Review on Lec 1 & Lec 2

Hello GPU

```
__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];
}
```

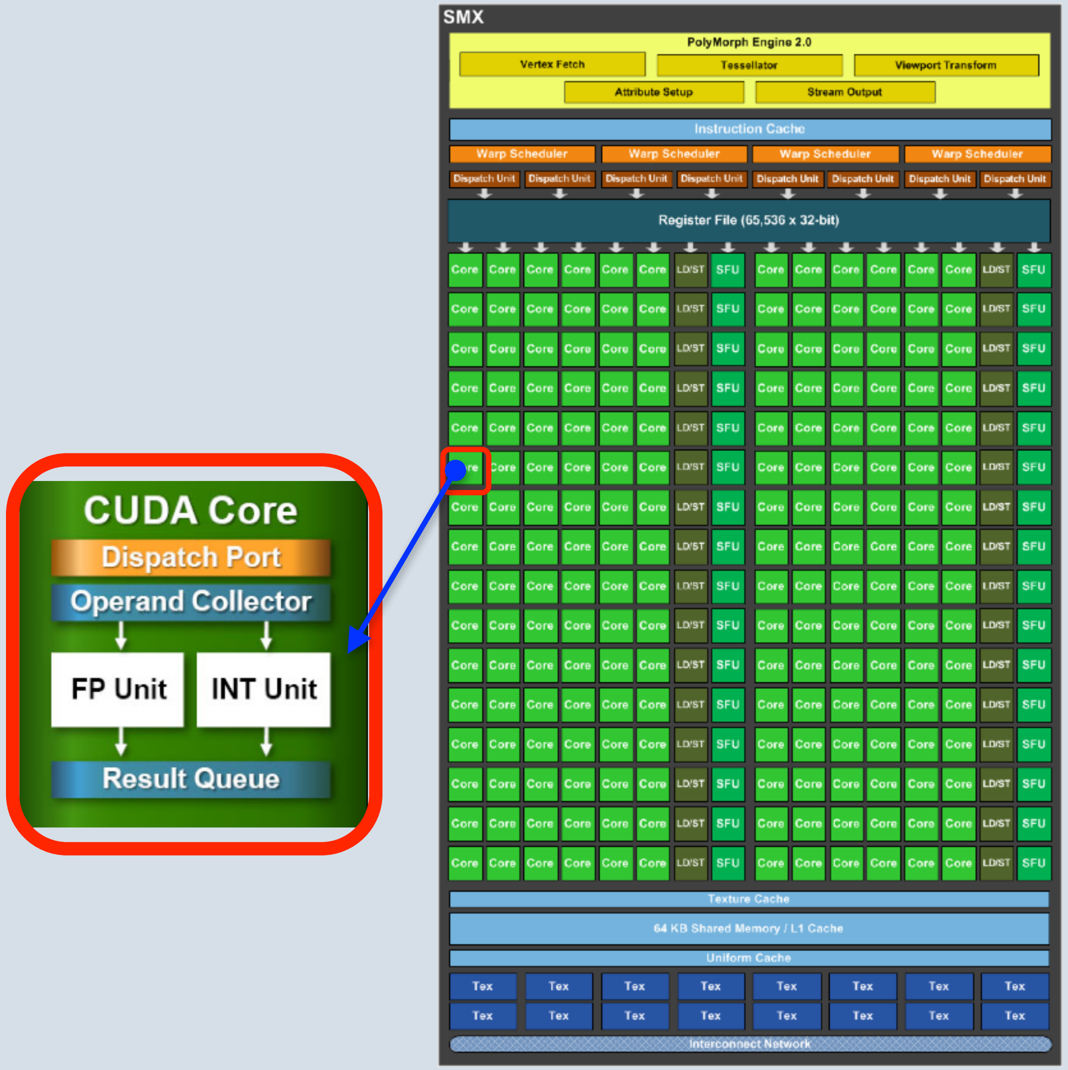
```
void main(){
         ......
         int *dev_a,*dev_b,*dev_c;
         cudaMalloc((void**)&dev_c, 128* sizeof(int));
         ......
         cudaMemcpy(dev_a, a, 128* sizeof(int), cudaMemcpyHostToDevice);
         cudaMemcpy(dev_b, b, 128* sizeof(int), cudaMemcpyHostToDevice);

         // 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

Tonghua Su
School of Software
Harbin Institute of Technology

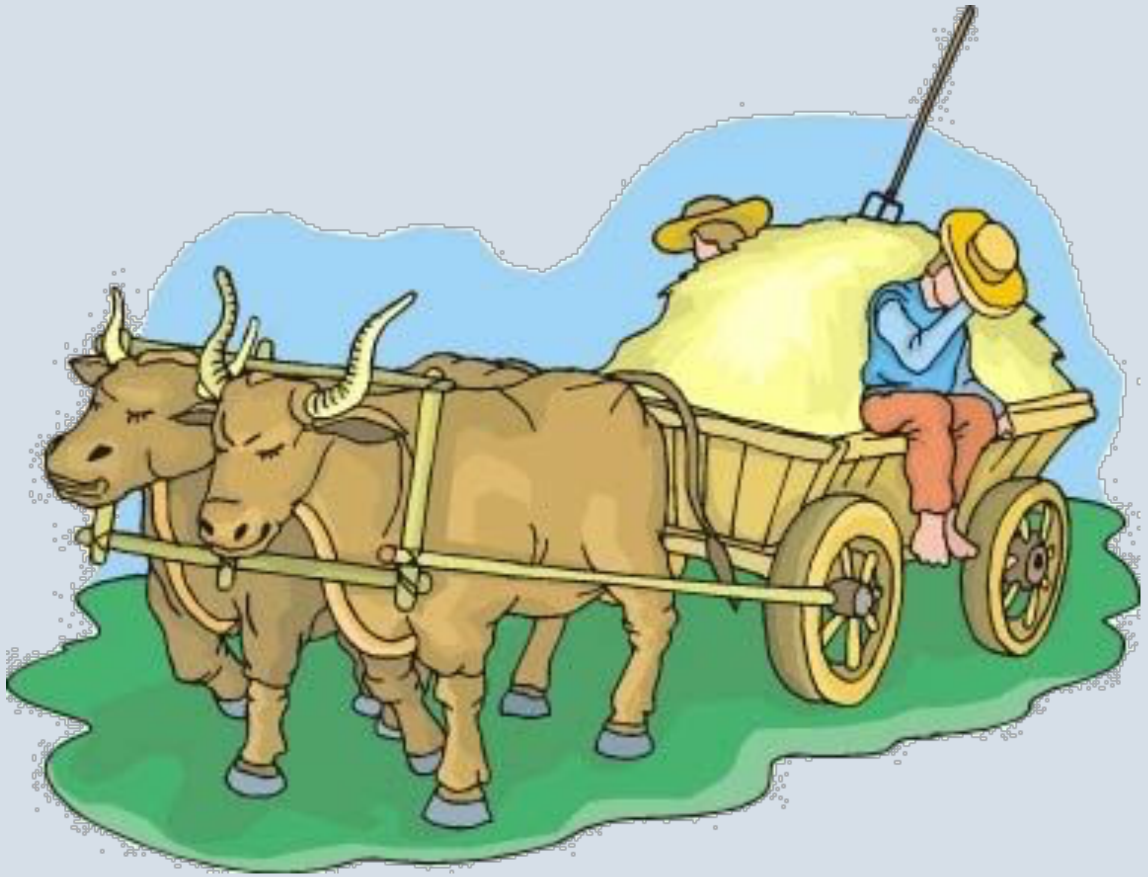
Outline

Outline

Design Philosophy

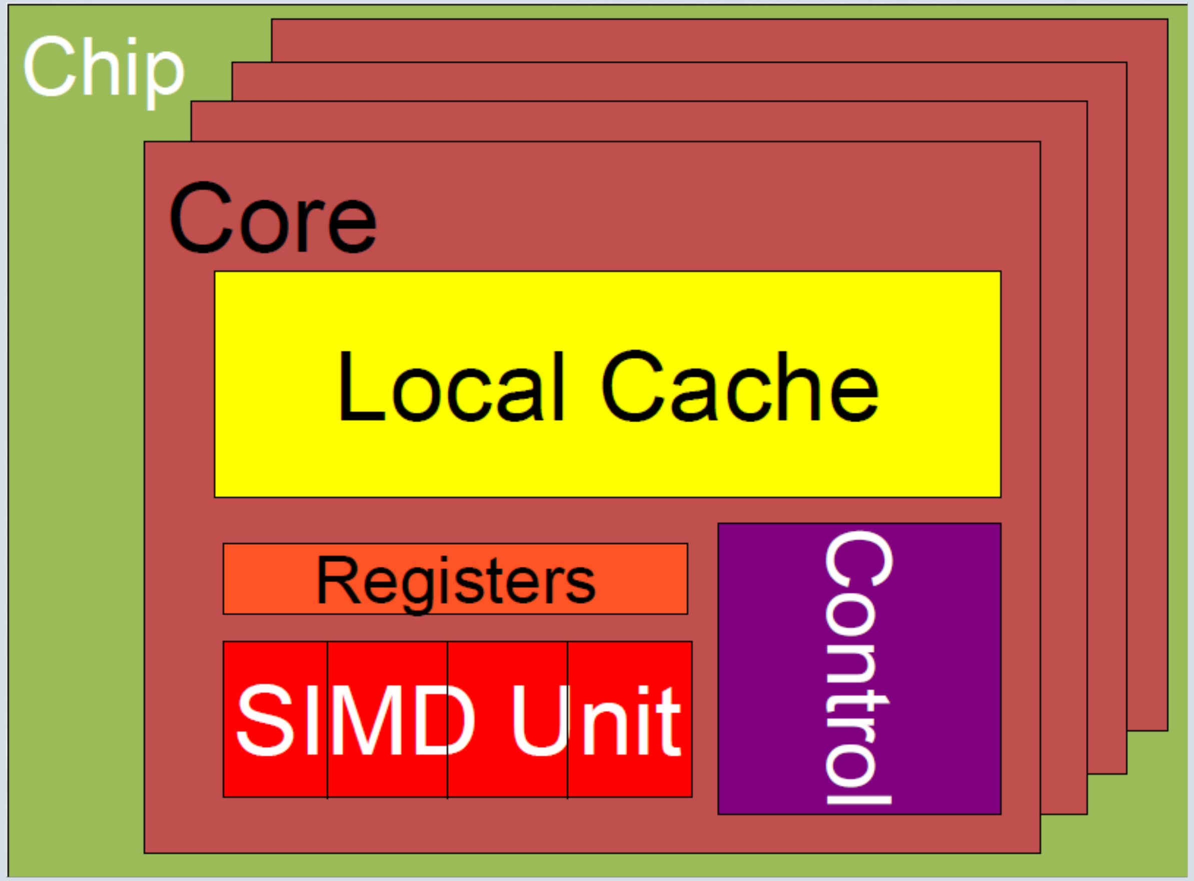
CPU: Latency Oriented Cores

GPU: Throughput Oriented Cores

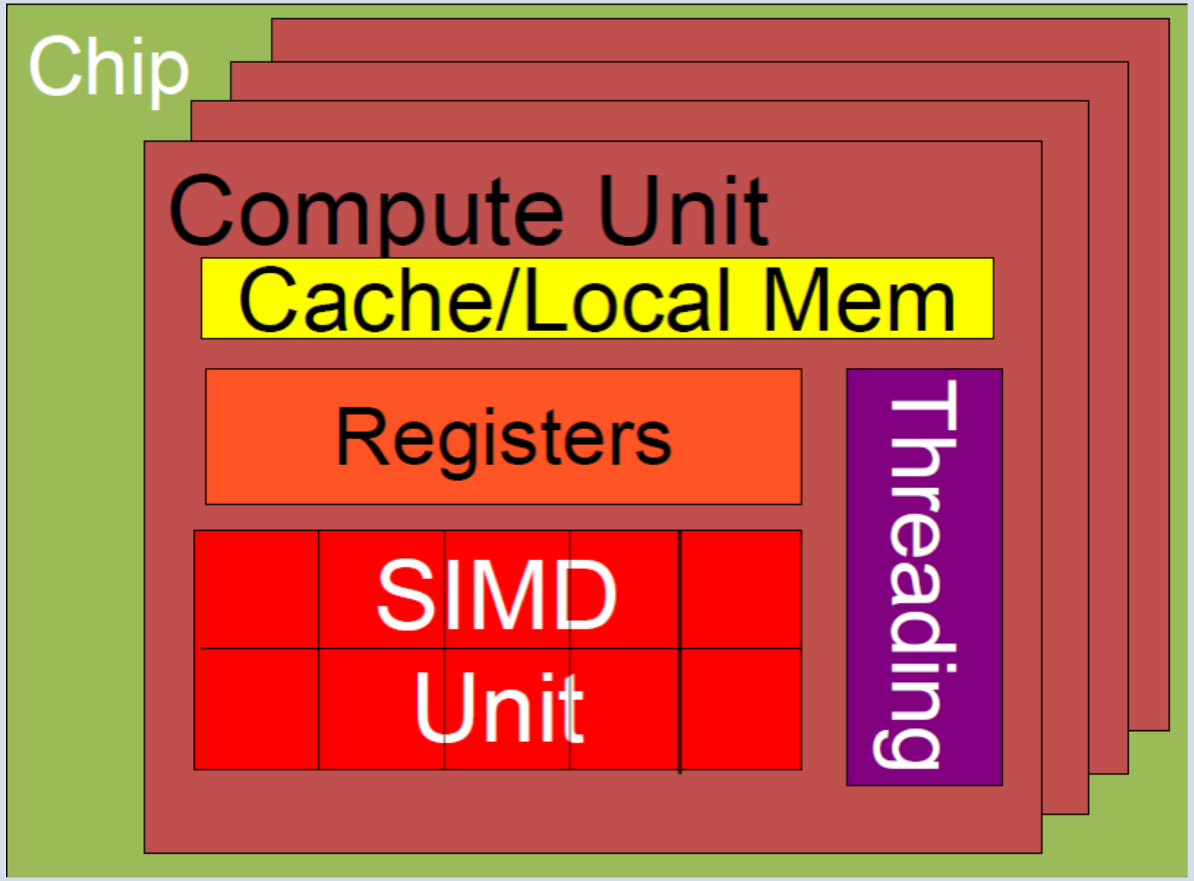


Design Philosophy

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**

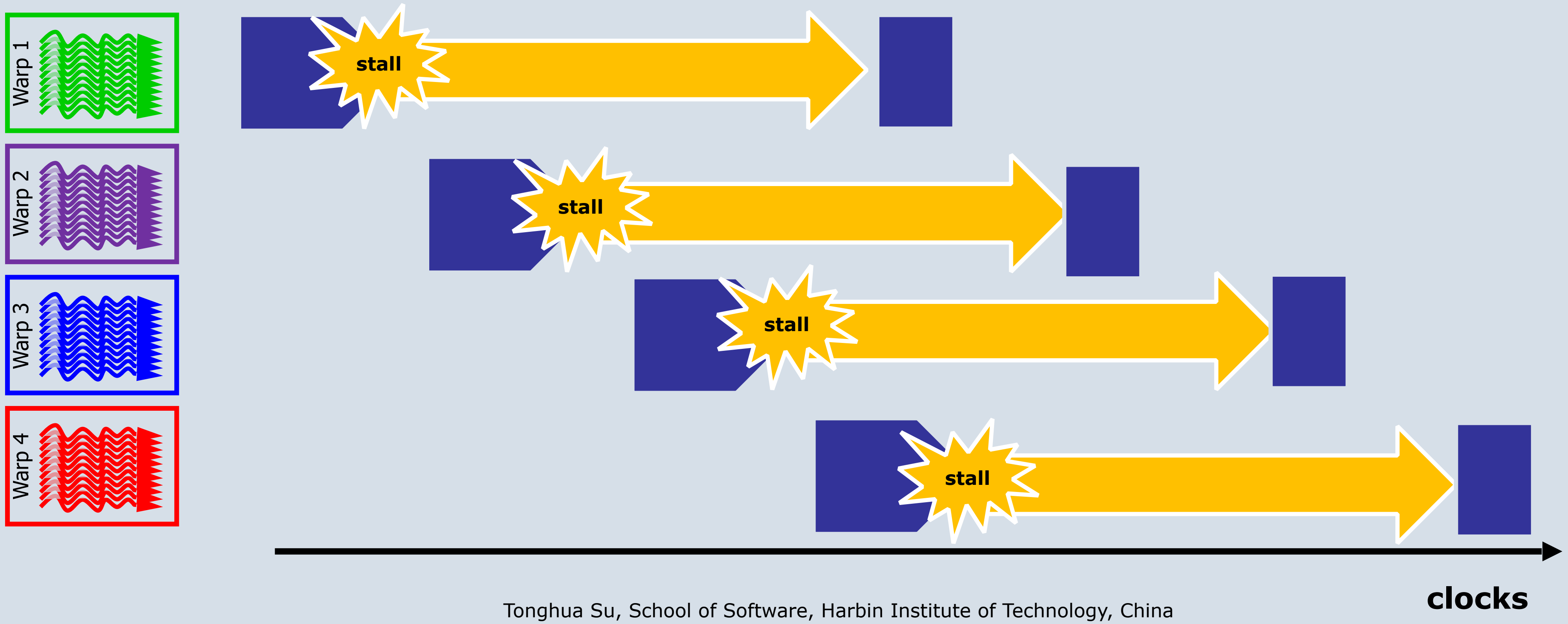
GPU Multithreading

- **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，那么另外需要多少个就绪的线程束才能掩盖停滞带来的时间缝隙?

作业预热

- 请编写程序，实现两个矩阵相乘
  - 先编写CPU版程序，然后给出GPU代码
  - 每个矩阵用线性数组表示
  - 考虑多个block
  - 考虑矩阵尺寸不是block尺寸的整倍数

Outline

CUDA

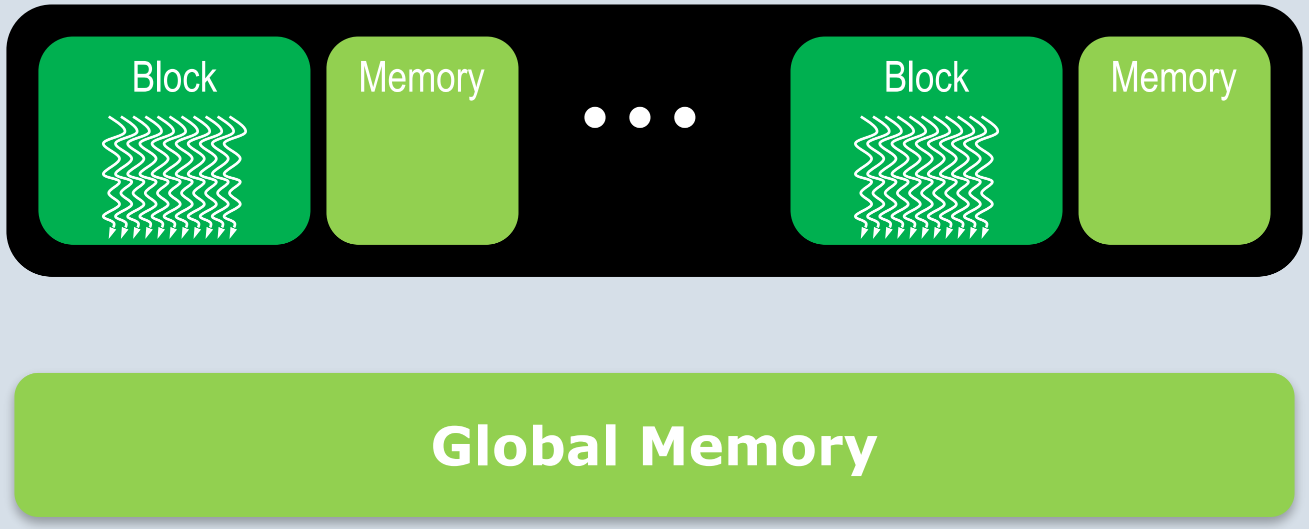
- **CUDA(Compute Unified Device Architecture) is developed by Nvidia around 2007**
  - ✓ **2-4 week learning curve for those with experience of OpenMP and MPI programming**
  - ✓ **large user community on NVIDIA forums**
- **CUDA is a parallel computing platform and programming model that enables dramatic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

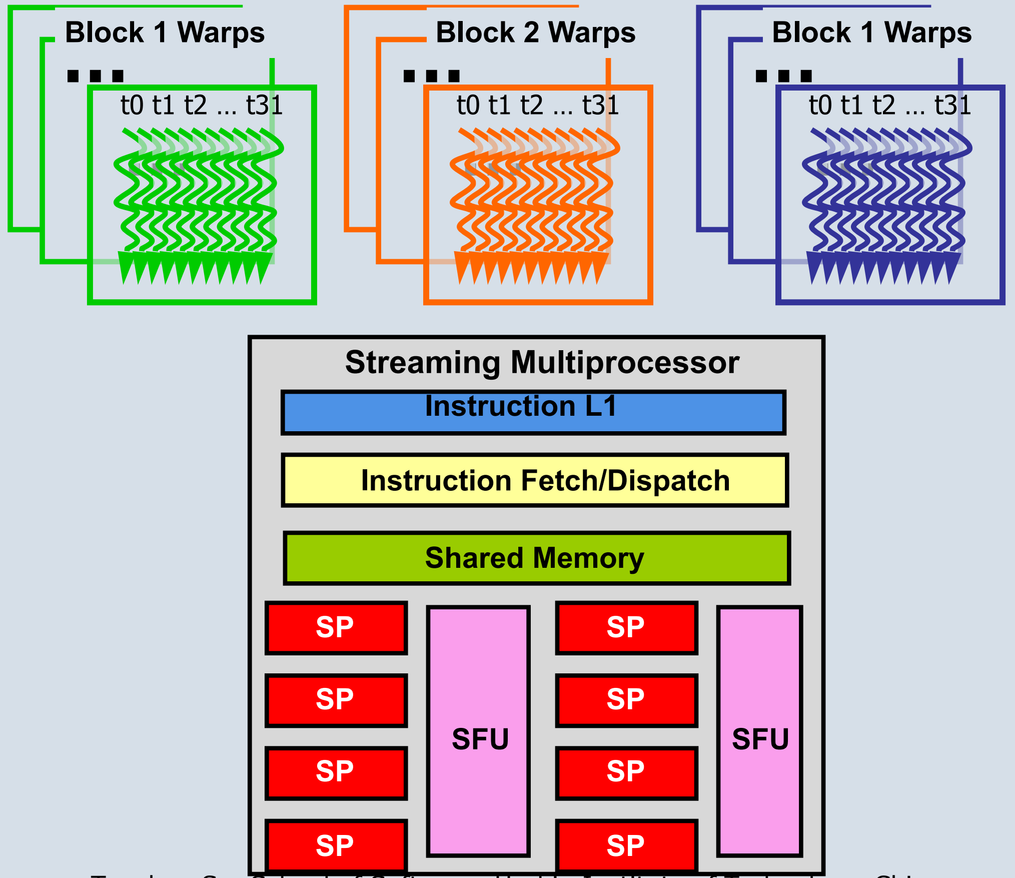
- **CUDA Virtualizes the Physical Hardware**
  - ✓ thread is a virtualized CUDA cores (registers, PC, state)
  - ✓ block is a virtualized streaming multiprocessor (threads, shared mem.)
- **Scheduled onto Physical Hardware without Pre-emption**
  - ✓ threads/blocks launch & run to completion/suspension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



CUDA Abstraction

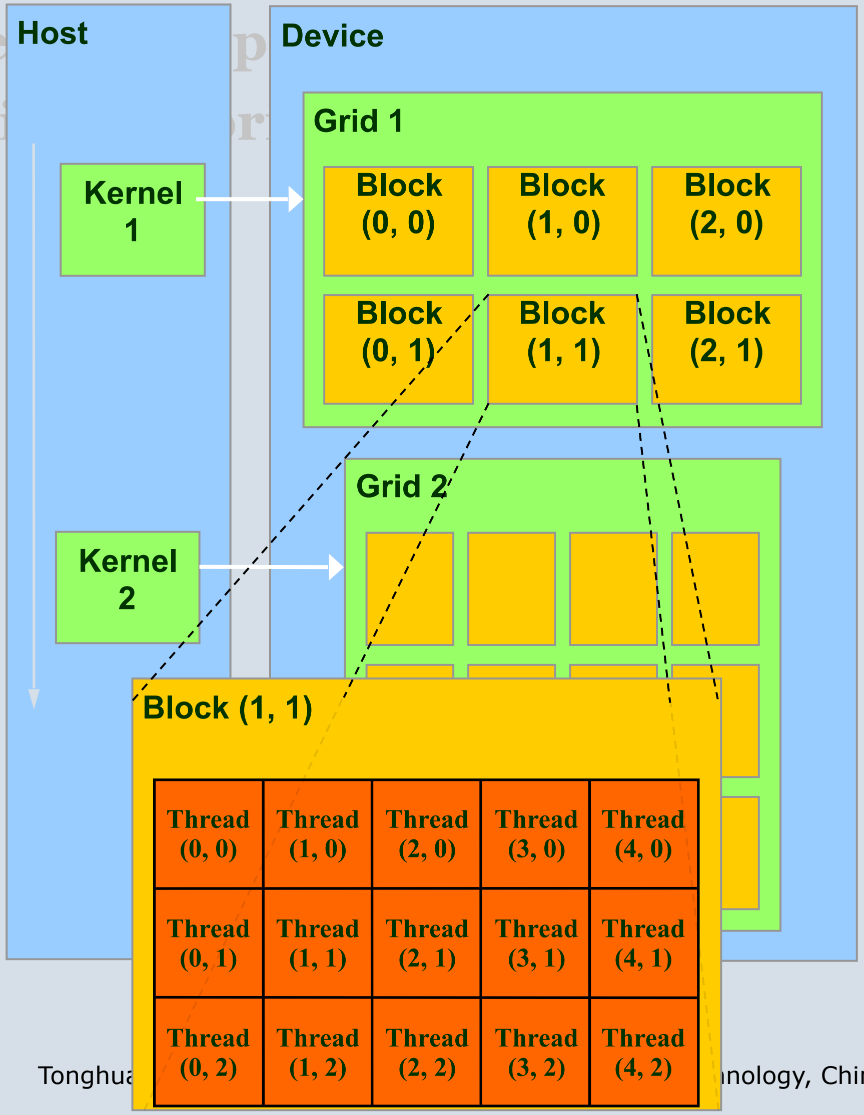
●Key Parallel Abstractions in CUDA

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



CUDA Abstraction

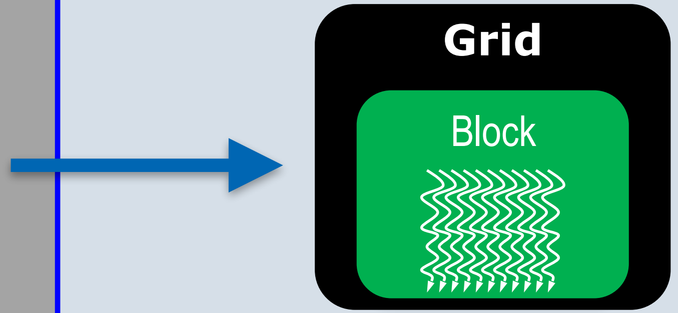
- **Key Parallel Abstractions in CUDA**
  - ✓ **Hierarchy of concurrent threads**
  - ✓ Shared memory mode
  - ✓ Lightweight synchroni



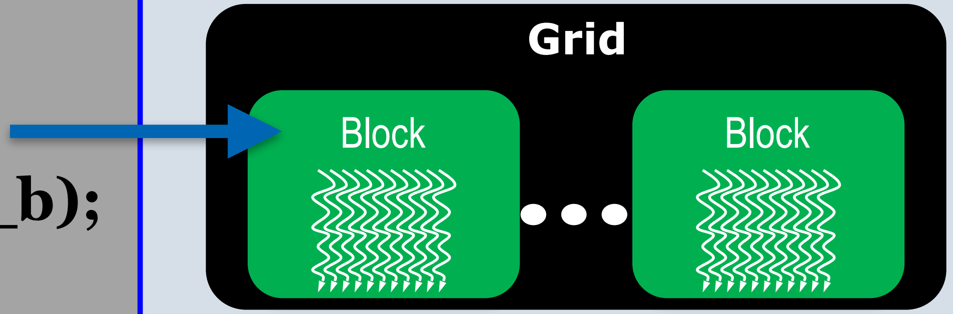
Thread Hierarchy

- **Thread —> Block—> Grid**

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

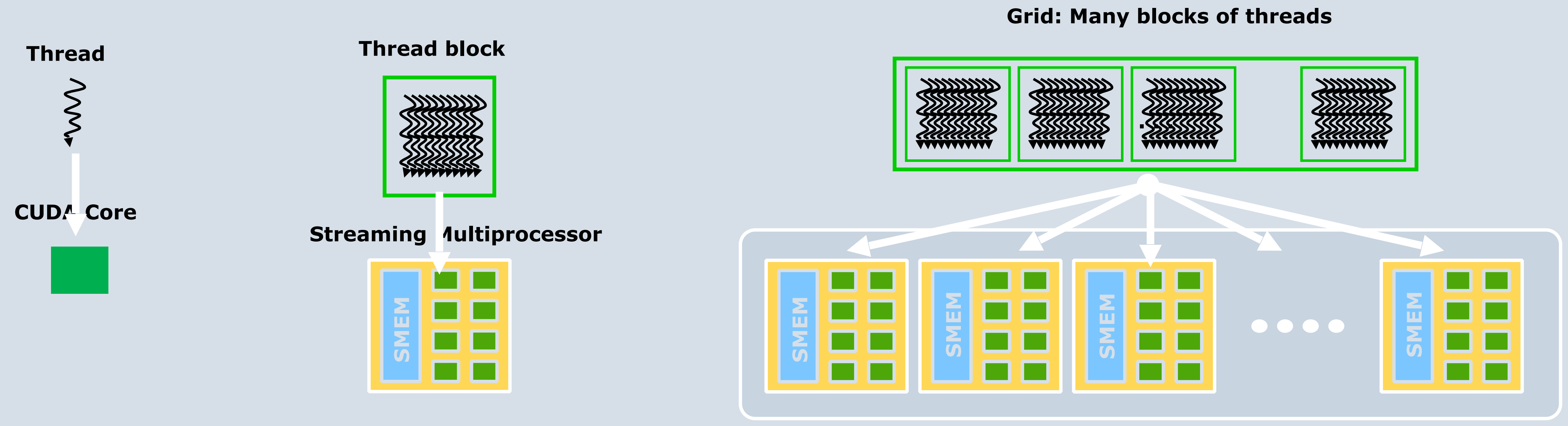


```
void main(){
    ......
    int *dev_a,*dev_b,*dev_c;
    ......
    addKernel<<<100, 128>>>(dev_c, dev_a, dev_b);
    ......
}
```

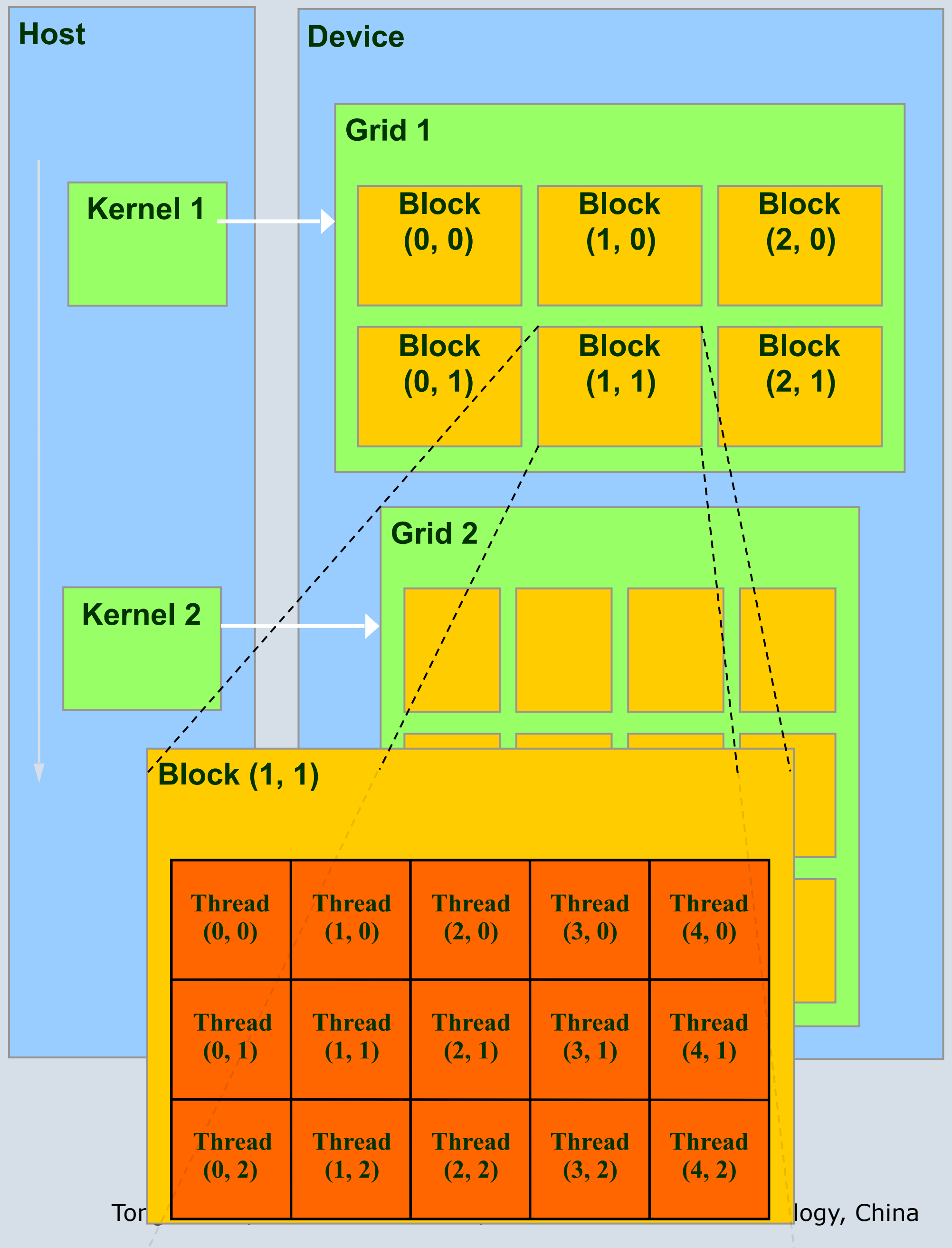


Thread Hierarchy

- **Thread Mapping**



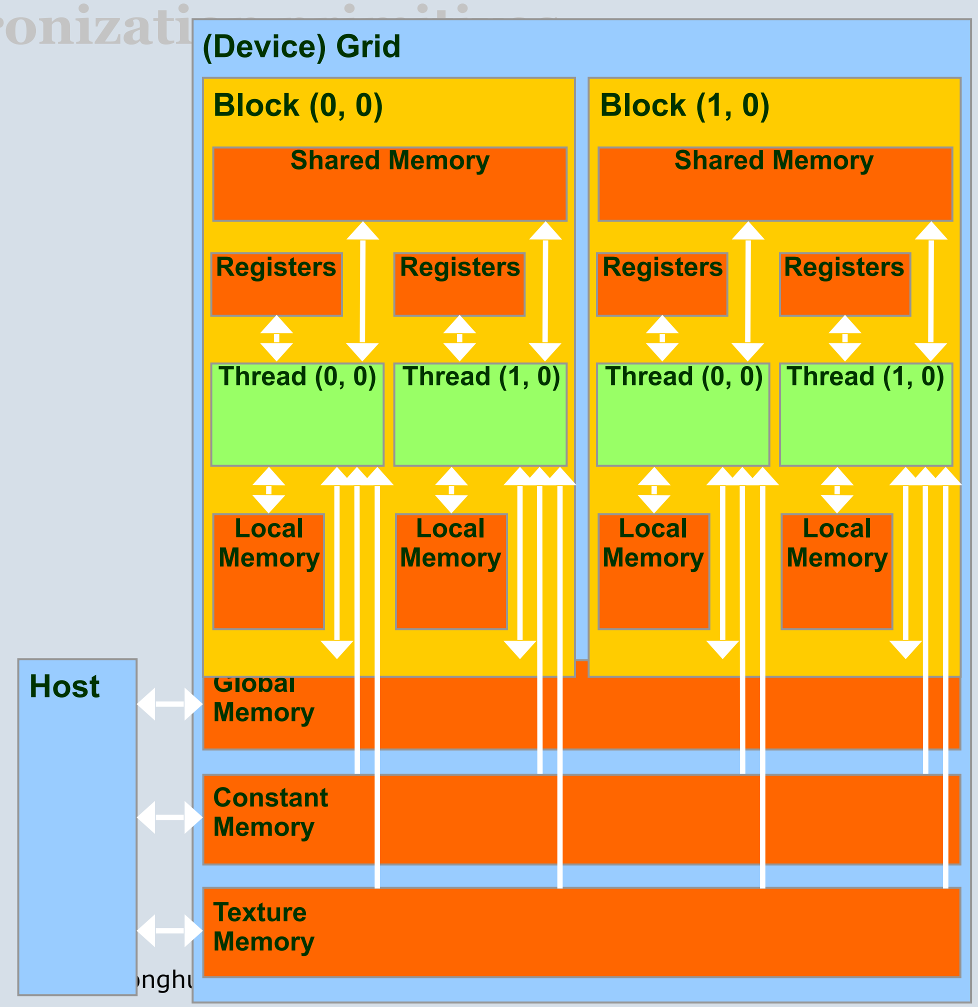
Thread Hierarchy



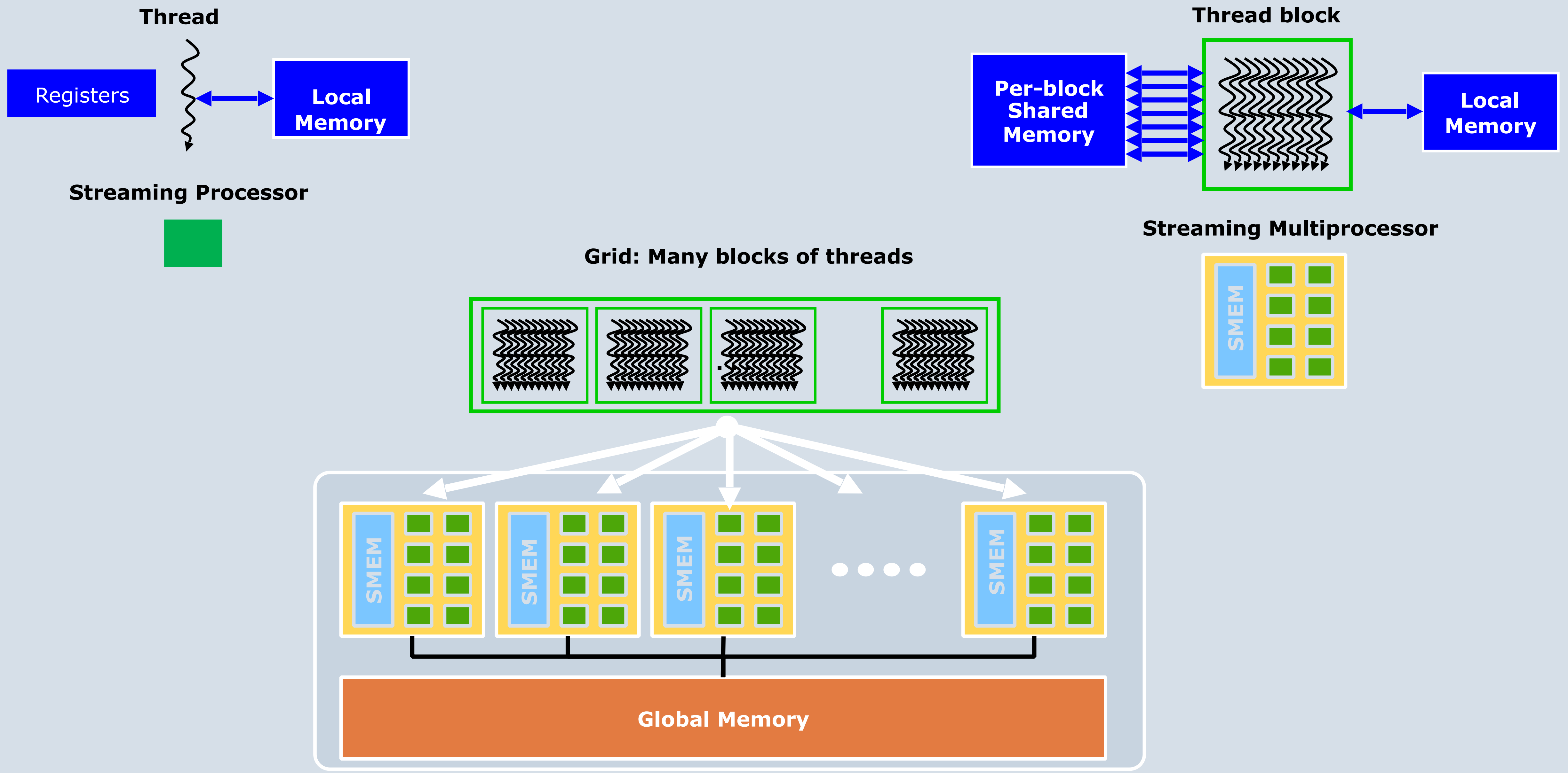
CUDA Abstraction

•Key Parallel Abstractions in CUDA

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

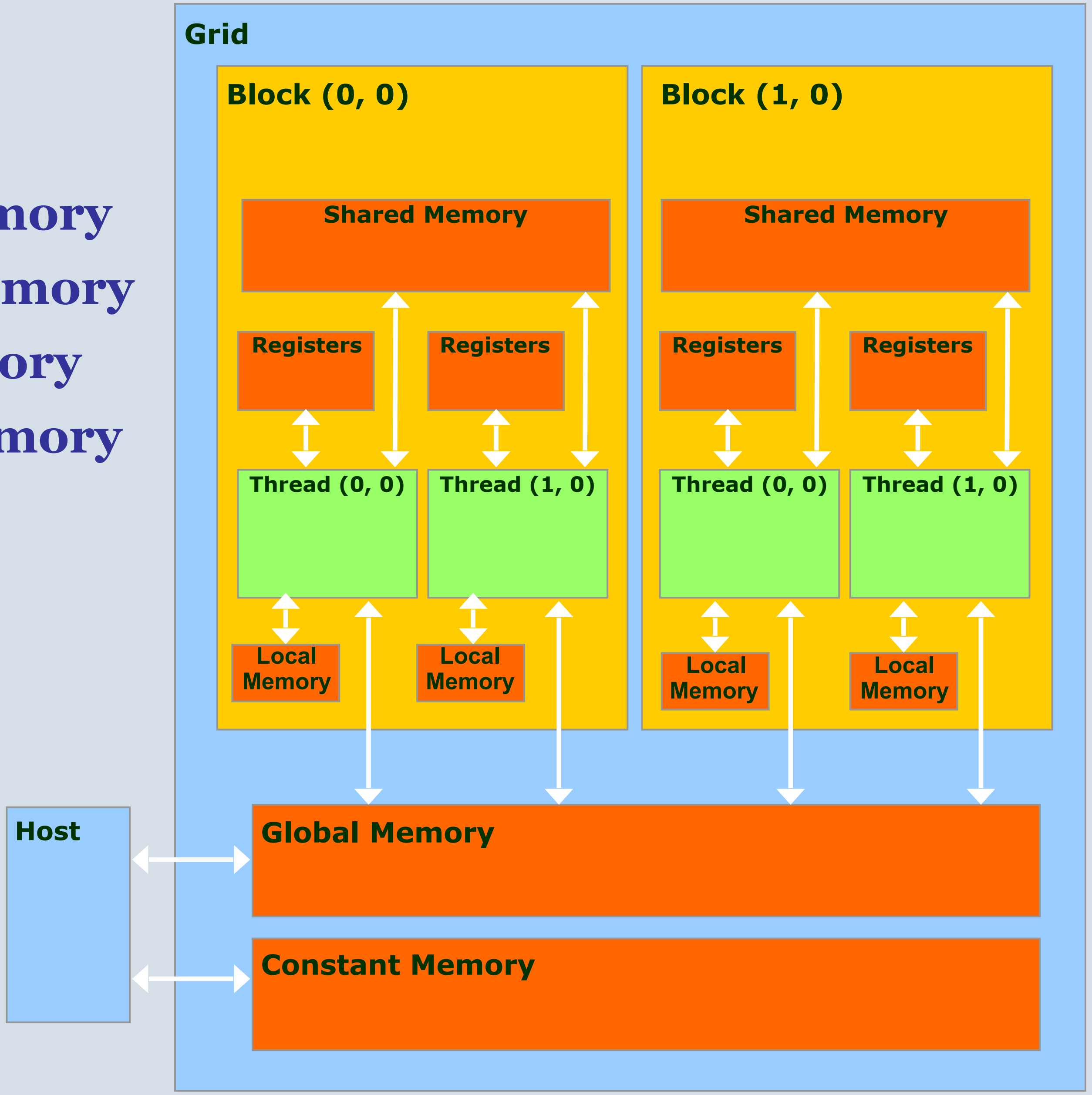


Memory Model



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



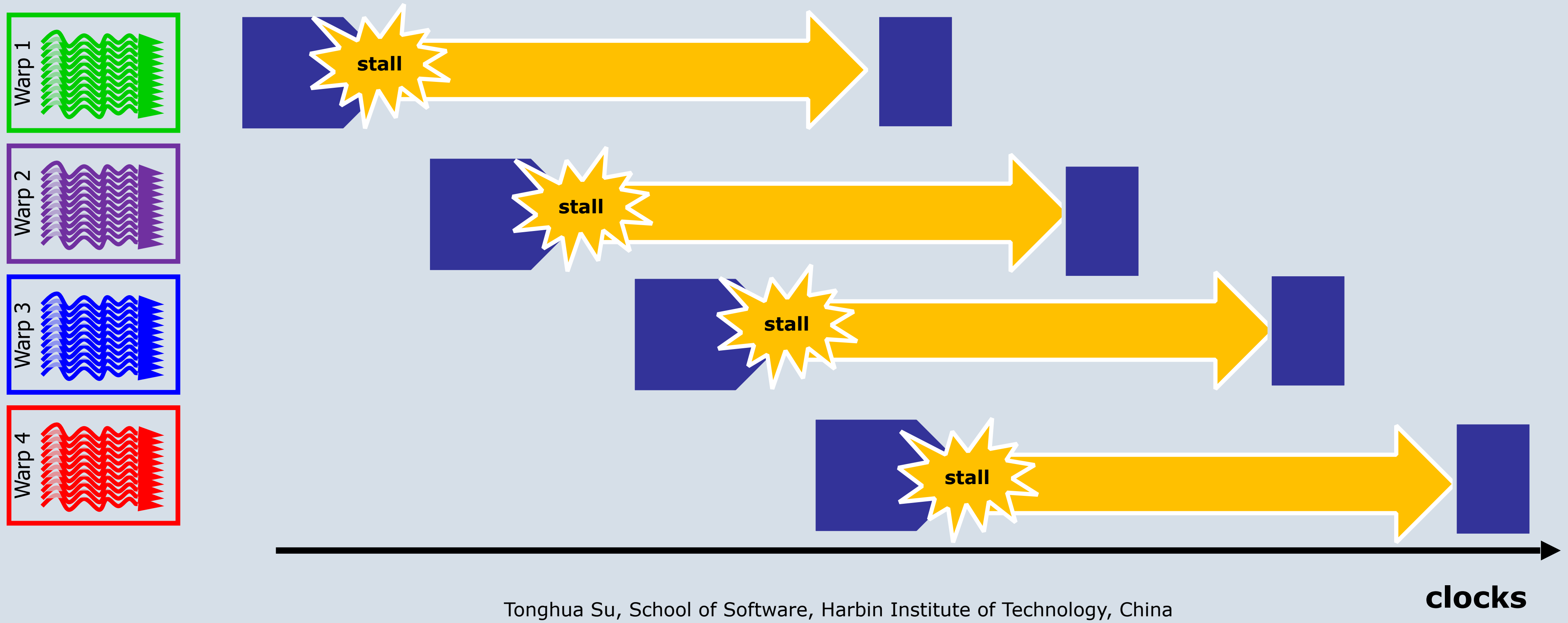
GPU Multithreading Review

- **Multithreaded GPU**
  - ✓ **CUDA core is multithreaded processor**
  - ✓ **Run in group of 32 threads (called a warp)**
  - ✓ **Zero-cost "context switching"**
    - Each thread has its own registers
  - ✓ **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**

Multithreading Review

•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

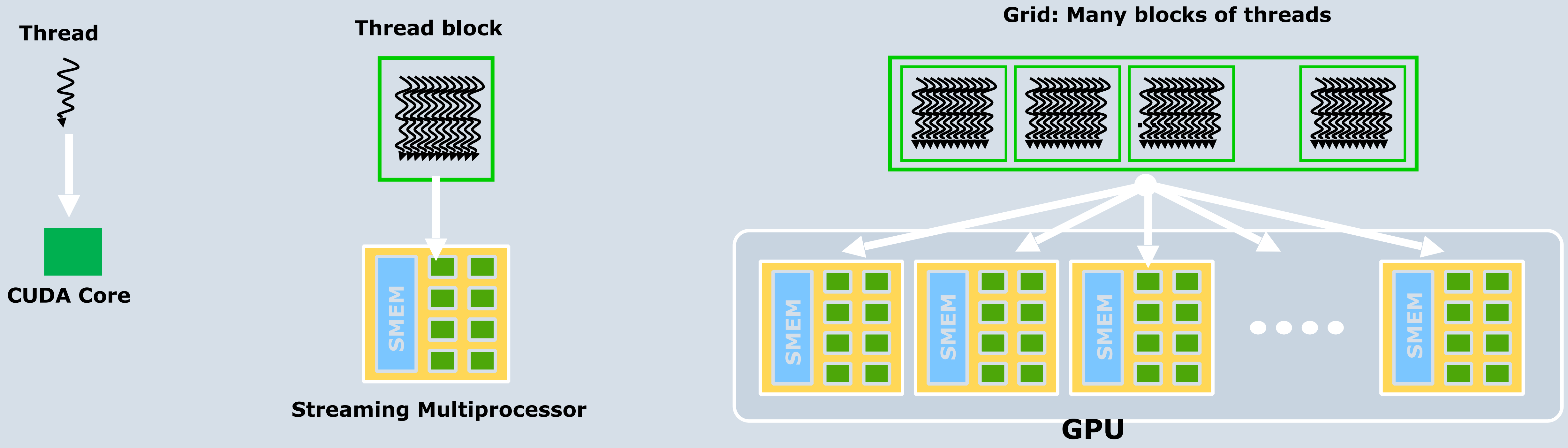


CUDA Abstractions Review

- CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the GPU
- Key Parallel Abstractions in CUDA
  - ✓ Hierarchy of concurrent threads
  - ✓ Shared memory model for cooperating threads
  - ✓ Lightweight synchronization primitives

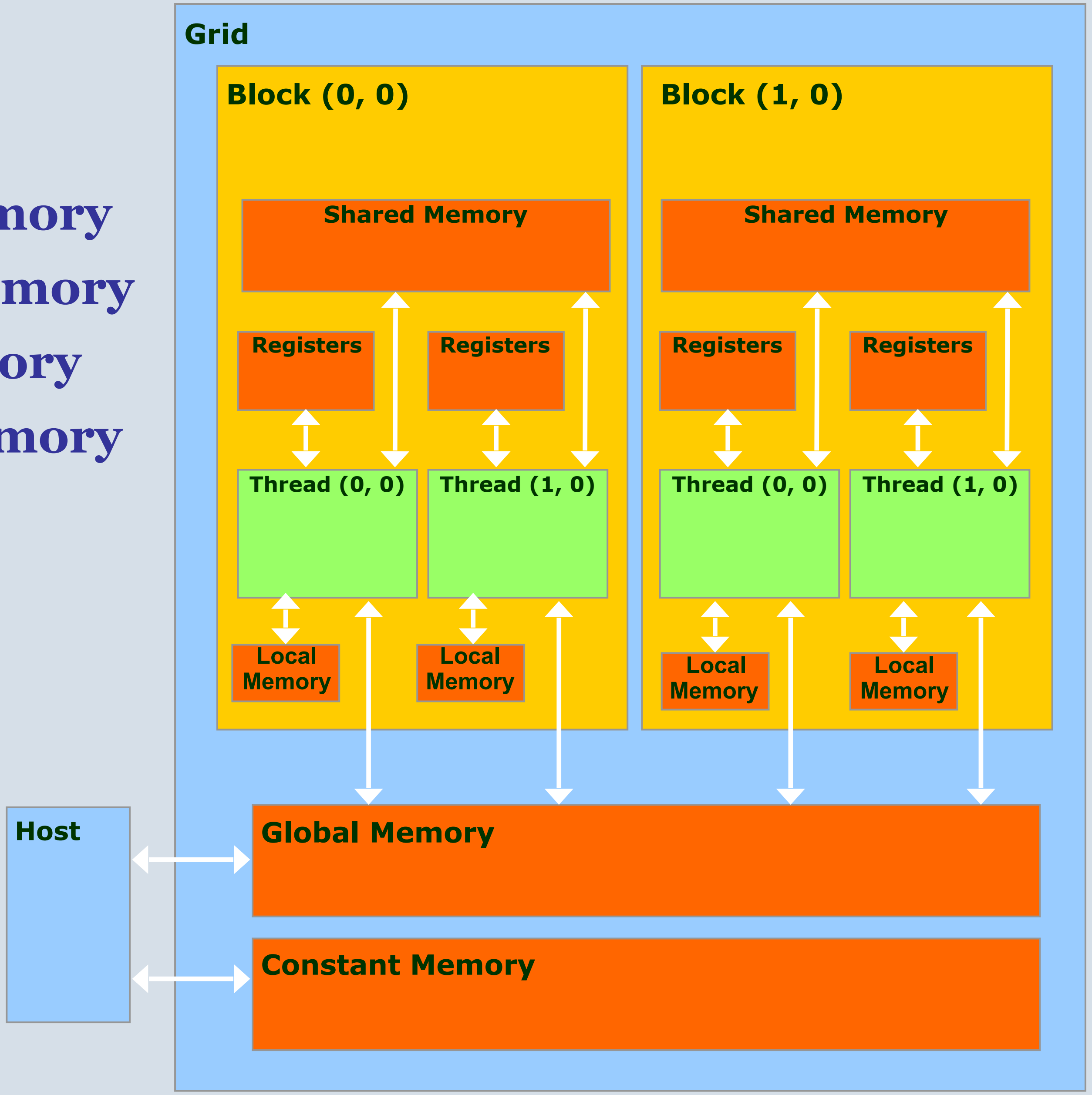
CUDA Abstractions Review

- **Thread Hierarchy and Thread Mapping**



CUDA Abstractions Review

- **Memory Model**
  - ✓ 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()**

```
__device__  unsigned int count = 0;
__shared__  bool isLastBlockDone;
```

```
__global__ void sum(const float* array, unsigned int N, volatile float* result)
{
    // Each block sums a subset of the input array.
    float partialSum = calculatePartialSum(array, N);
    if (threadIdx.x == 0) {
        // Thread 0 of each block stores the partial sum to global memory.
        result[blockIdx.x] = partialSum;
        // Thread 0 makes sure the partial sum has been written to global memory.
        __threadfence();
        // Thread 0 signals that it is done.
        unsigned int value = atomicInc(&count, gridDim.x);
        // Thread 0 determines if its block is the last block to be done.
        isLastBlockDone = (value == (gridDim.x - 1));
    }
    // make sure that each thread reads the correct value of isLastBlockDone.
    __syncthreads();
    if (isLastBlockDone) {
        // The last block sums the partial sums stored in result[0 .. gridDim.x-1]
        float totalSum = calculateTotalSum(result);
        if (threadIdx.x == 0) {
            result[0] = totalSum;
            count = 0;}
    }
}
```

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

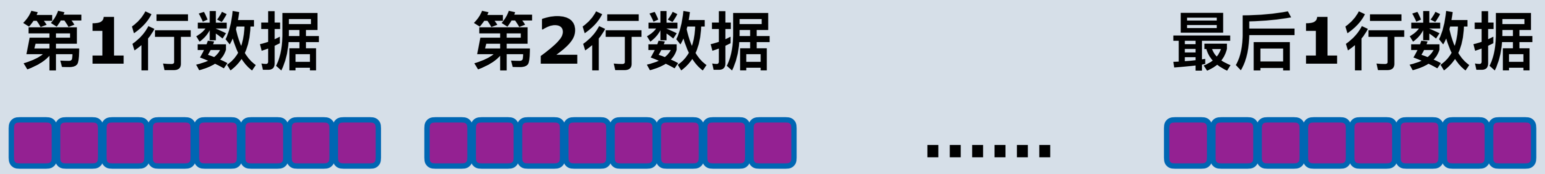
Synchronization

- **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!

  - ✓ e.g. int atomicSub(int* address, int val);

Quiz

- 对输入的一张灰度图，要统计其灰度直方图，请编写CPU版代码
  - 可以通过OpenCV载入灰度图片，也可以用unsigned char数组gray模拟
  - 请采用线性内存存储数据
  - 暂且假定图片的长乘宽不大于1024



Quiz

- 对输入的一张灰度图，要统计其灰度直方图，请编写可以完成此功能的GPU程序
  - 使用原子操作进行替换
  - 与CPU版代码结果对比

```
// Determine frequency of gray level in a picture
// gray have already been converted into unsigned char
// Each thread looks at one pixel and increments
// a counter atomically
__global__ void histogram(unsigned char* gray,
                          int* buckets)
{
  int i = threadIdx.x;
  unsigned char c = gray[i];
  buckets[c]++;
}
```