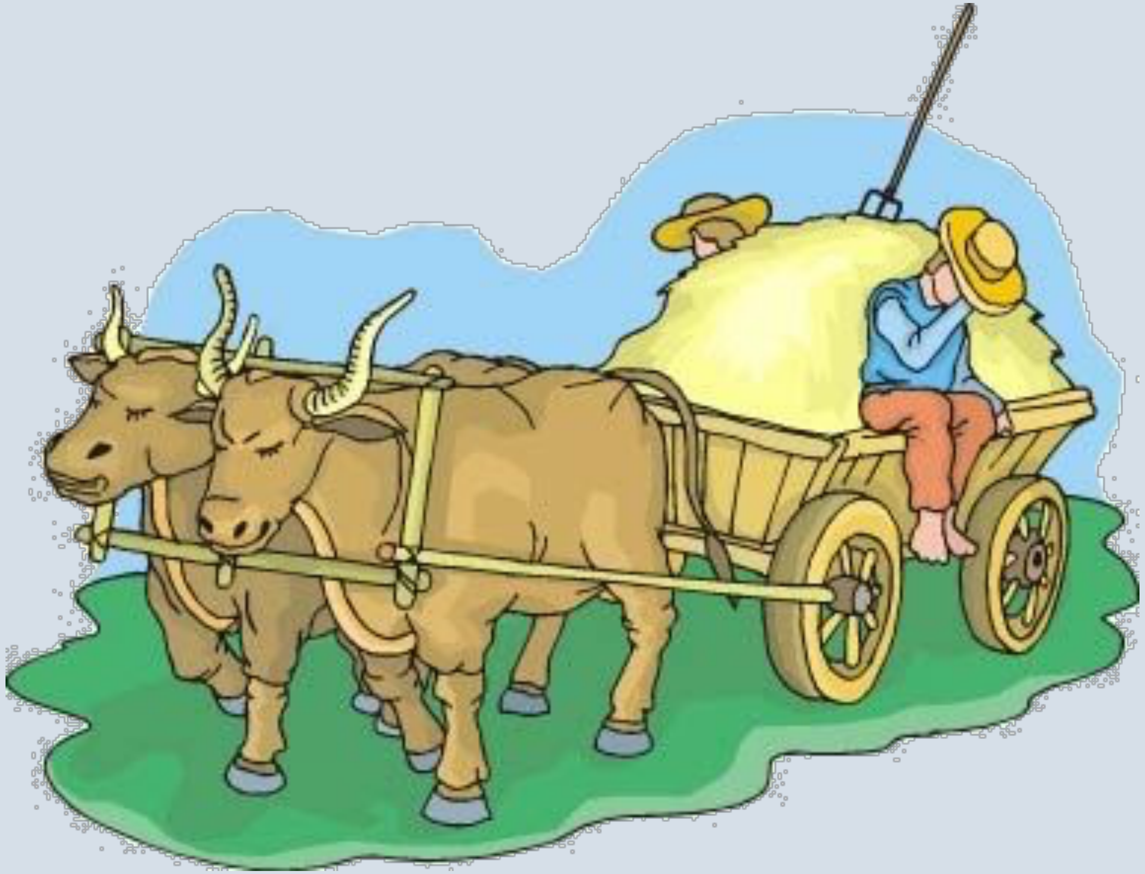
Outline

1. Multithreading
2. CUDA Abstraction
3. Kernel Execution
4. Warp Scheduling
5. CUDA Toolchain

Design Philosophy

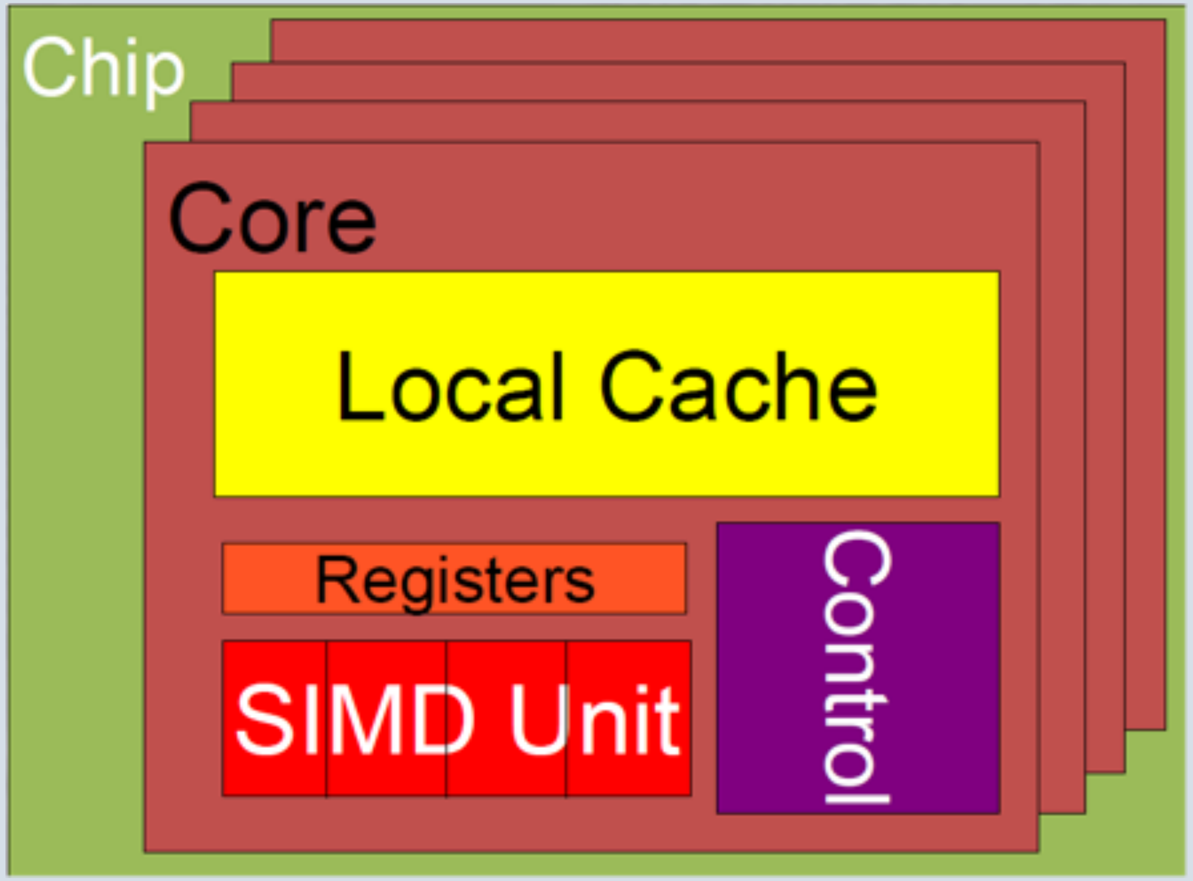
CPU: Latency Oriented Cores

GPU: Throughput Oriented Cores

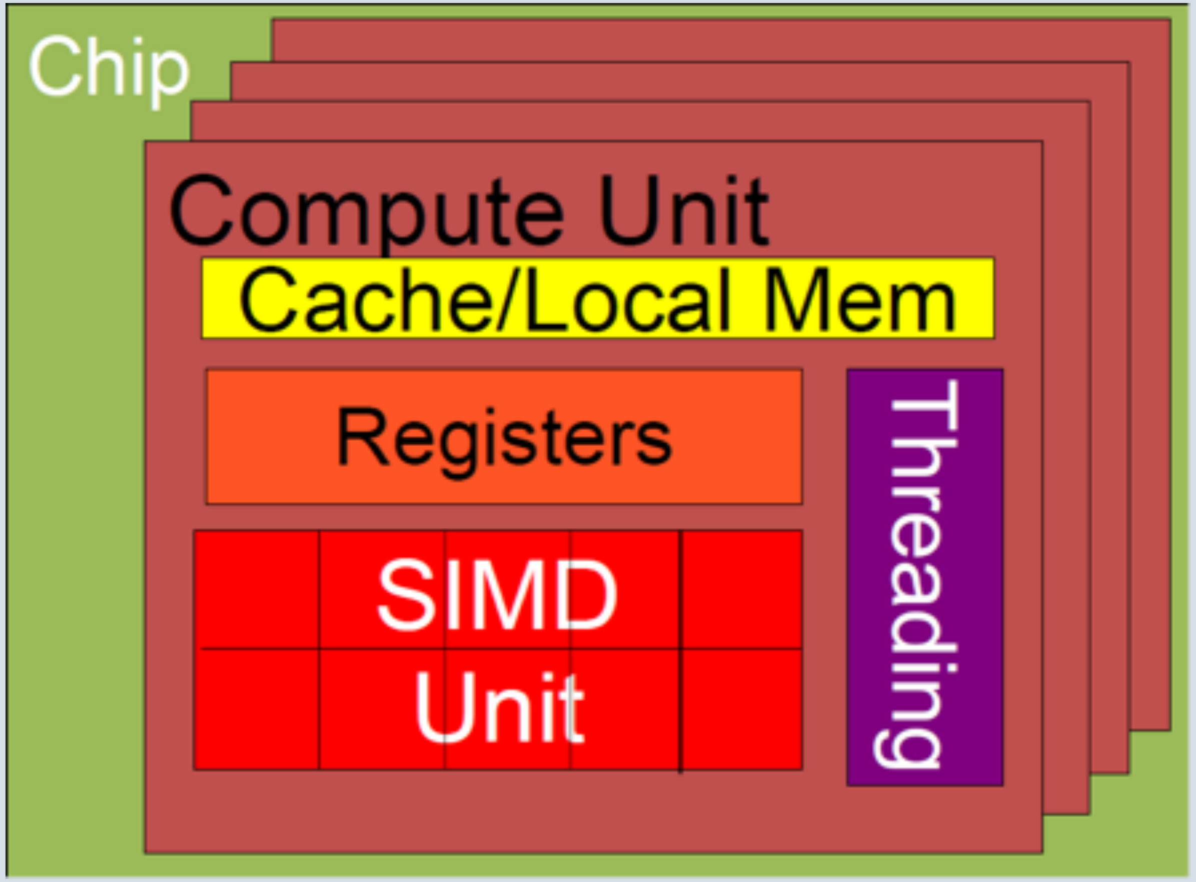


Design Philosophy

CPU: Latency Oriented Cores



GPU: Throughput Oriented Cores



Multithreading

- **SIMD in CPU**
  - ✓ All cores execute the same instructions simultaneously, but with different data
  - ✓ Similar to vector computing on CRAY supercomputers
  - ✓ e. g. SSE4, AVX instruction set
- **SIMT in SMX**
  - ✓ Multithreaded CUDA core
  - ✓ Threads on each SMX execute in group sharing same instruction
  - ✓ Fine-grained parallelism
  - ✓ Natural for graphics processing and much scientific computing
  - ✓ SIMT is also a natural choice for many-core chips to simplify each core

Multithreading

- Thread: instruction stream with own PC and data
  - ✓ Owning private register, private memory, program counter and thread execution state
  - ✓ Thread Level Parallelism(TLP): Exploit the parallelism inherent between threads
- Multithreading
  - ✓ Multiple threads to share the functional units of 1 processor via overlapping
  - ✓ Processor must duplicate independent state of each thread
    - e.g., a separate copy of register file, a separate PC
  - ✓ Often, hardware for fast thread switch
  - ✓ Memory to be shared
  - ✓ Solving the memory access stall

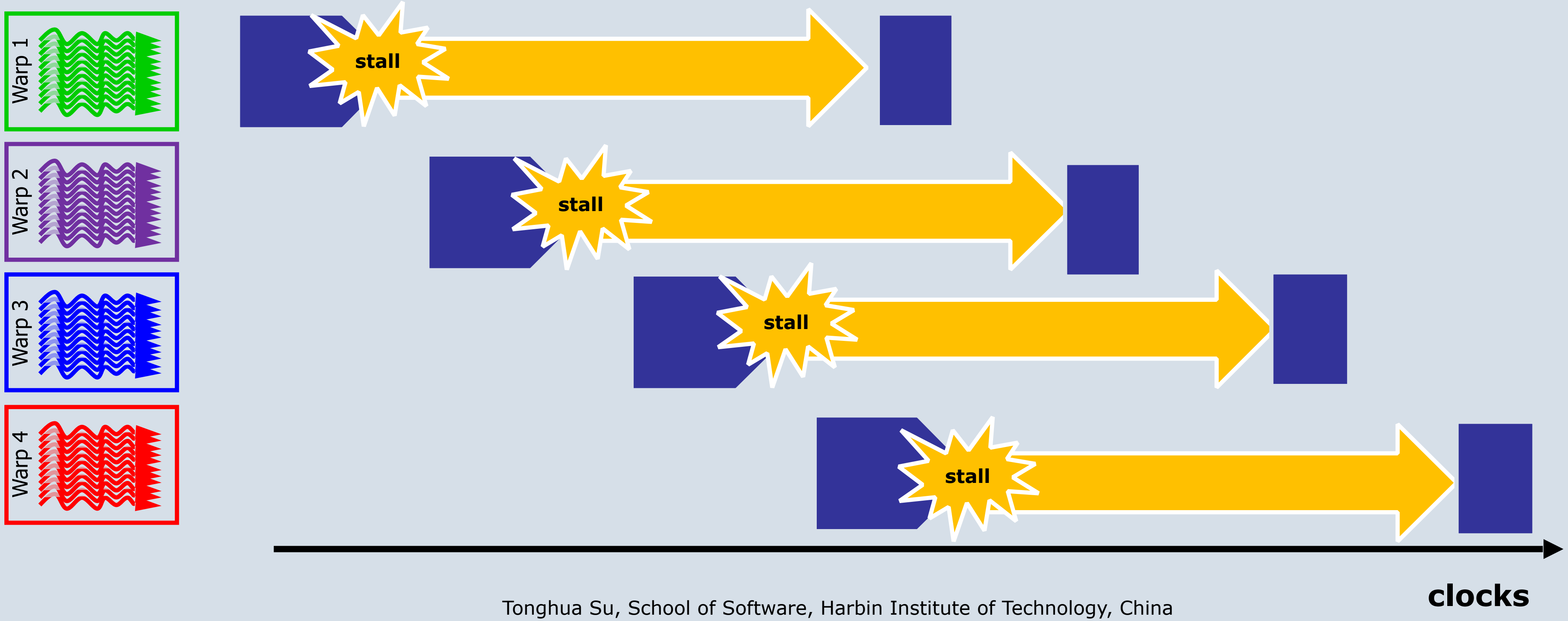
GPU Multithreading

- **Multithreaded Hardware**
  - ✓ **CUDA core is multithreaded processor**
    - Supporting 96 threads in GTX 8800
  - ✓ **Run in group of 32 threads (called a warp)**
  - ✓ **Zero-cost “context switching”**
    - Each thread has its own registers (which limits the number of active threads
  - ✓ **Shared memory/L1 cache**
- **Fine-grained multithreading**
  - ✓ **Able to switch between warps on each instruction**
  - ✓ **Schedule without pre-emption**
  - ✓ **Scheduling eligible warps in a round-robin fashion**

GPU Multithreading

●Hiding Latency Stalls

- ✓ Fetch → Decode → Execute → Memory → Writeback
- ✓ Execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data
- ✓ Lots of active warps is the key to high performance



Quiz

- 定量计算内存访问带来的停滞
  - ✓ 假设从显存读取数据的延时是 400时钟周期
  - ✓ 如果每个线程束（warp）可以运行10个周期，那么需要多少个就绪的线程束才能掩盖停滞带来的时间缝隙?

Outline

1. Multithreading
2. CUDA Abstraction
3. Kernel Execution
4. Warp Scheduling
5. CUDA Toolchain

CUDA

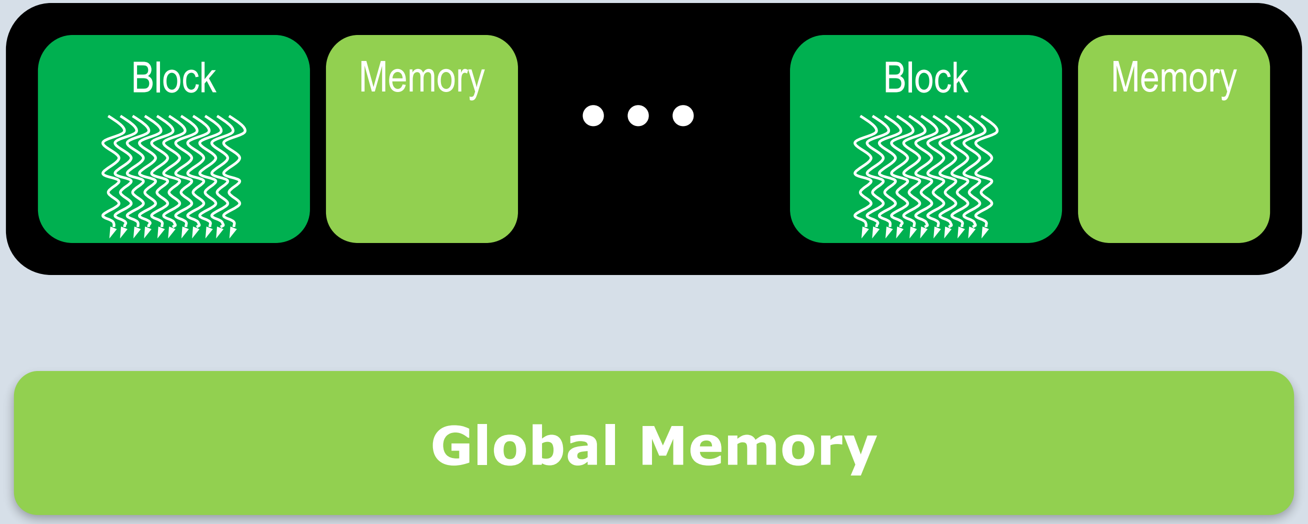
- CUDA(Compute Unified Device Architecture) is developed by Nvidia around 2007
  - ✓ 2-4 week learning curve for those with experience of OpenMP and MPI programming
  - ✓ large user community on NVIDIA forums
- CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the GPU

CUDA

- **CUDA as Parallel Computing Platform**
  - ✓ Language: CUDA C which based on C with some extensions(extensive C++ support)
  - ✓ Editor: Eclipse/Visual Studio
  - ✓ Complier: nvcc
  - ✓ SDK: CUDA toolkit, Libraries, Samples
  - ✓ Profiler & Debugger: Nsight
- **CUDA as Programming Model**
  - ✓ Software Abstraction of GPU hardwares
  - ✓ Independent to OSs, CPUs, Nvidia GPUs

CUDA Abstraction

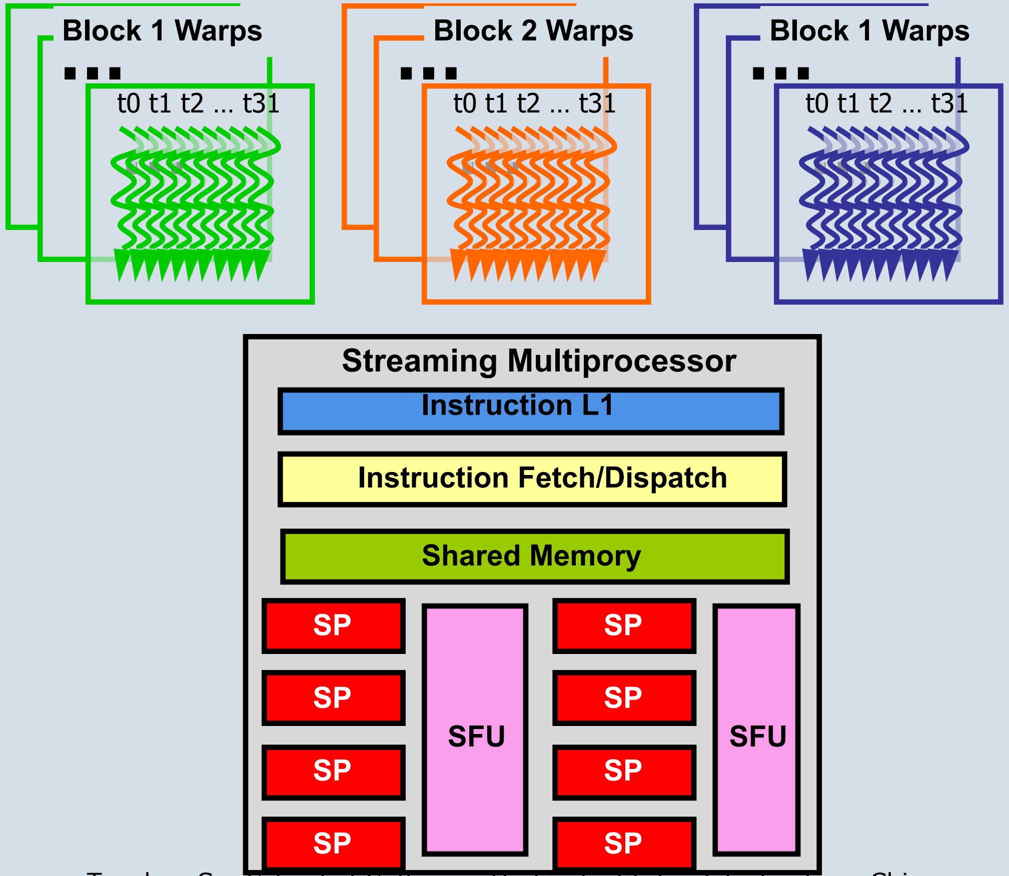
- **CUDA Virtualizes the Physical Hardware**
  - ✓ thread is a virtualized CUDA cores (registers, PC, state)
  - ✓ block is a virtualized streaming multiprocessor (threads, shared mem.)
- **Scheduled onto Physical Hardware without Pre-emption**
  - ✓ threads/blocks launch & run to completion/suspension
  - ✓ blocks should be independent



CUDA Abstraction

●Key Parallel Abstractions in CUDA

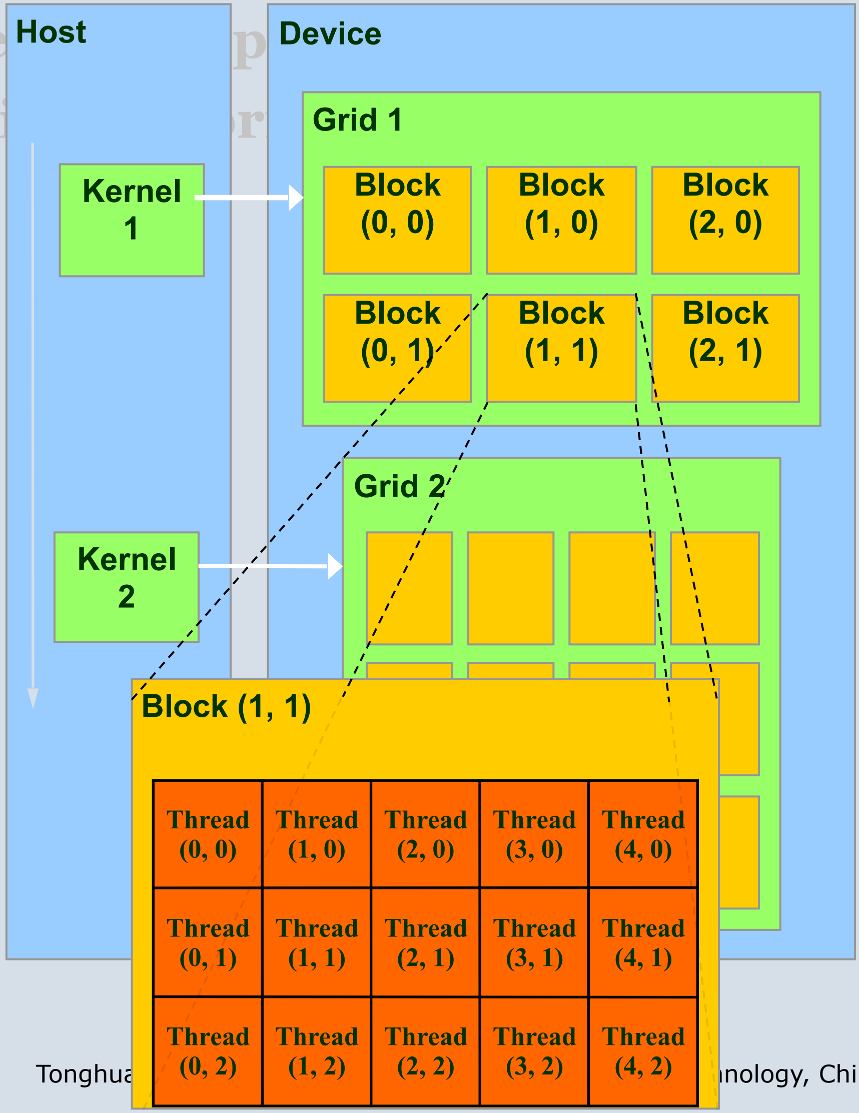
- ✓ Hierarchy of concurrent threads
- ✓ Shared memory model for cooperating threads
- ✓ Lightweight synchronization primitives



CUDA Abstraction

●Key Parallel Abstractions in CUDA

- ✓ Hierarchy of concurrent threads
- ✓ Shared memory mode p
- ✓ Lightweight synchroni pr

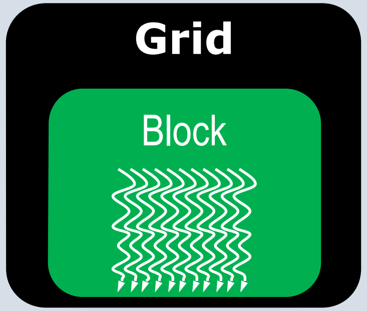


Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)

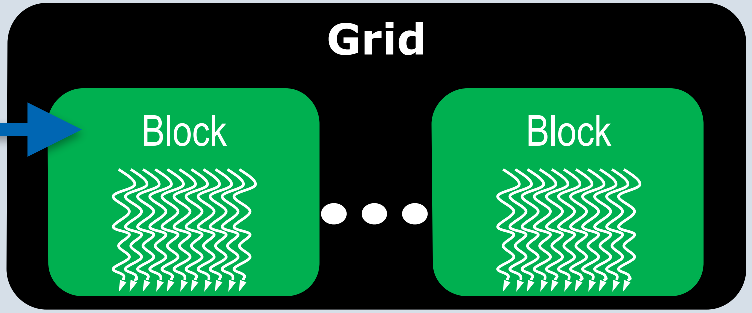
Thread Hierarchy

- **Thread —> Block—> Grid**

```
void main(){
     ……
     int *dev_a,*dev_b,*dev_c;
     ……
     addKernel<<<1, 128>>>(dev_c, dev_a, dev_b);
     ……
}
```

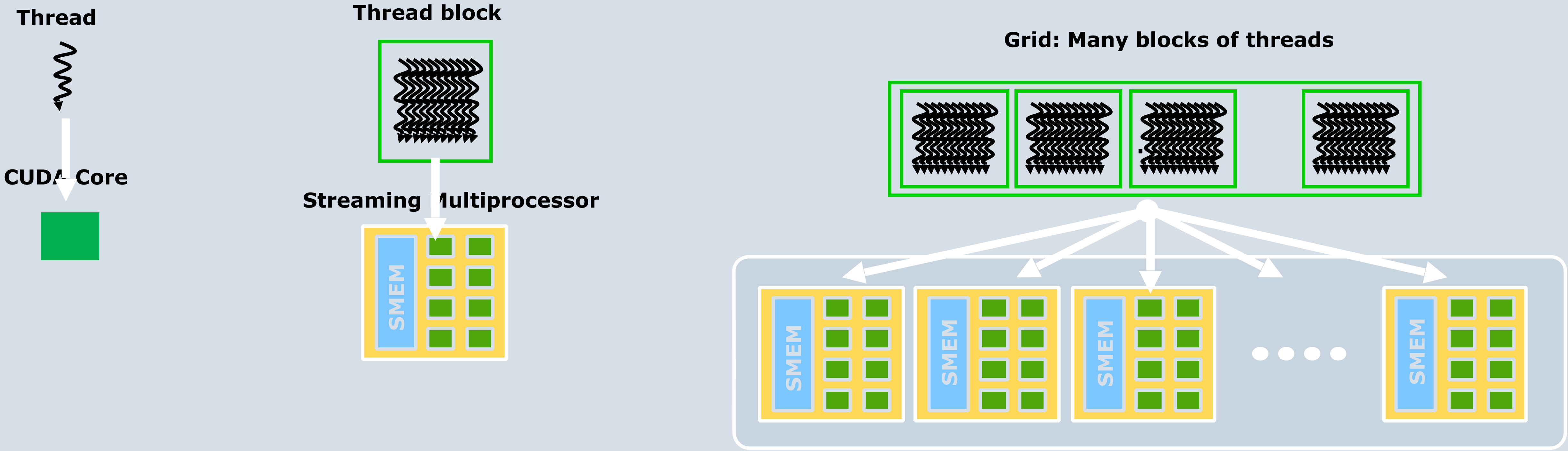


```
void main(){
     ……
     int *dev_a,*dev_b,*dev_c;
     ……
     addKernel<<<100, 128>>>(dev_c, dev_a, dev_b);
     ……
}
```

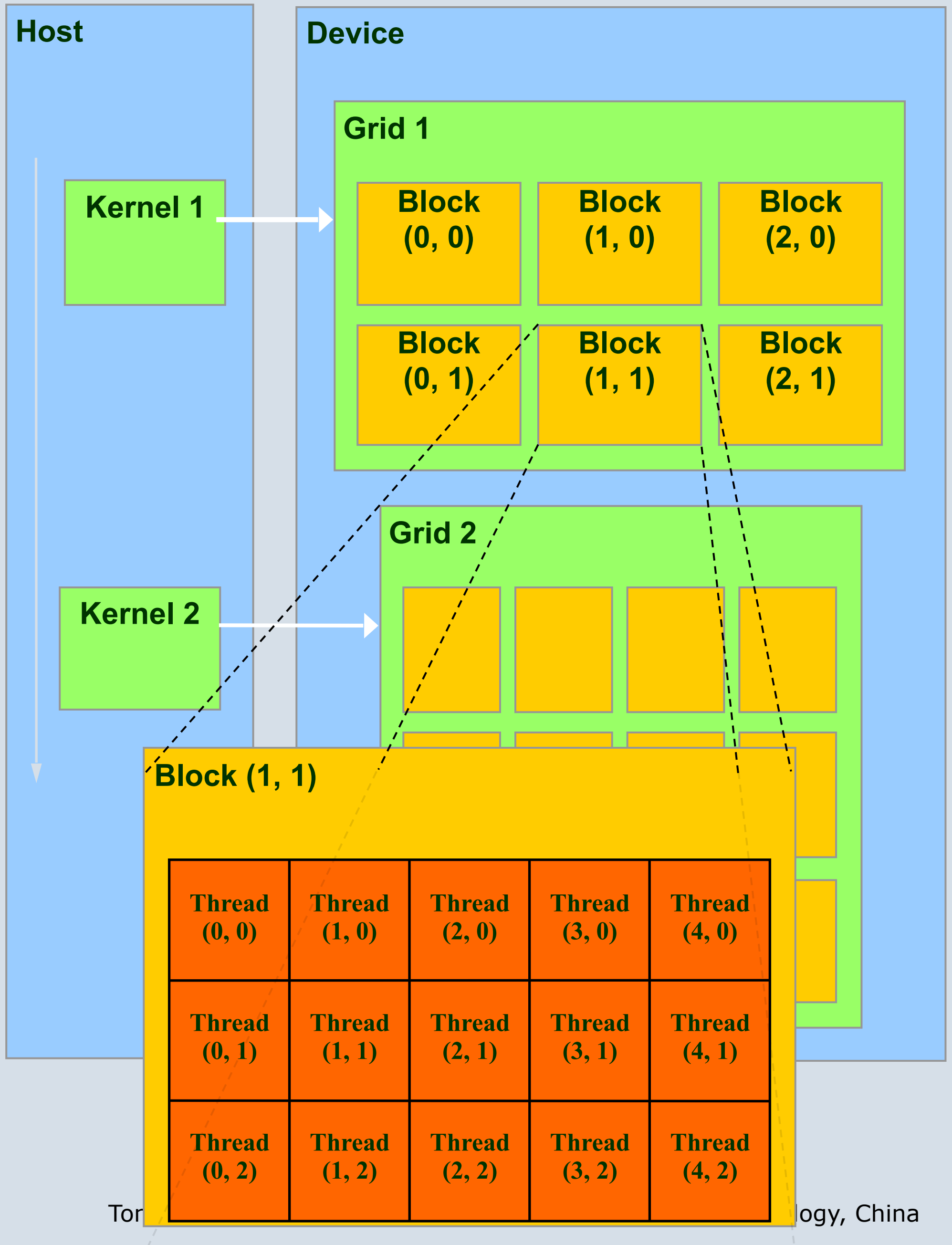


Thread Hierarchy

- **Thread Mapping**

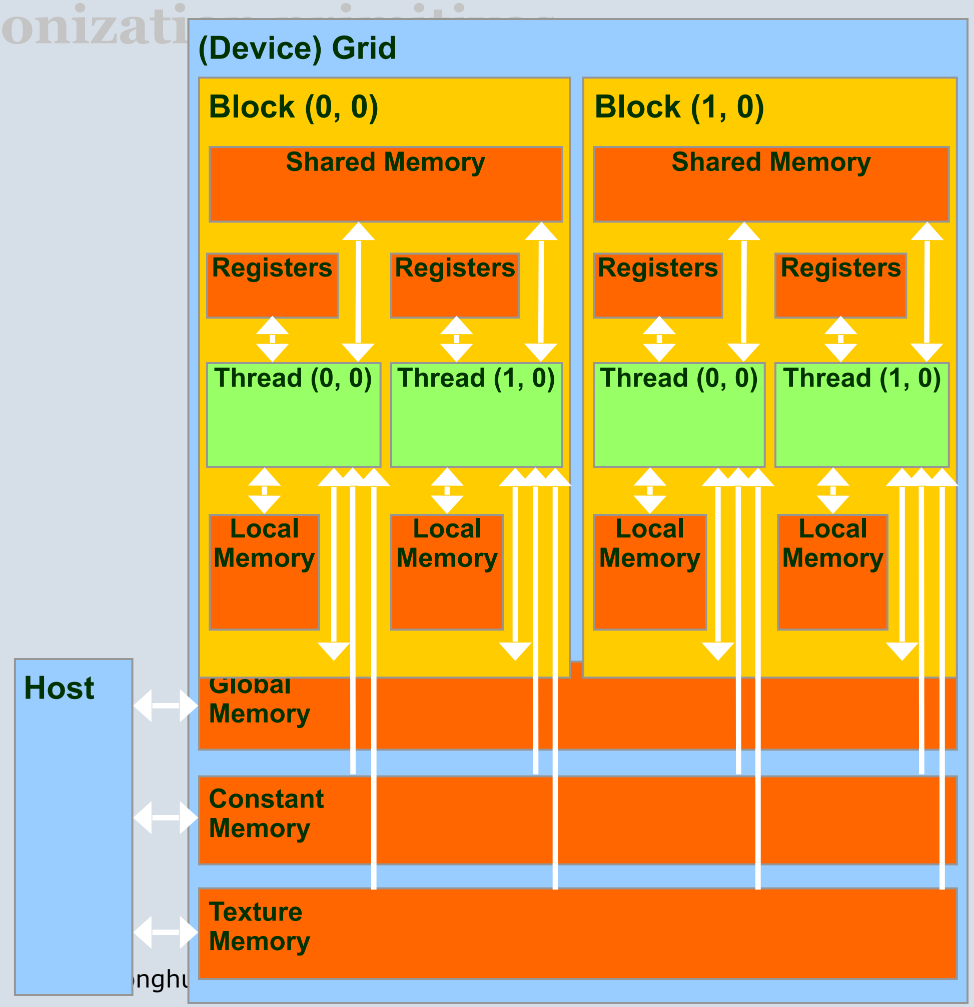


Thread Hierarchy

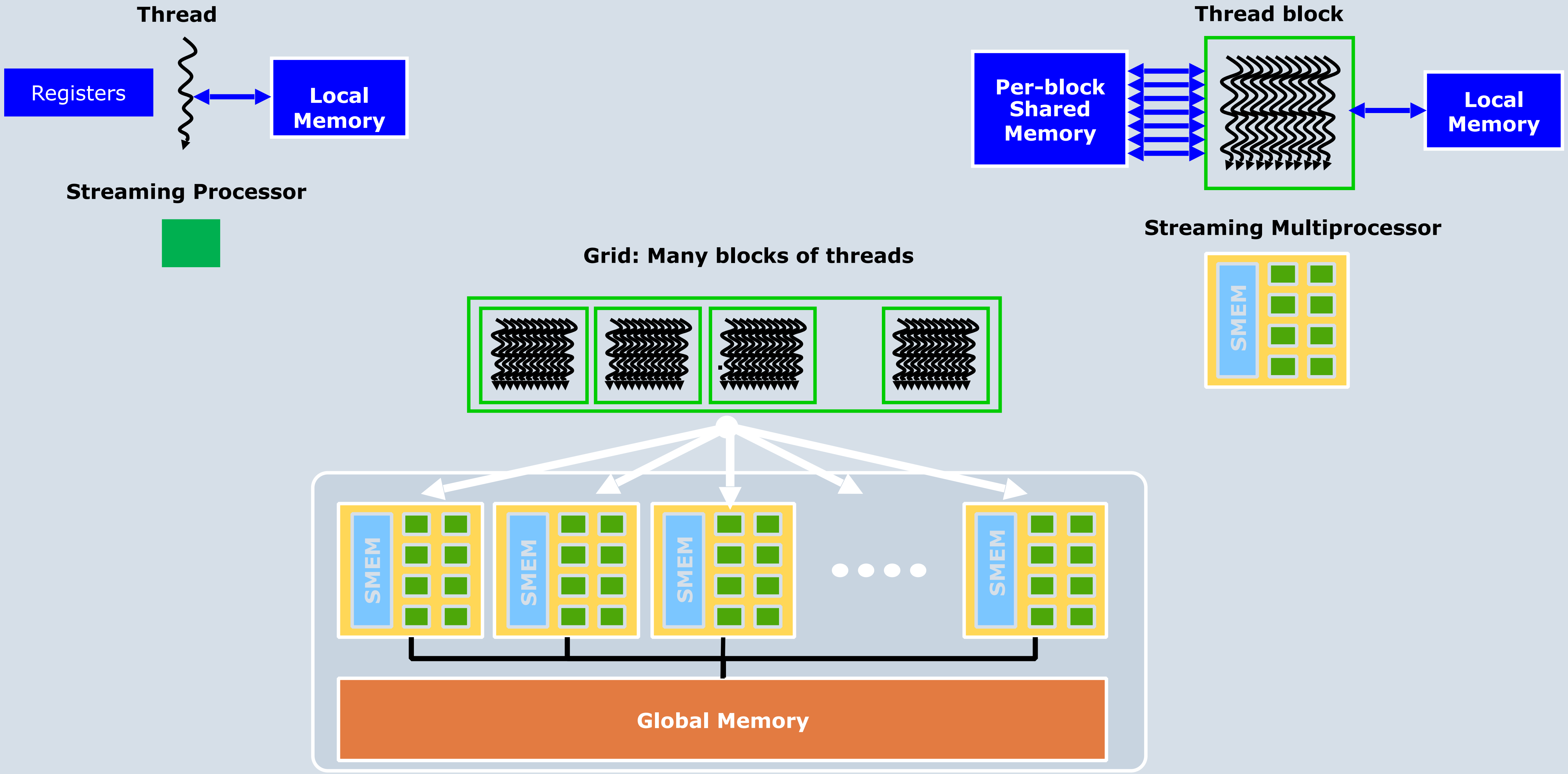


CUDA Abstraction

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

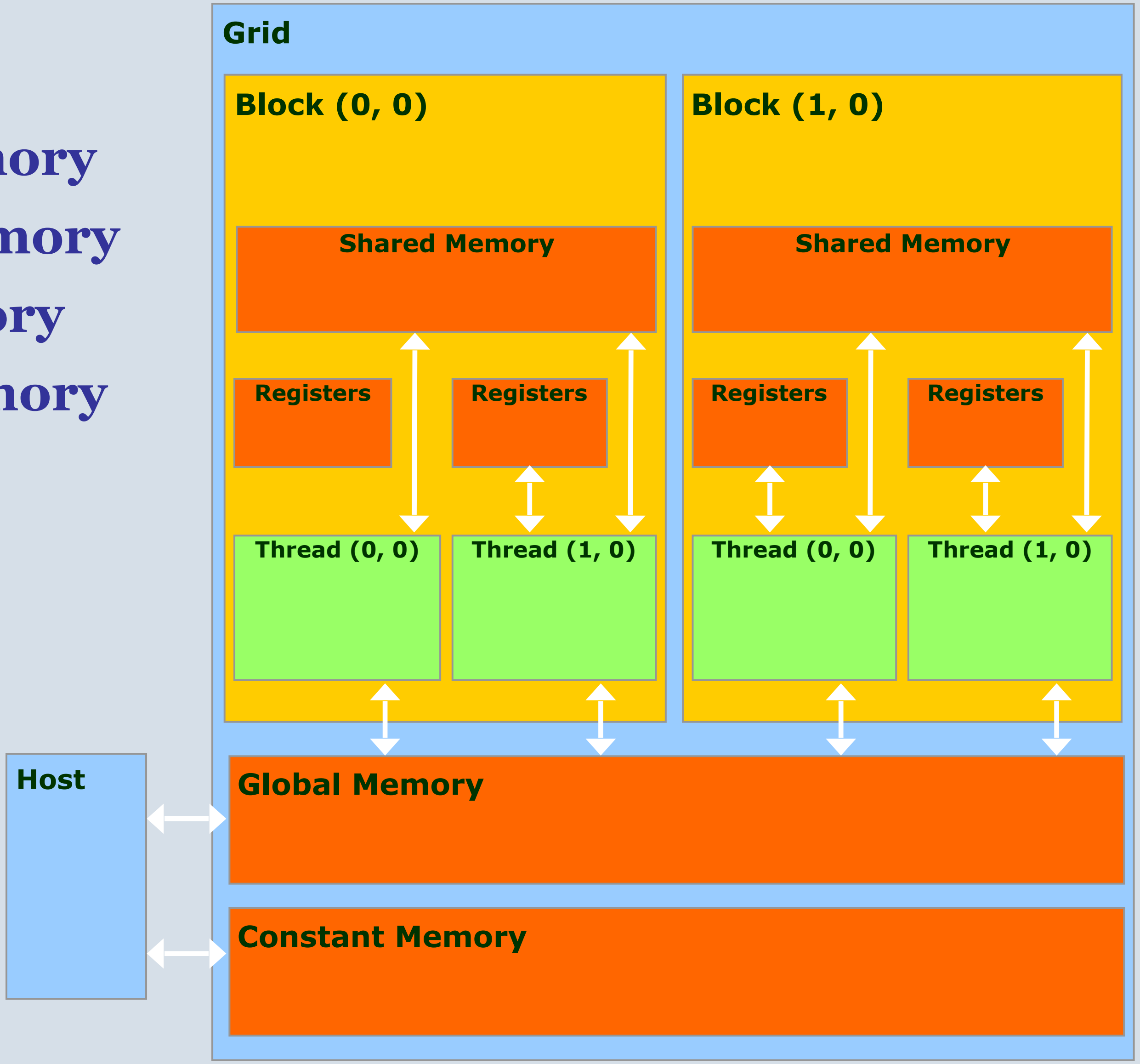


Memory Model



Memory Model

- **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**

Synchronization

●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

Synchronization

●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];
  }
}
```

Synchronization

●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

Synchronization

- **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!

Synchronization

- **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);
}
```

Quiz

- 填空
  - ✓ CUDA中，( )是对CUDA core的抽象，( )是对SMX的抽象
  - ✓ CUDA的三大抽象技术分别是( )、( )和( )
- 简答
  - ✓ 请自查资料，说明atomicCAS的作用，如可能给出实例

Outline

1. Multithreading
2. CUDA Abstraction
3. Kernel Execution
4. Warp Scheduling
5. CUDA Toolchain

CUDA Function

●Kernel Function

- ✓ C-style function that is executed in parallel by more than one thread
- ✓ defined using __global__ declaration specifier and returns void

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

●CUDA Function

- ✓ __device__ and __host__ can be used together

Sytax	*Executed on*	*Only callable from*
__device__ float DeviceFunc()	***device***	***device***
__global__ void KernelFunc()	***device***	***host, device(since sm3.x)***
__host__ float HostFunc()	***host***	***host***

Kernel Function

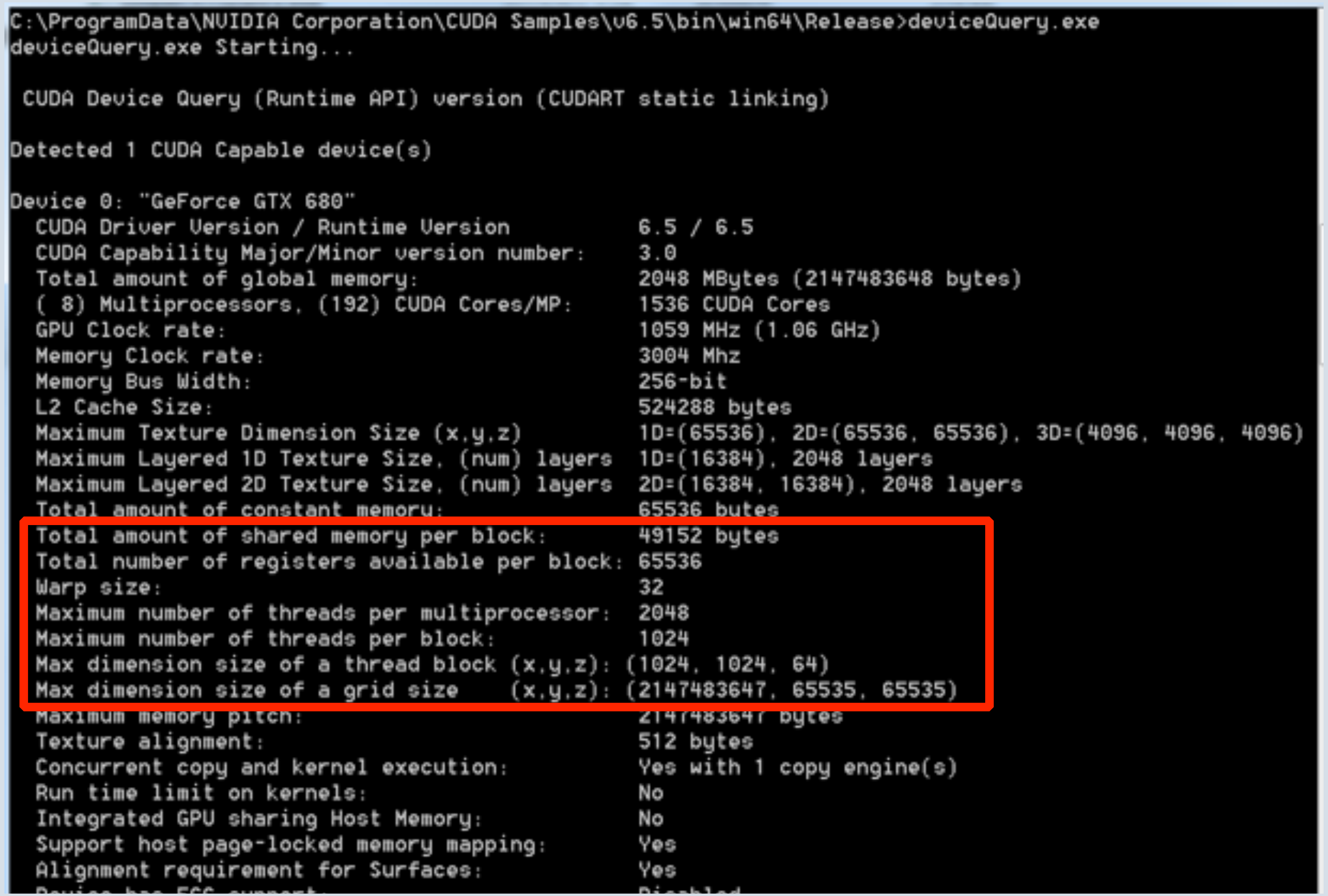
- **Kernel Invocation Looks Like:**

  KernelFunc<<<gridDim, blockDim, nSMem,iStream>>>(args);

- ✓ **<<<>>> identify the execution configuration of the kernel**
- ✓ **gridDim is the dimension of the grid**
- ✓ **blockDim is the dimension of each block**
- ✓ **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 (dim3 type) to simplify application programs**

Kernel Function

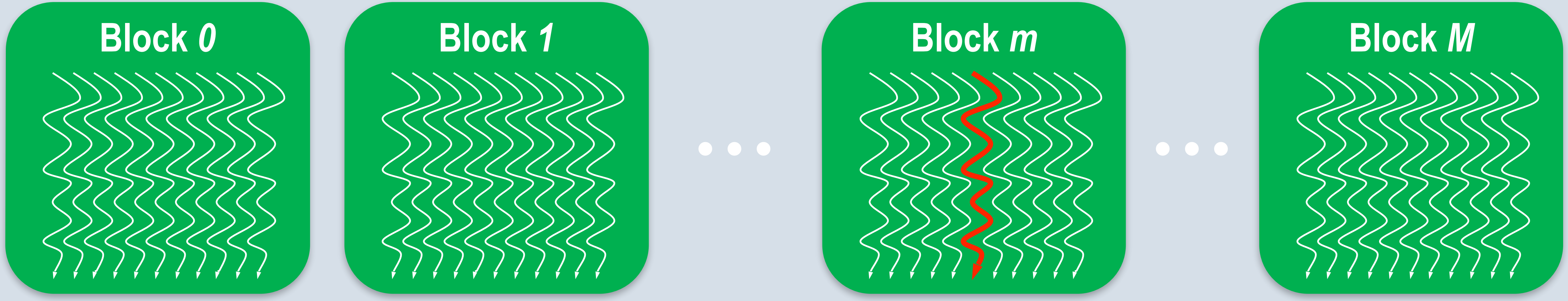
●Kernel Dimensional Limitation:



```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v6.5\bin\win64\Release>deviceQuery.exe
deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 680"
  CUDA Driver Version / Runtime Version          6.5 / 6.5
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Clock rate:                                1059 MHz (1.06 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
```

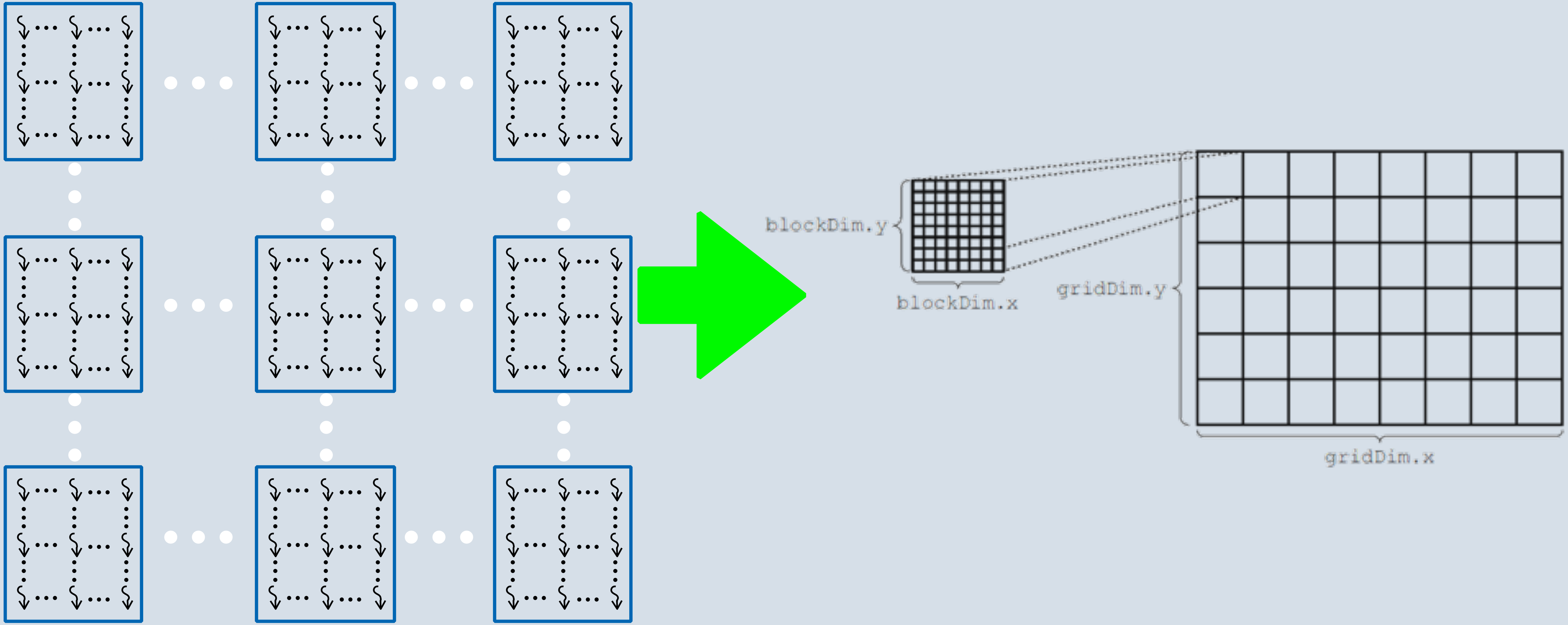
Thread Indexing

- 1D block and 1D grid



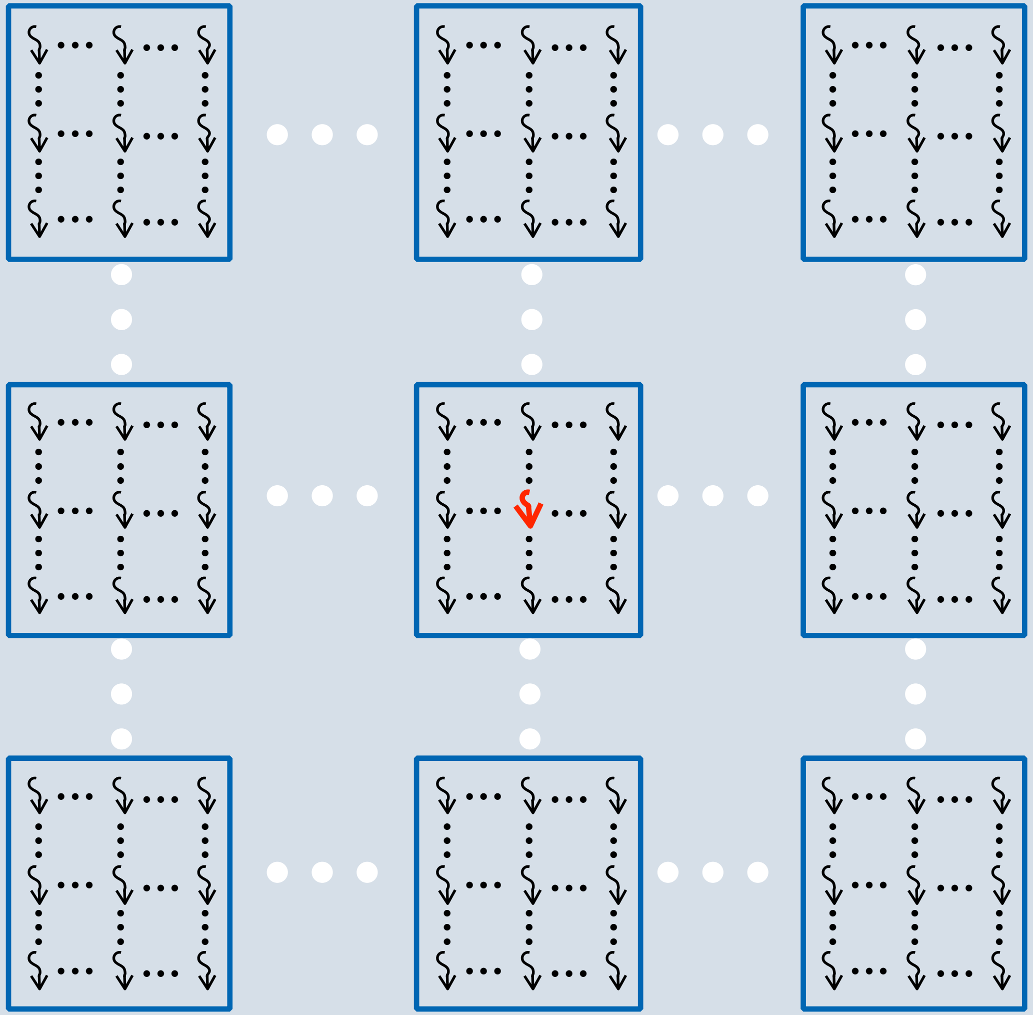
Thread Indexing

- **2D block and 2D grid**



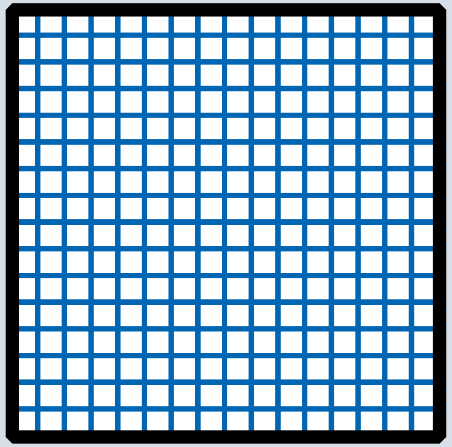
Thread Indexing

- 2D block and 2D grid

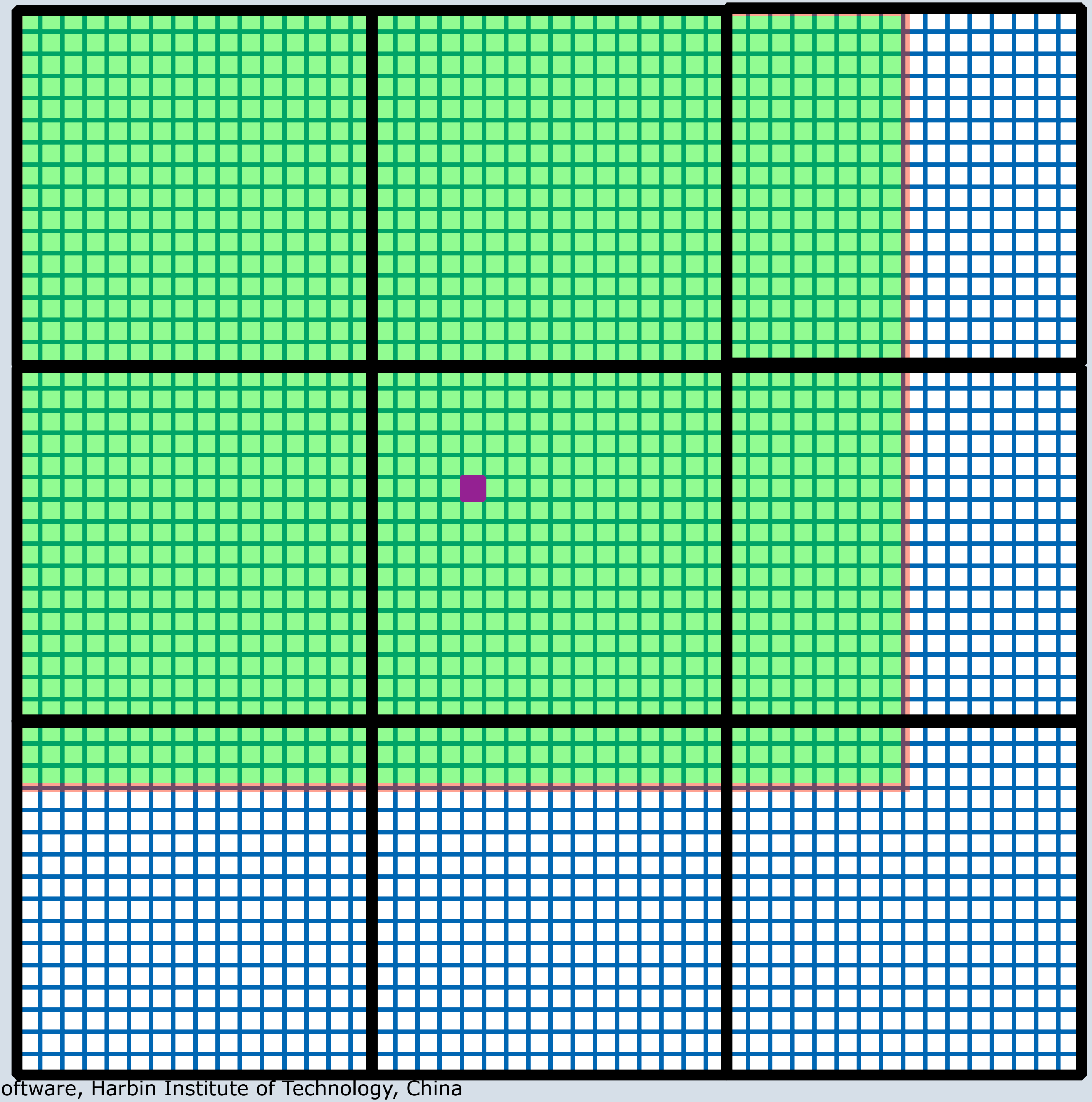


Data Indexing

- 2D block and 2D grid
  - ✓ Using the blocks to tile the data

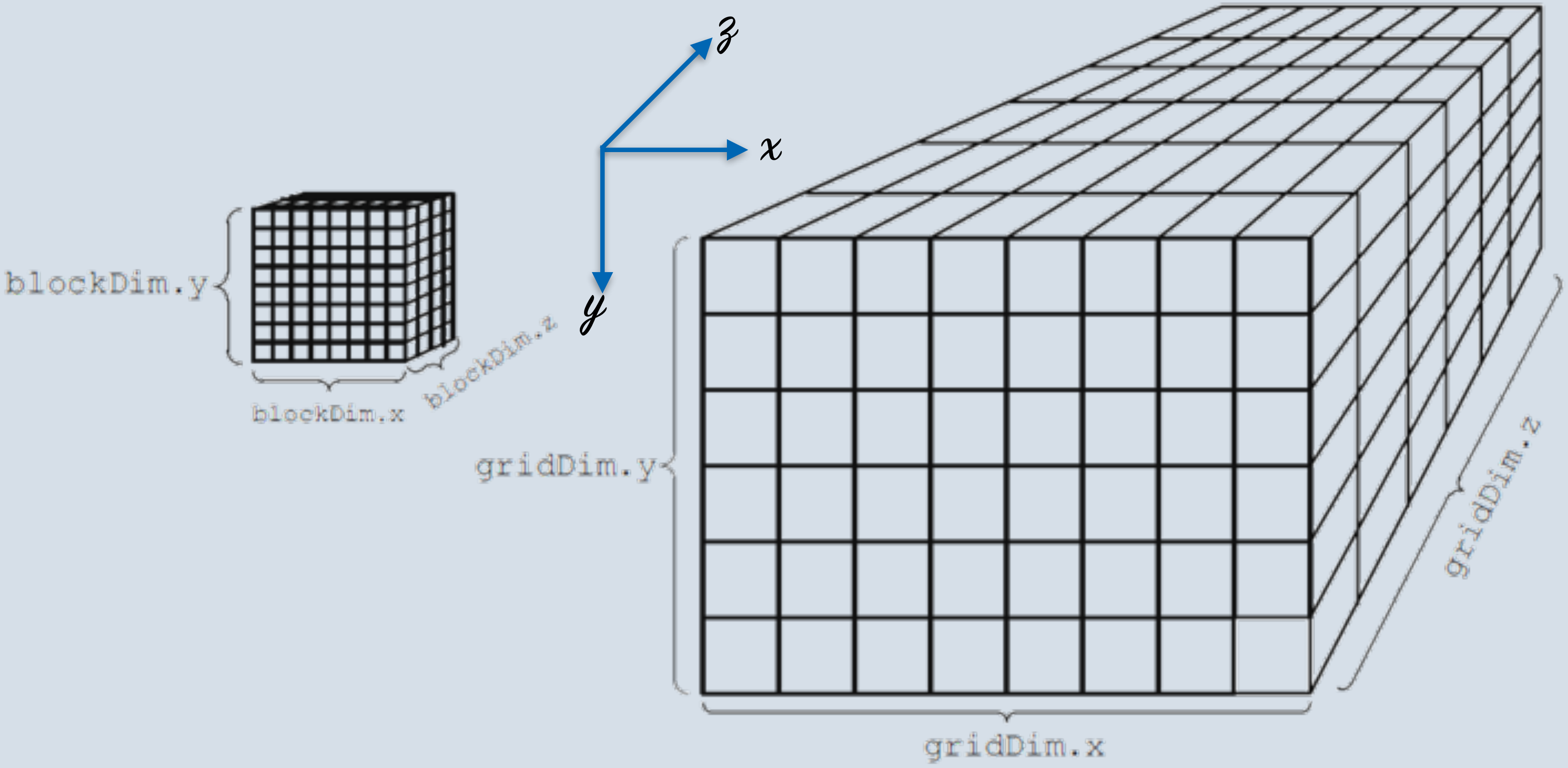


16*16 block



Thread Indexing

- 3D block and 3D grid



Occupancy

- **Occupancy is a useful metric**
  - ✓ **the ratio of the number of active warps per multiprocessor to the maximum number of warps that can be active on the multiprocessor at once**
  - ✓ **the denominator is a constant that depends only on compute capability of the device**
  - ✓ **the numerator of this expression is a function of the following:**
    - compute capability
    - threads per block
    - registers per thread
    - shared memory per block and shared memory configuration

Quiz 1

- 计算下面内核配置在**Kepler**架构**GPU**上的设备占用率

```
dim3 blockDim, gridDim;
blockDim.x = 16;
blockDim.y = 16;
gridDim.x = (1000-blockDim.x+1)/blockDim.x;
gridDim.y = (3755-blockDim.y+1)/blockDim.y;
distKernel<<<gridDim,blockDim>>>(...)
```

Quiz 2

- 计算线程对应的数组下标
  - ✓ 灰度图像大小为100*100大小，每个元素的取值在0~255之间，
  - ✓ 现在按行展开的方式存储在一维数组a里
  - ✓ 现在启动如下配置的内核，每个线程操作对应于a的一个元素，请问第(p,q)个block的第(m,n)个线程要操作数组a的第几个元素?

```
blockDim.x = 16;
blockDim.y = 16;
gridDim.x = (100-blockDim.x+1)/blockDim.x;
gridDim.y = (100-blockDim.y+1)/blockDim.y;
calKernel<<<gridDim,blockDim>>>(...)
```

Outline

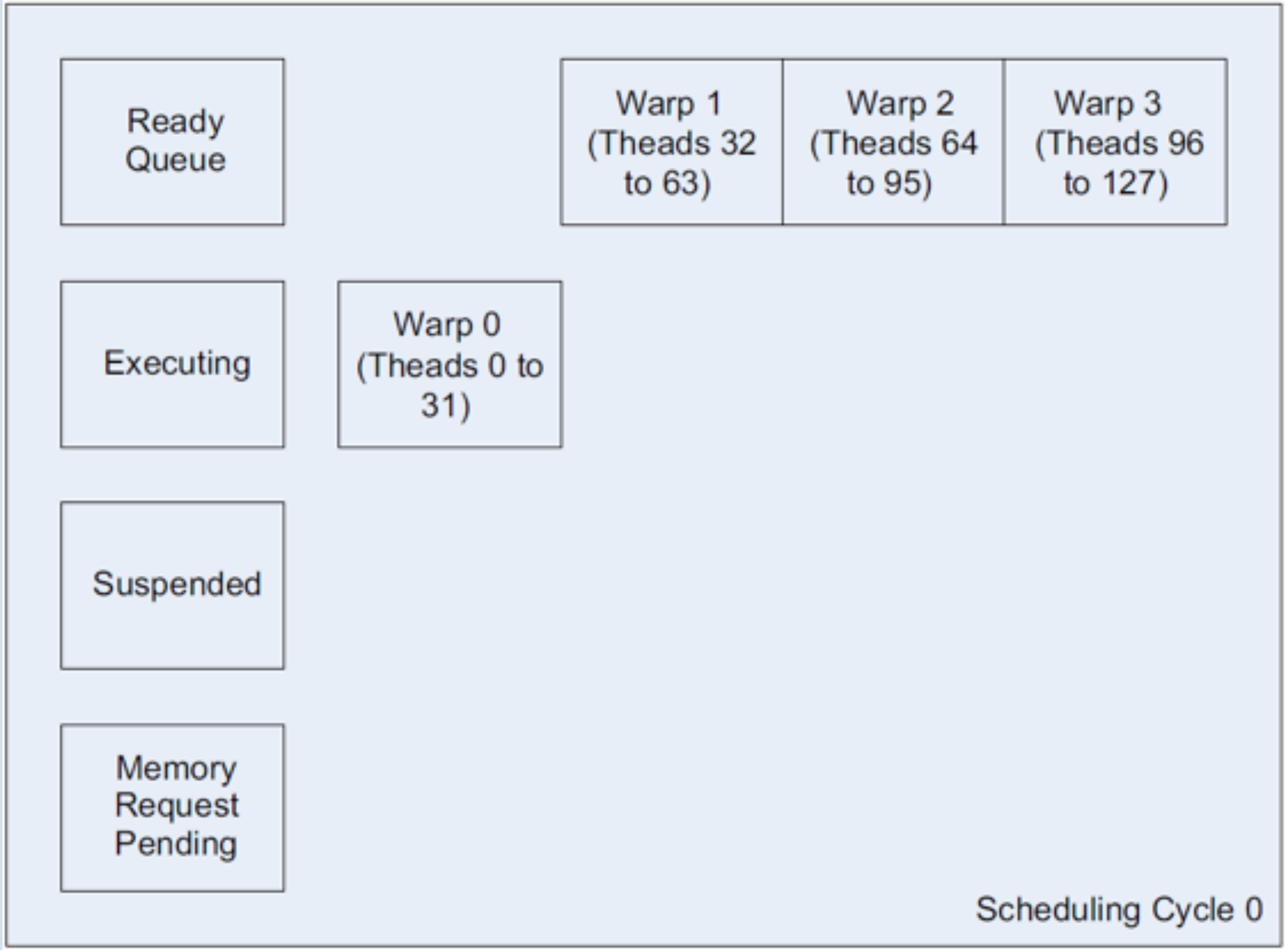
1. Multithreading
2. CUDA Abstraction
3. Kernel Execution
4. Warp Scheduling
5. CUDA Toolchain

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

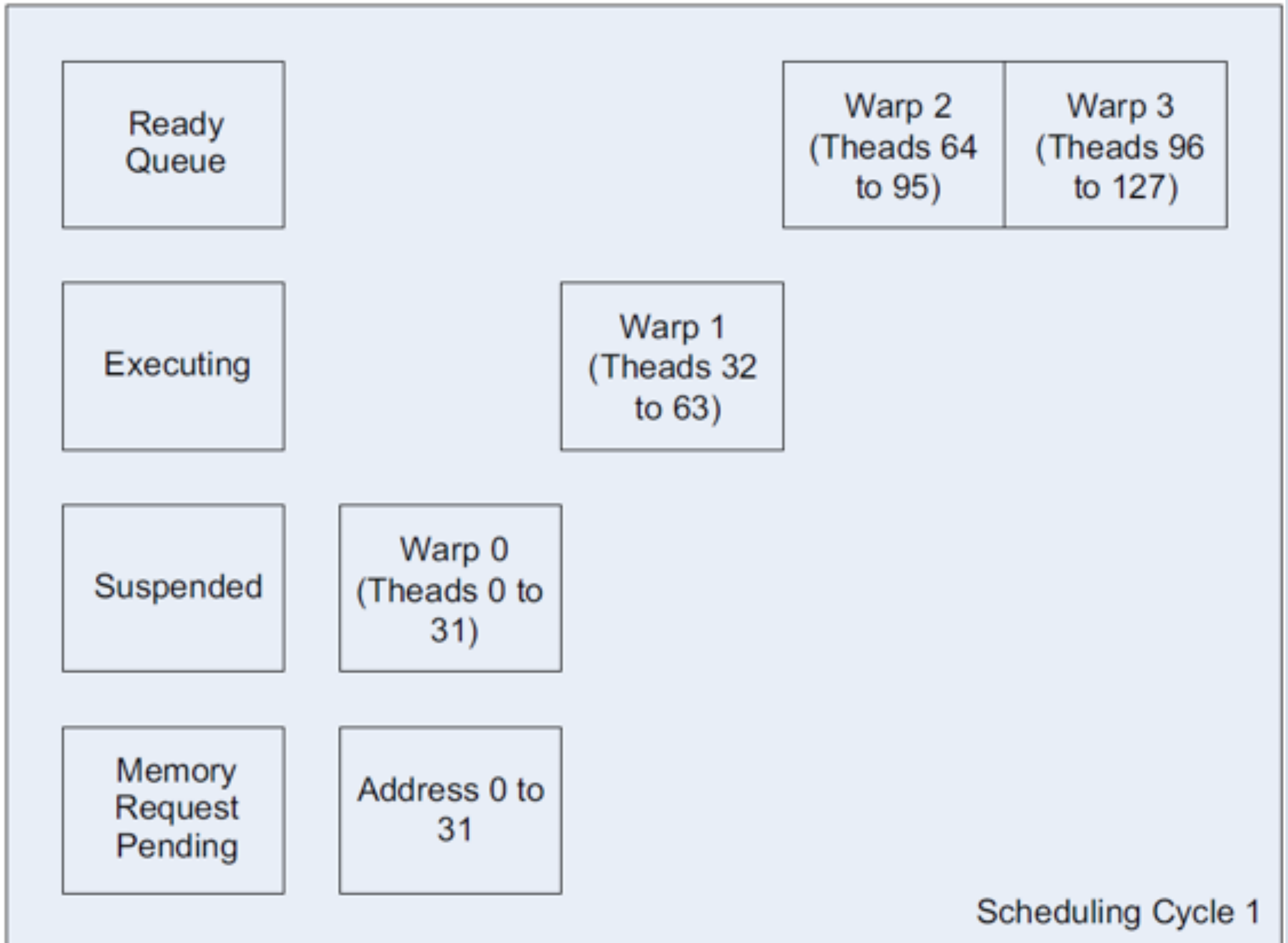
- **Block Scheduling**
  - ✓ Execute in warps of 32 threads



CUDA Scheduling

●Block Scheduling

✓ Execute in warps of 32 threads



Ready Queue

Warp 2 (Theads 64 to 95)

Warp 3 (Theads 96 to 127)

Executing

Warp 1 (Theads 32 to 63)

Suspended

Warp 0 (Theads 0 to 31)

Memory Request Pending

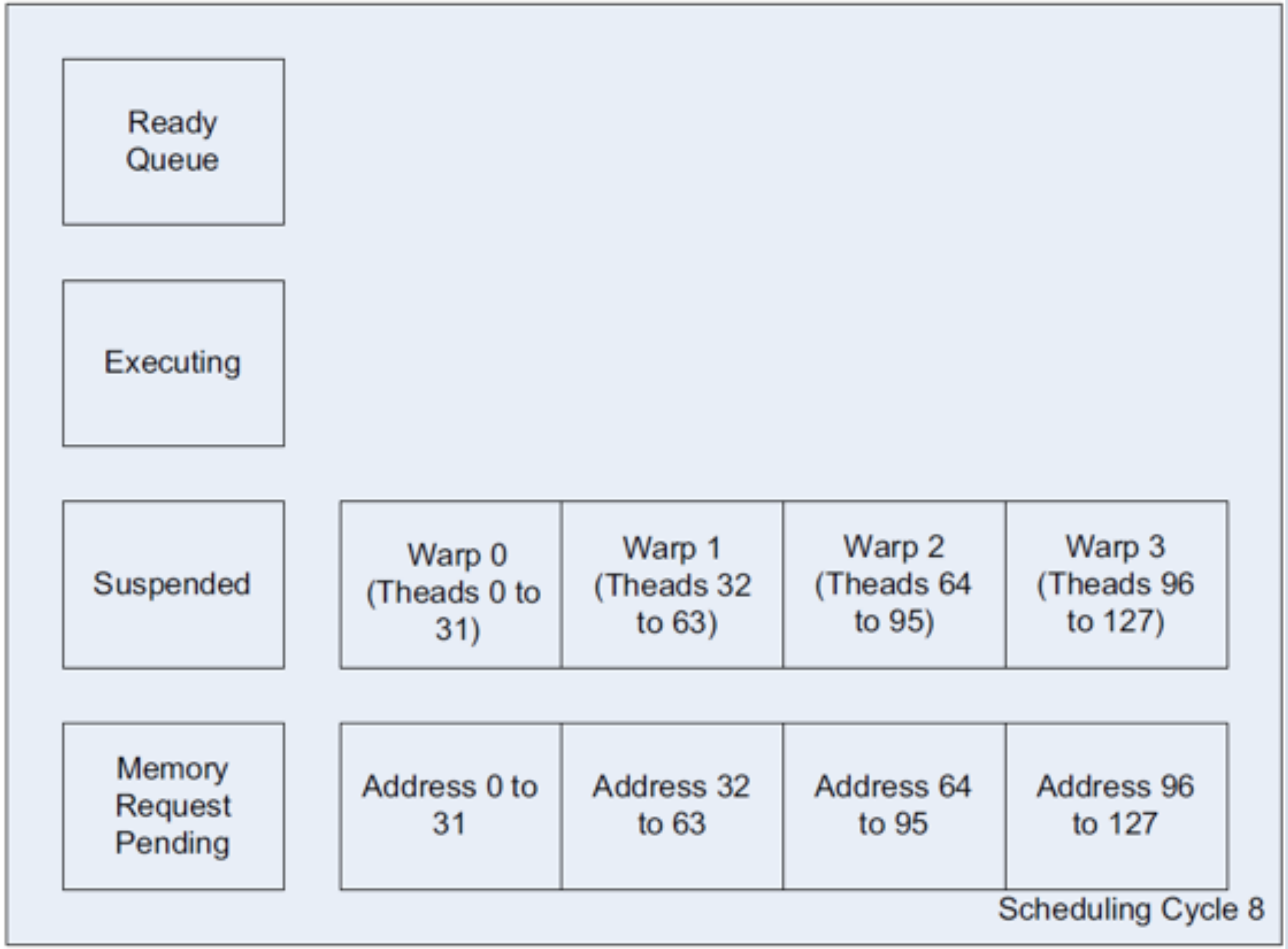
Address 0 to 31

Scheduling Cycle 1

CUDA Scheduling

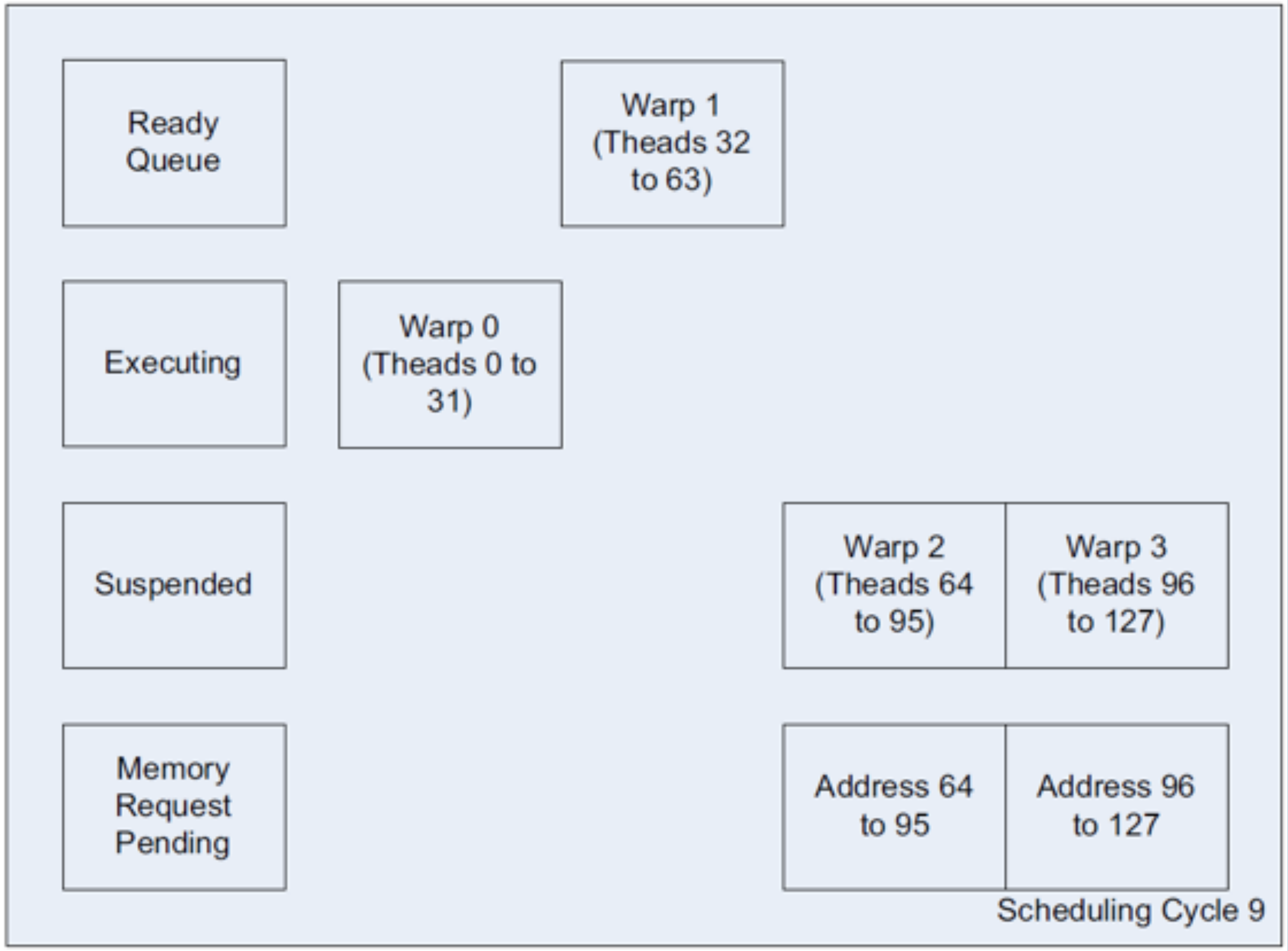
●Block Scheduling

✓ Execute in warps of 32 threads



CUDA Scheduling

- **Block Scheduling**
  - ✓ Execute in warps of 32 threads



Ready Queue

Warp 1 (Theads 32 to 63)

Executing

Warp 0 (Theads 0 to 31)

Suspended

Warp 2 (Theads 64 to 95)

Warp 3 (Theads 96 to 127)

Memory Request Pending

Address 64 to 95

Address 96 to 127

Scheduling Cycle 9

Lab 2.1 Warp Scheduling

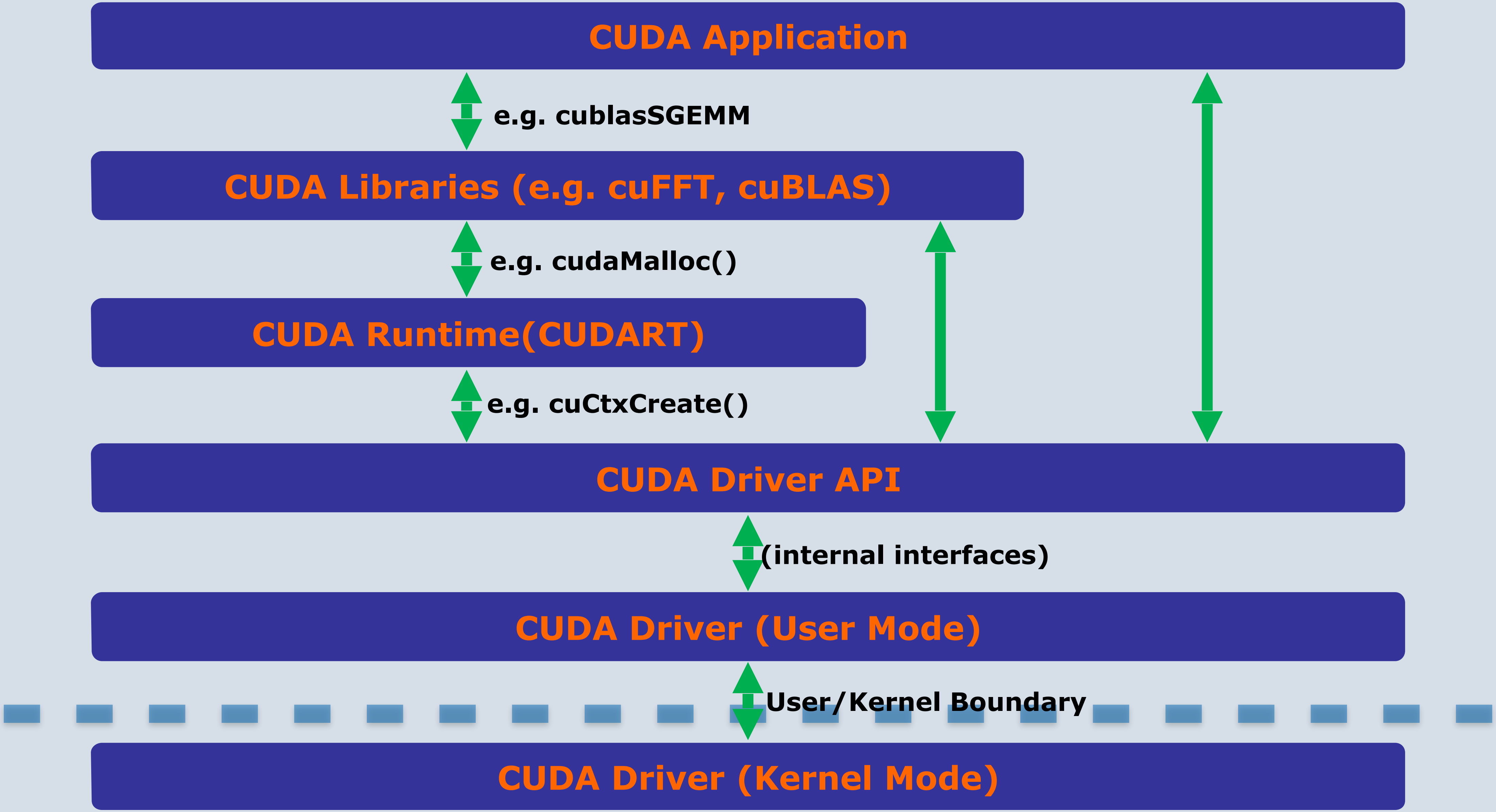
- 理解线程束的调度机制
  - ✓ 验证warp的线程数量
  - ✓ 加入计时功能，对warp的调度时间进行输出，并绘出散点图进行分析
  - ✓ 变大block和grid的大小会如何?
  - ✓ 给出对线程束调度机制的理解
  - ✓ 参见COOK 5.3 和WILT 7.3.3

```
__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;
}
```

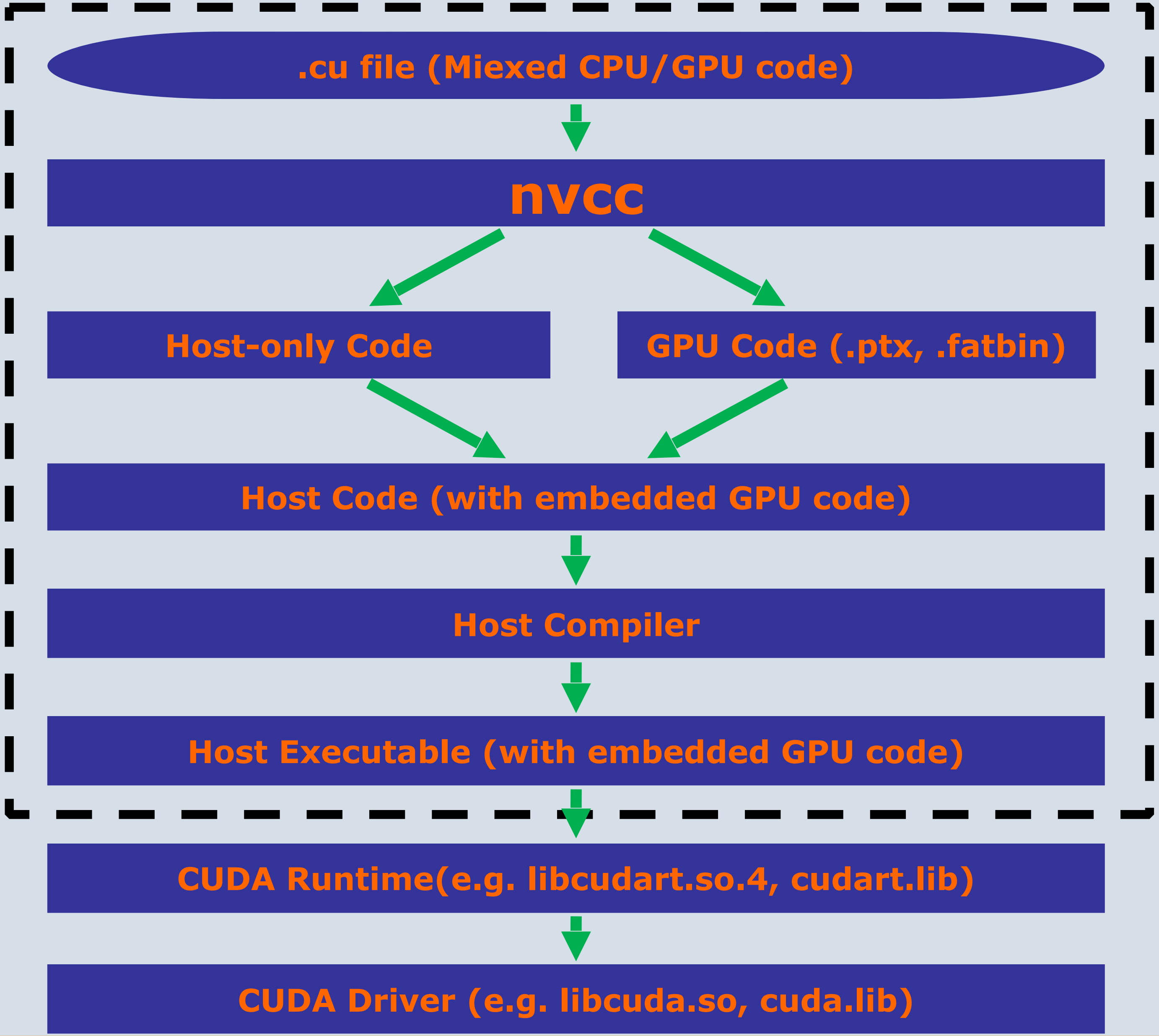
Outline

1. Multithreading
2. CUDA Abstraction
3. Kernel Execution
4. Warp Scheduling
5. CUDA Toolchain

Software Architecture



Compiling Process


.cu file (Miexed CPU/GPU code)
nvcc
Host-only Code
GPU Code (.ptx, .fatbin)
Host Code (with embedded GPU code)
Host Compiler
Host Executable (with embedded GPU code)
CUDA Runtime(e.g. libcudart.so.4, cudart.lib)
CUDA Driver (e.g. libcuda.so, cuda.lib)

Refer to: “CUDA Compiler Driver NVCC”, v7.5, 2015.

NVCC Case Study

- Useful Options
  - ✓ -g
  - ✓ -G
  - ✓ -lineinfo
  - ✓ -o outfile
  - ✓ -include xxx.h
  - ✓ -l yyy.lib
  - ✓ -arch sm_??

Refer to: "CUDA Compiler Driver NVCC", v7.5, 2015.

CUDA Software Ecosystem

- Numerical Analysis Tools
  - ✓ e.g. Matlab, ArrayFire, Mathematica, LabView, Jacket
- GPU-Accelerated Libraries
  - ✓ e.g. Thrust, cuDNN, cuFFT, cuBLAS
- Language and APIs
  - ✓ e.g. CUDA Toolkit, OpenACC, CUDA Fortran, PyCUDA, PGI compilers
- Performance Analysis Tools
  - ✓ e.g. Nsight, NVIDIA Visual Profiler,
- Debugging Solutions
  - ✓ e.g. Nsight, CUDA-GDB, CUDA-MEMCHECK

Quiz

- 请在命令行使用nvcc编译你的cuda程序
  - ✓ 在vs2013里，nvcc的调用在哪里，尝试解释其中的参数含义