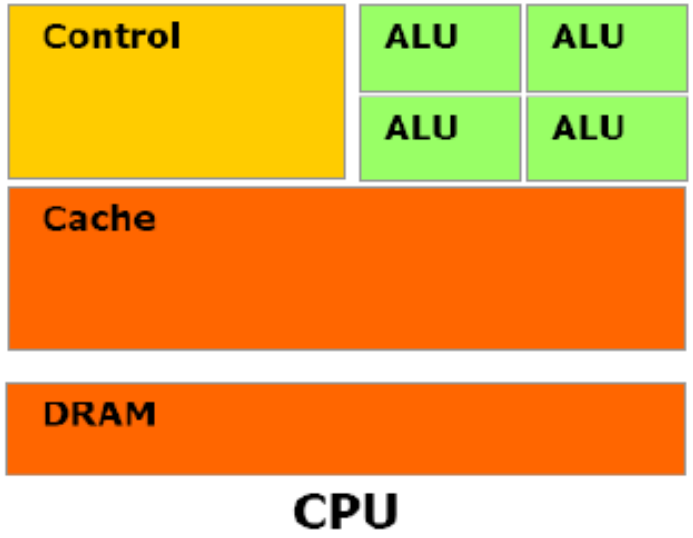
GPU环境下并行程序设计

目录

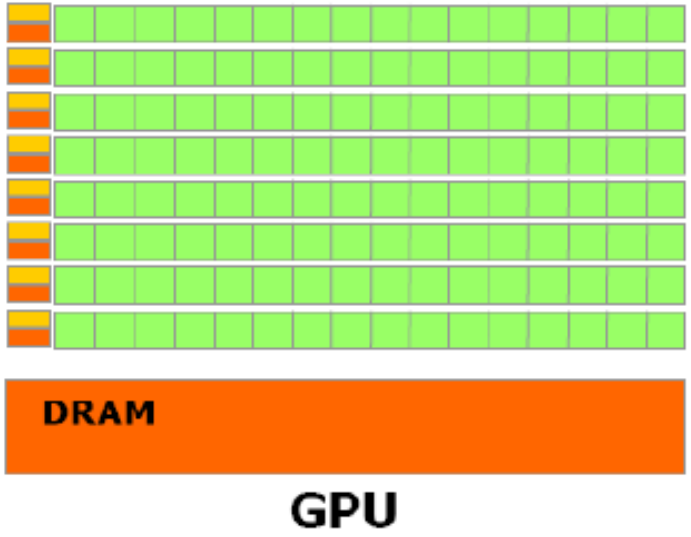
CUDA简介

GPU

- GPU *VS* CPU: 硬件体系



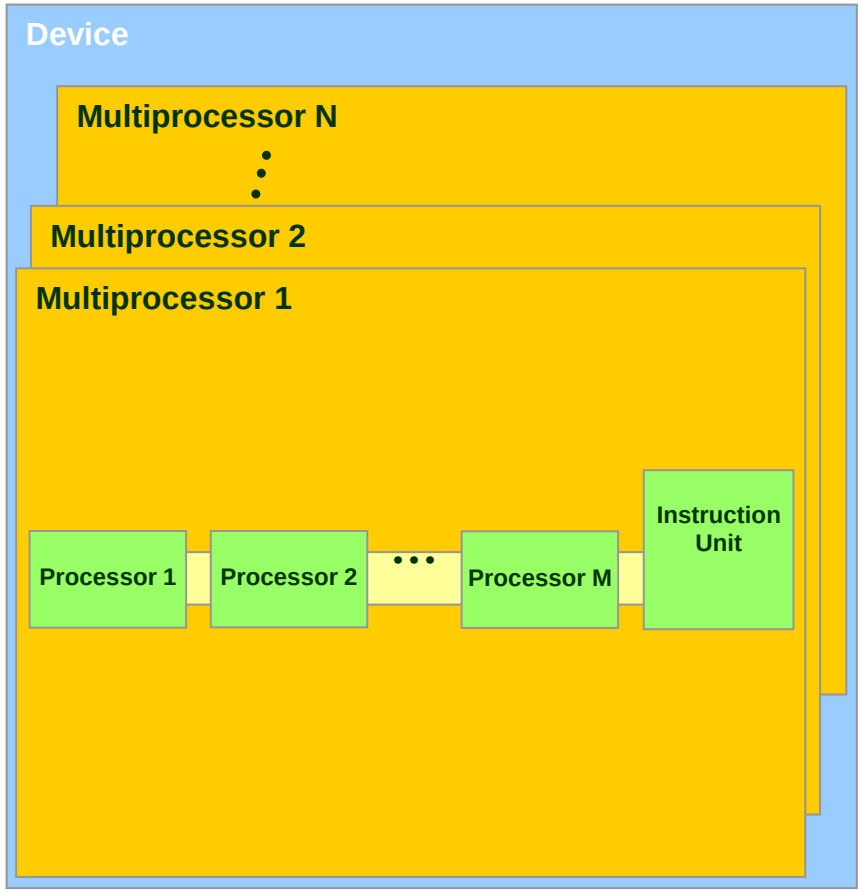
- **目标: 最小的指令延迟**
- **巨大的缓存**
- **复杂的控制逻辑**



- **目标: 最大吞吐量**
- **更多寄存器、高带宽**
- **通过线程轮换隐藏延迟**
- **多个线程共享控制逻辑**

GPU: A Set of SIMD Multiprocessors

- 一设备有一组multiprocessors
- 每一multiprocessor由一组具有SIMD体系结构的处理器构成
  - Shared instruction unit
- 在每一时钟周期，multiprocessor在一组线程上执行同样的指令。



The NVIDIA Kepler

NVIDIA Kepler GK110 Whitepaper

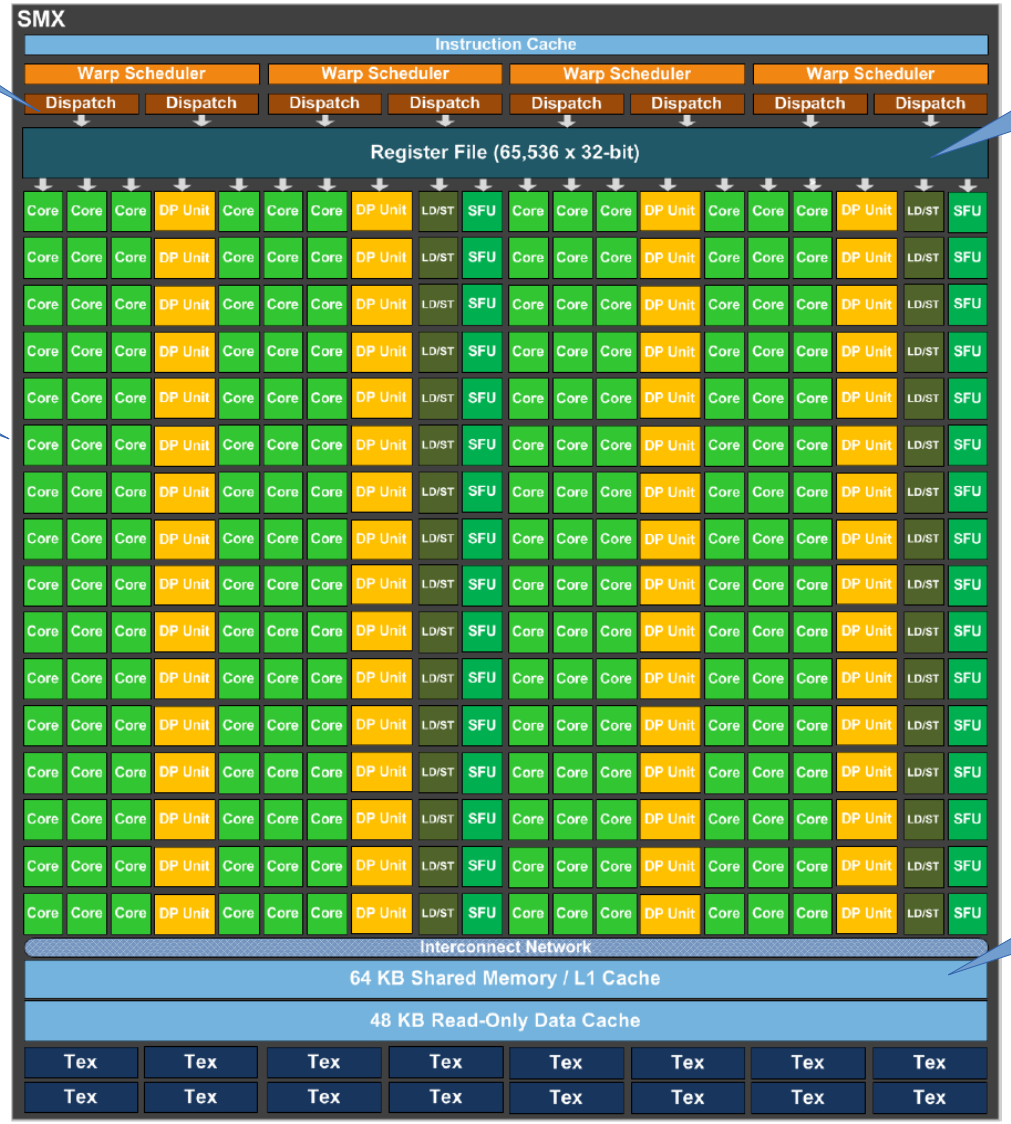
NVIDIA Kepler SMX

2-way
In-order

192 FP/Int
64 DP
32 LD/ST

256KB!

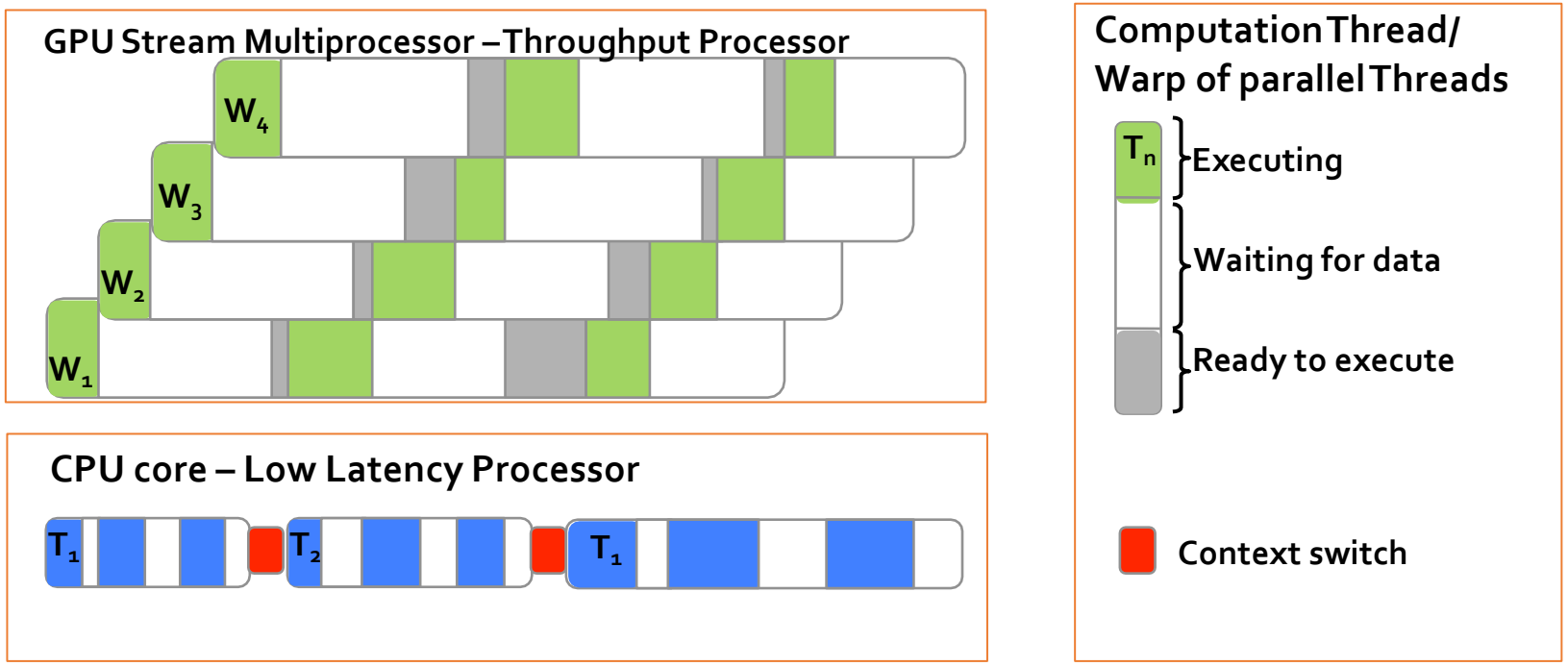
Partitioned
(user-defined)



NVIDIA Kepler GK110 Whitepaper

多线程

- CPU体系结构努力减少每一线程内的延迟
- GPU architecture hides latency with computation from other thread warps



GPU计算

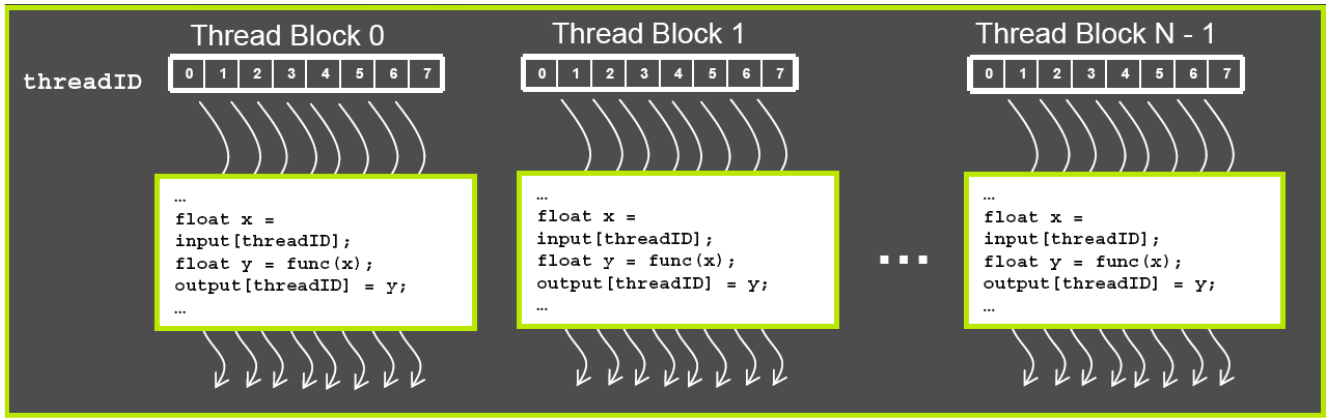
- The GPU is a highly parallel compute device
  - serves as a coprocessor for the host CPU
  - has its own device memory on the card
  - executes many threads in parallel
- Parallel kernels run a single program in many threads
- GPU threads are extremely lightweight
  - Thread creation and context switching are essentially free

CUDA

- CUDA– Compute Unified Device Architecture
  - 由NVIDIA 2007年推出的通用并行计算架构
  - 在采用了统一架构的GPU上运行
  - NVIDIA提供CUDA开发工具包和SDK
- Programing system for machines with GPUs
  - Programming Language
  - Compilers
  - Runtime Environments
  - Drivers
  - Hardware

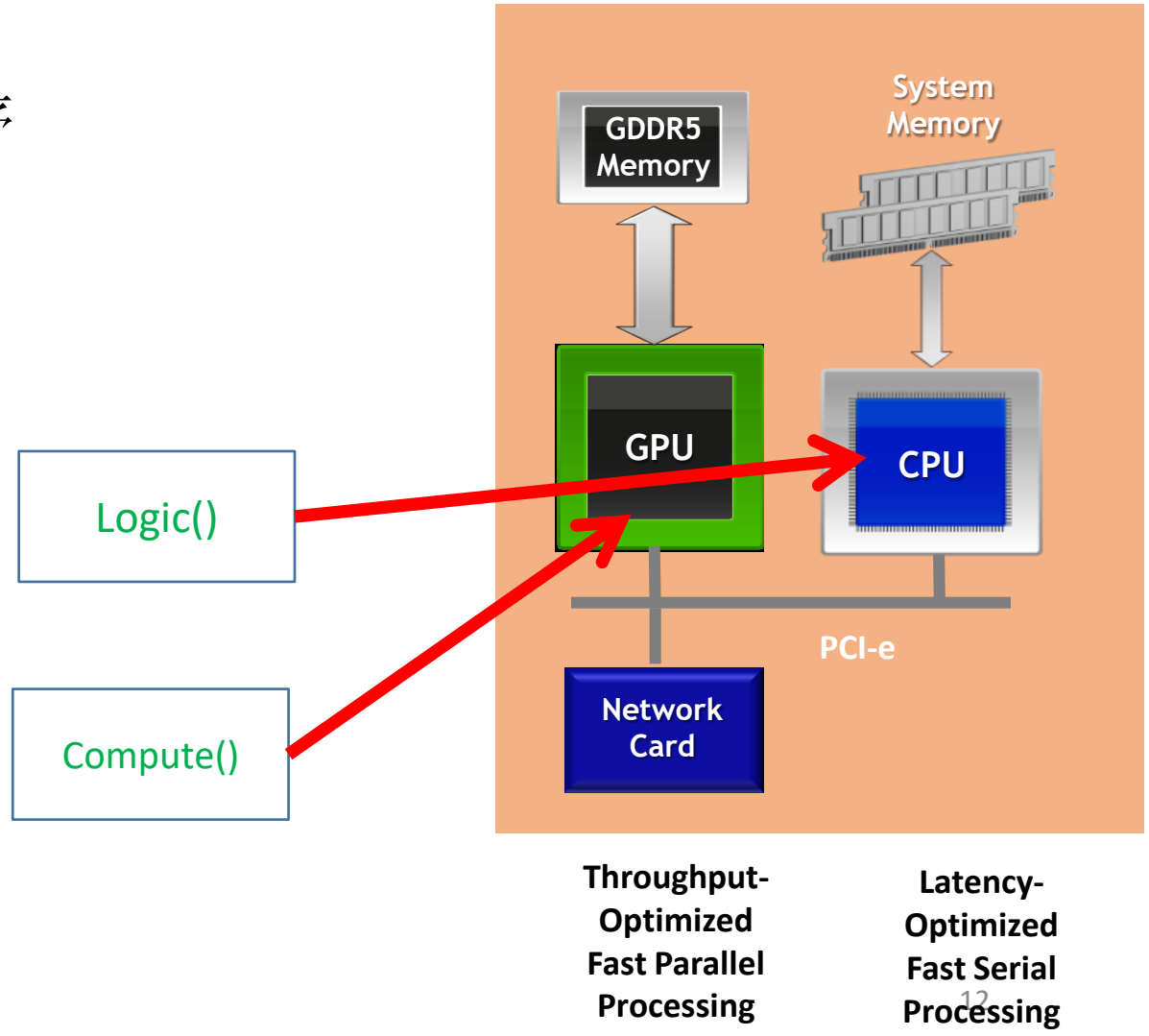
CUDA

- 通用并行计算模型
  - 单指令、多数据执行模式 (SIMD)
    - 所有线程执行同一段代码
    - 大量并行计算资源处理不同数据
  - 隐藏存储器延时
    - 提升计算 / 通信比例
    - 合并相邻地址的内存访问
    - 快速线程切换1 cycle@GPU vs. ~1000 cycles@CPU



CUDA

- 异构计算模型: 集成CPU + GPU应用程序
  - CPU: 顺序执行代码
  - GPU = 超大规模数据并行协处理器
    - “批发”式执行大量细粒度线程



GPU Programming

Fortran	OpenACC, CUDA Fortran
C	OpenACC, CUDA C
C++	CUDA C++, Thrust, Hemi, ArrayFire
Python	Anaconda Accelerate, PyCUDA, Copperhead
.NET	CUDAfy.NET, Alea.cuBase

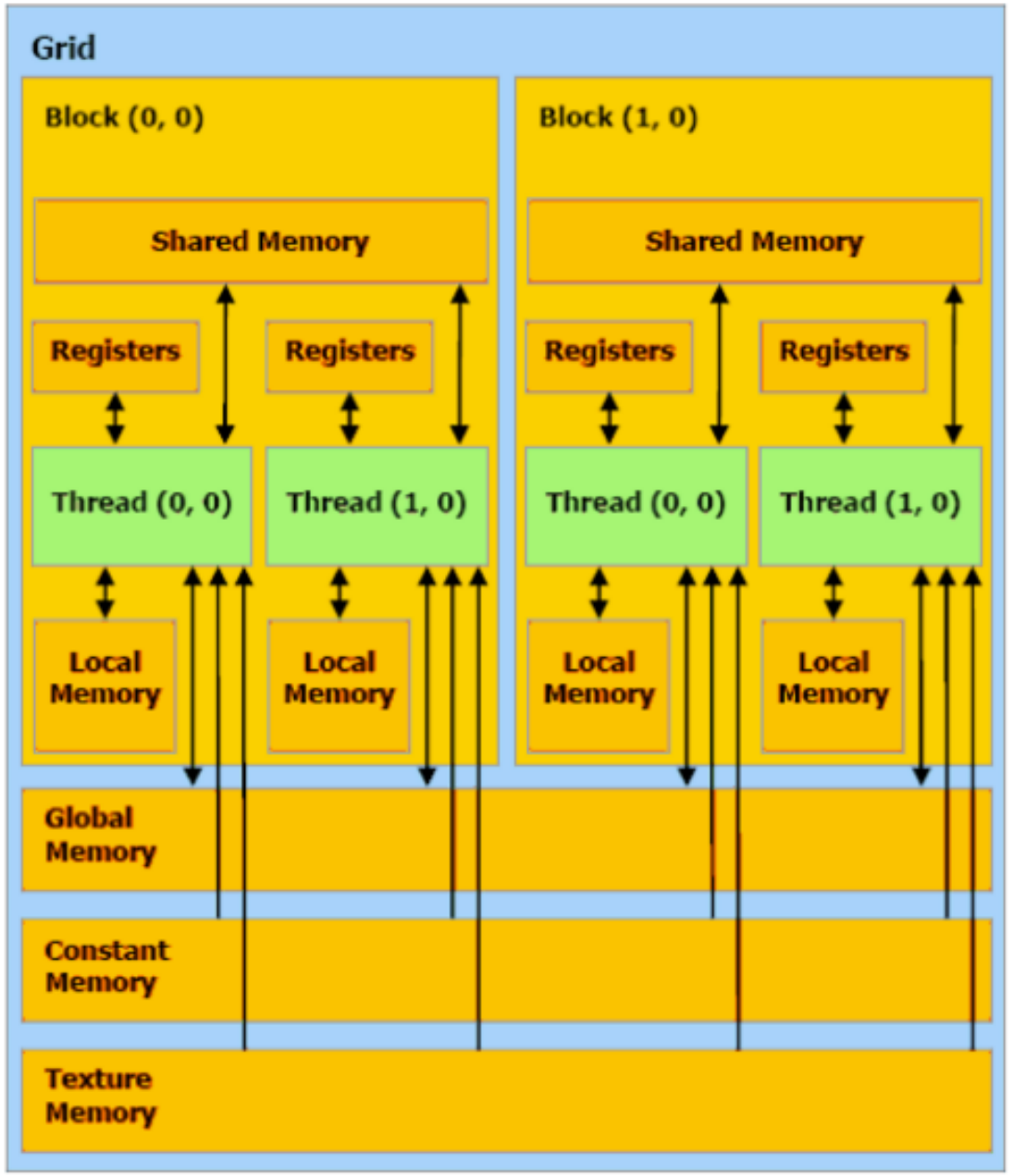
Hierarchy of GPU Programming Models

Model	GPU	CPU Equivalent
Vectorizing Compiler	PGI CUDA Fortran	gcc, icc, etc.
“Drop-in” Libraries	cuBLAS	ATLAS
Directive-driven	OpenACC, OpenMP-to-CUDA	OpenMP
High-level languages	pyCUDA, OpenCL, CUDA	python
Mid-level languages		pthreads + C/C++
Low-level languages		PTX, Shader
Bare-metal	Assembly/Machine code	SASS

GPU存储器组织

GPU Memory Breakdown

- Registers
- L1/L2/L3 cache
- Local memory
- Shared memory
- Global memory
- Constant memory
- Texture memory
- Read-only cache (CC 3.5+)



主要内存

- Registers: The fastest form of memory on the multi-processor. Is only accessible by the thread. Has the lifetime of the thread.
- Local memory: Resides in global memory and can be 150x slower than register or shared memory. Is only accessible by the thread. Has the lifetime of the thread.
- Shared Memory: Can be as fast as a register when there are no bank conflicts or when reading from the same address. Accessible by any thread of the block from which it was created. Has the lifetime of the block.
- Global memory: Potentially 150x slower than register or shared memory -- watch out for uncoalesced reads and writes. Accessible from either the host or device. Has the lifetime of the application—that is, it persistent between kernel launches.

Global Memory

- **Global memory** is separate hardware from the GPU core (containing SM's, caches, etc).
  - The vast majority of memory on a GPU is global memory
  - If data doesn't fit into global memory, process it in chunks that do fit in global memory.
- Global memory IO is the slowest form of IO on GPU
  - except for accessing host memory
- Because of this, access global memory as little as possible

Shared Memory

- Very fast memory located in the SM (streaming multiprocessor)
- Same hardware as L1 cache
  - ~5ns of latency
- Maximum size of ~48KB (varies per GPU)
- Scope of shared memory is the block

Registers

- A **Register** is a piece of memory used directly by the processor
  - Fastest “memory” possible, about 10x faster than shared memory
  - There are tens of thousands of registers in each SM
- Most stack variables declared in kernels are stored in registers
  - example: `float x;`

Local Memory

- **Local memory** is everything on the stack that can't fit in registers
- The scope of local memory is just the thread.
- Local memory is stored in global memory
  - much slower than registers

Cache

- each SM has its own L1 cache
- L2 cache
  - caches all global & local memory accesses
  - ~1MB in size
  - shared by all SM's
- L3 Cache
  - Another level of cache above L2 cache
  - Slightly slower (increased latency) than L2 cache but also larger

Constant Memory

Constant memory is global memory with a special cache

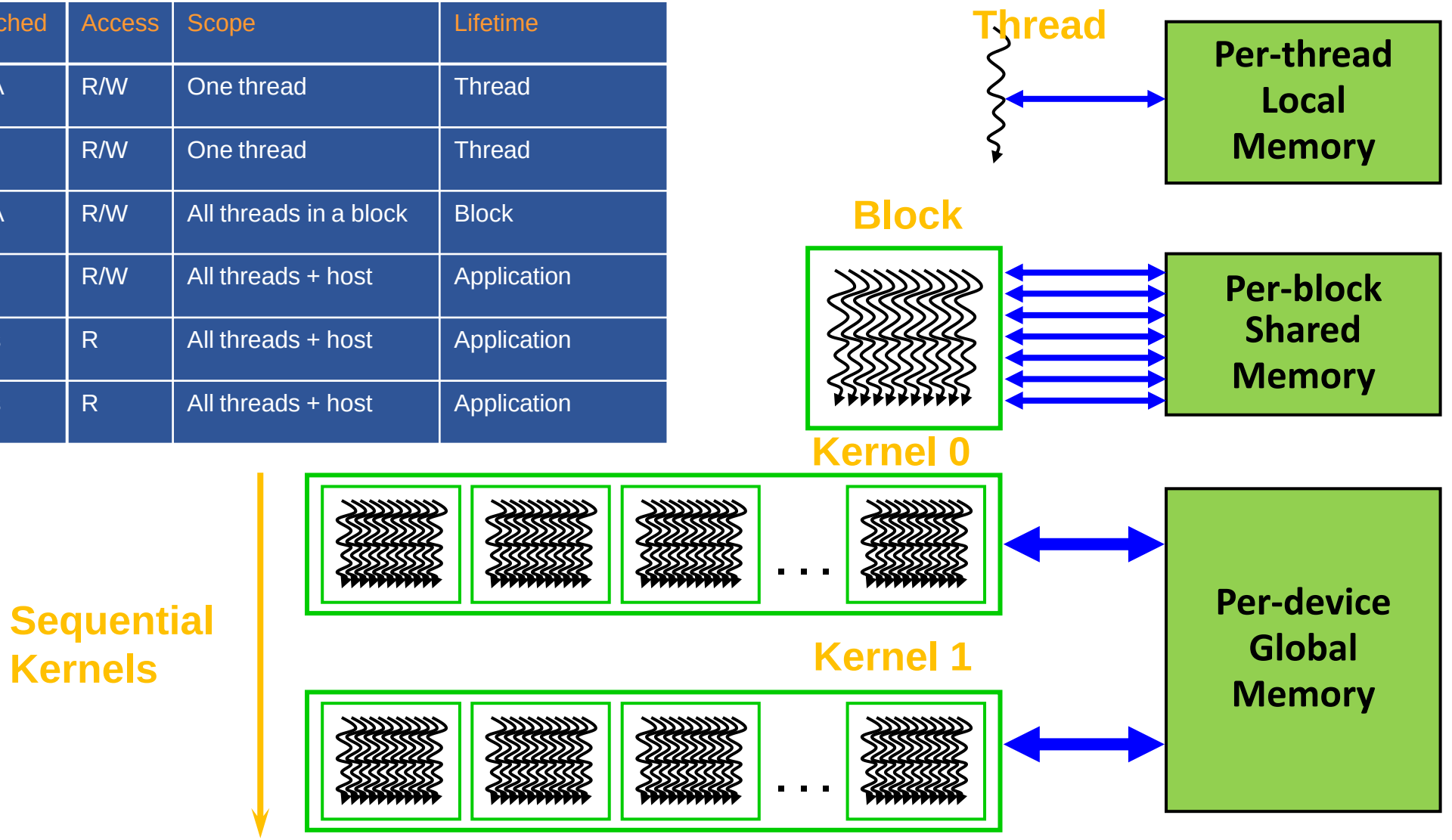
- Used for constants that cannot be compiled into program
- Constants must be set from host before running kernel.

Constant Cache

- 8KB cache on each SM specially designed to broadcast a single memory address to all threads in a warp (called static indexing)
  - Can also load any statically indexed data through constant cache using “load uniform” (LDU) instruction

存储器模型总结

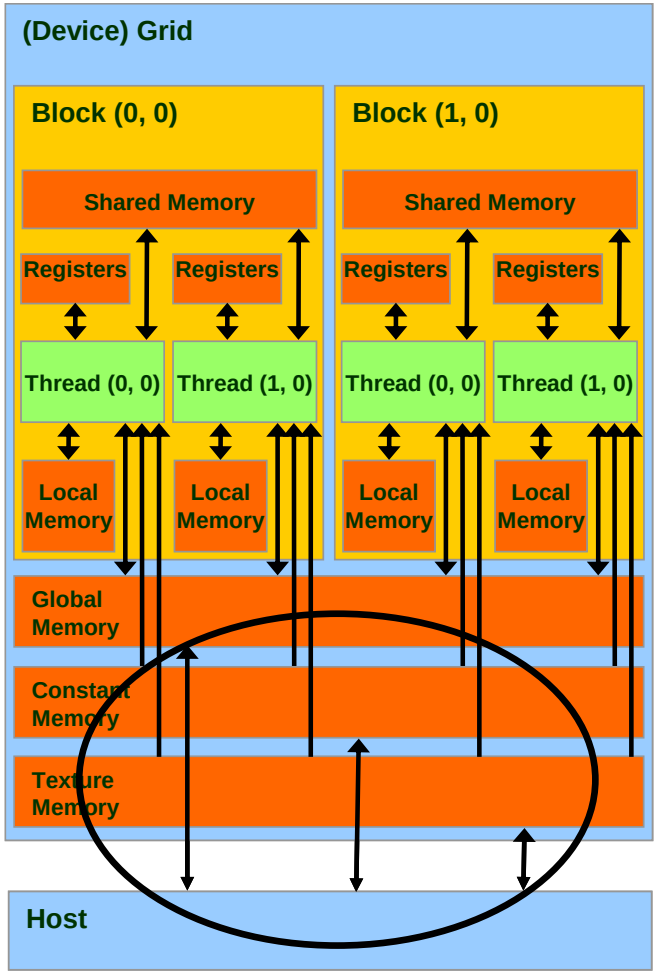
Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip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



Host – Device数据交换

- cudaMemcpy()
  - Memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer:Host to Host, Host to Device, Device to Host, Device to Device

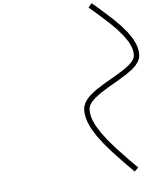
```
cudaMemcpy(Md, M.elements, size, cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md, size, cudaMemcpyDeviceToHost);
```



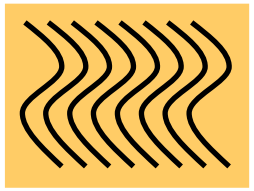
线程组织及计算模型

GPU及其程序设计模型

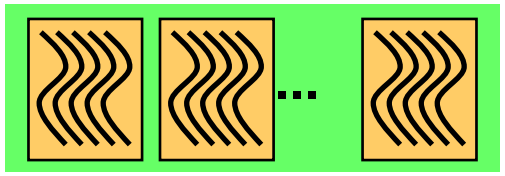
Software



Thread



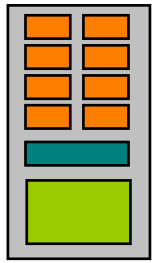
Thread
Block



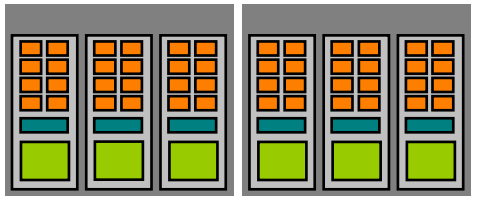
Grid

GPU

CUDA Core



Multiprocessor



Device

Distributed by the CUDA runtime (threadIdx).
Threads are executed by scalar processors

Block - A user defined group of 1 to ~512 threads (blockIdx).
Thread blocks are executed on multiprocessors

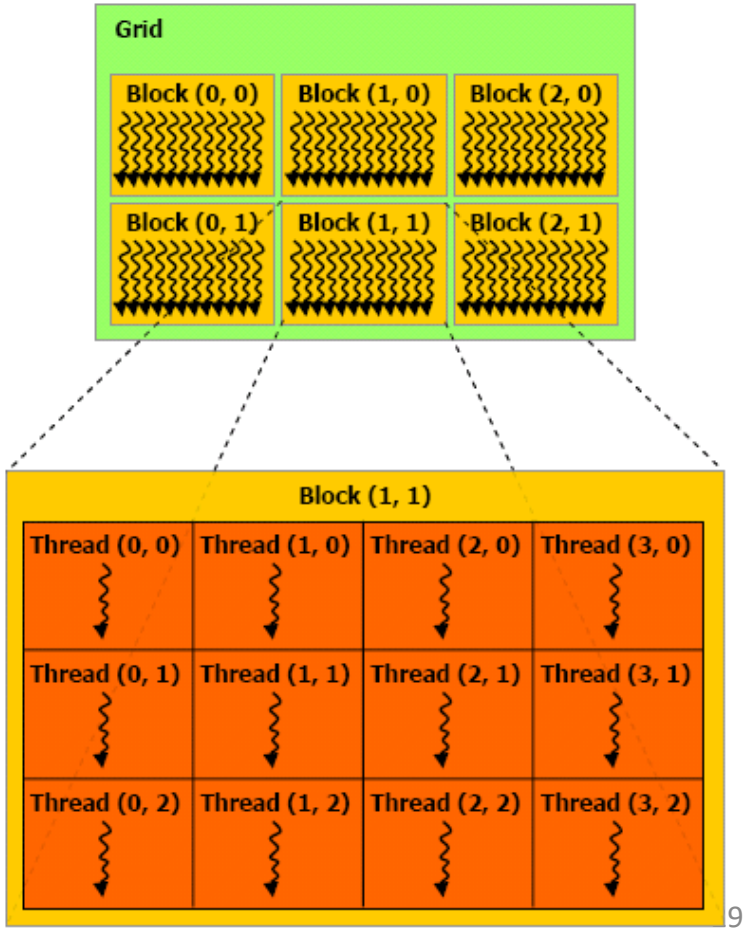
A group of one or more blocks. A grid is created for each CUDA kernel function called.

线程组织模型

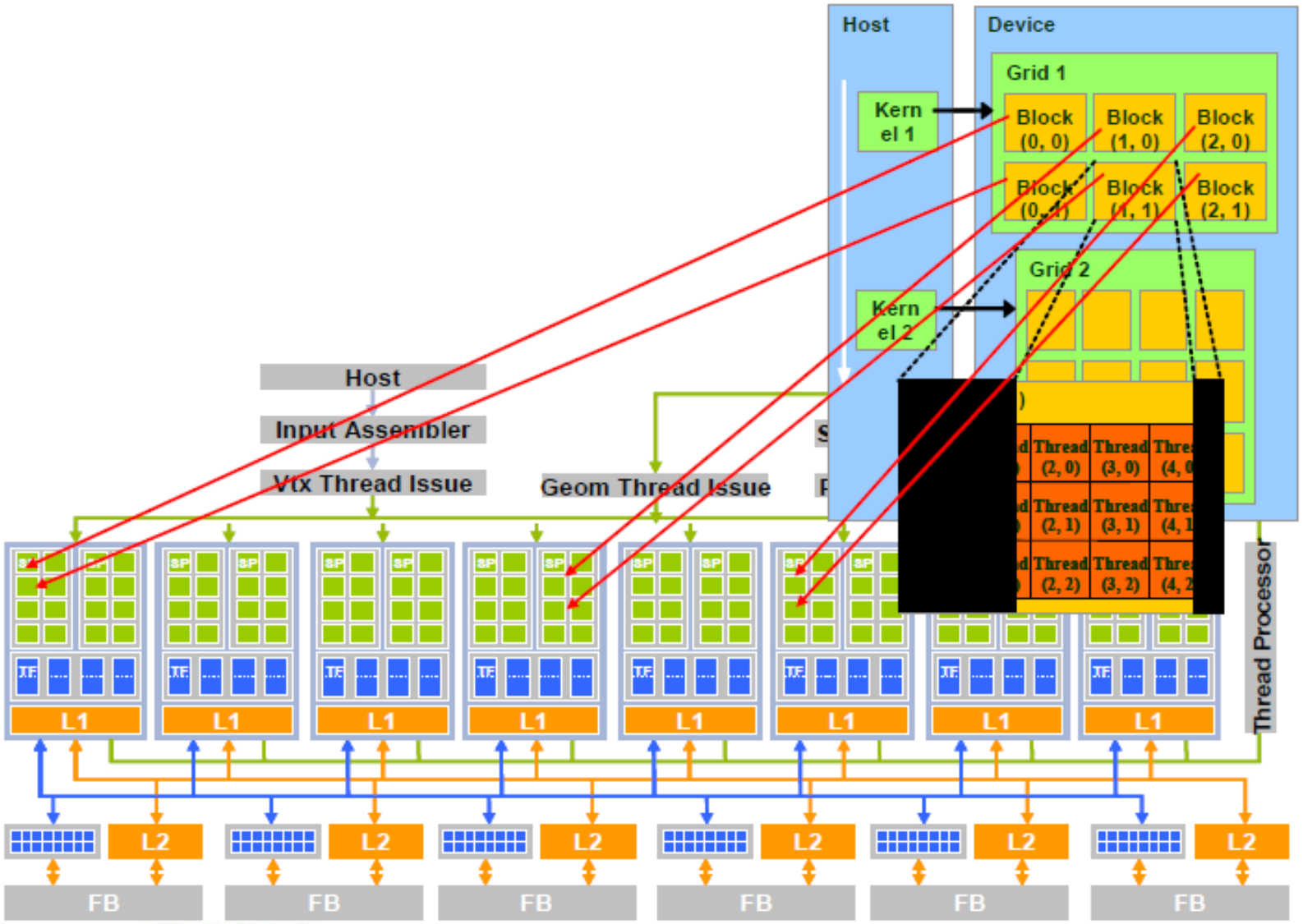
- 所有线程执行相同的顺序程序
- 层次结构：Grid -》 Block-》 Thread

- 线程被划分成线程块 (Block)
- 同一线程块内的线程可以通过共享SM资源相互协作

- 每个线程和线程块拥有唯一ID
- 通过内建变量读取



线程映射层次

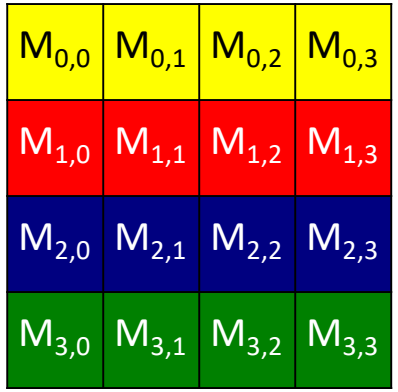


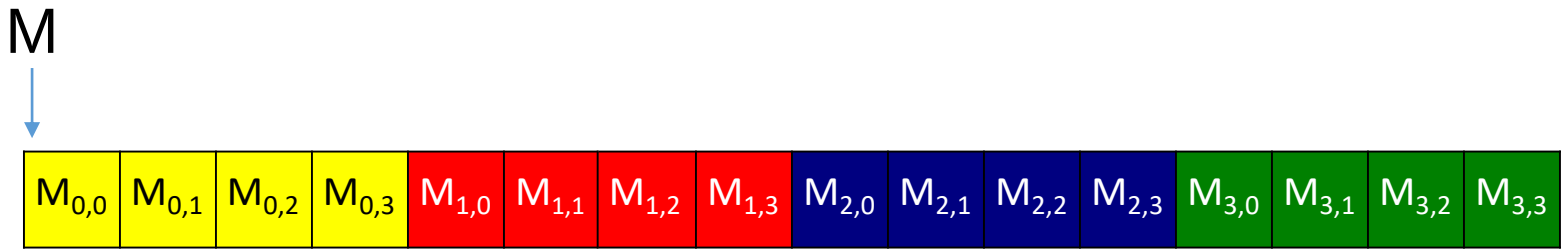
- 大量的线程并行执行
- 每个线程拥有私有变量/存储区域
- 线程之间拥有共享的存储区域

Block and Grid Dimensions

- For many parallelizeable problems involving arrays, it's useful to think of multidimensional arrays.
  - E.g. linear algebra, physical modelling, etc, where we want to assign unique thread indices over a multidimensional object

图像处理






```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int height, int width){
    // Calculate the row # of the d_Pin and d_Pout element
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    // each thread computes one element of d_Pout if in range
    if ((Row < height) && (Col < width)) {
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
    }
}
```

Dim3

Dim3是一定义在vector_types.h的结构，可用于定义Grid和Block的维数.

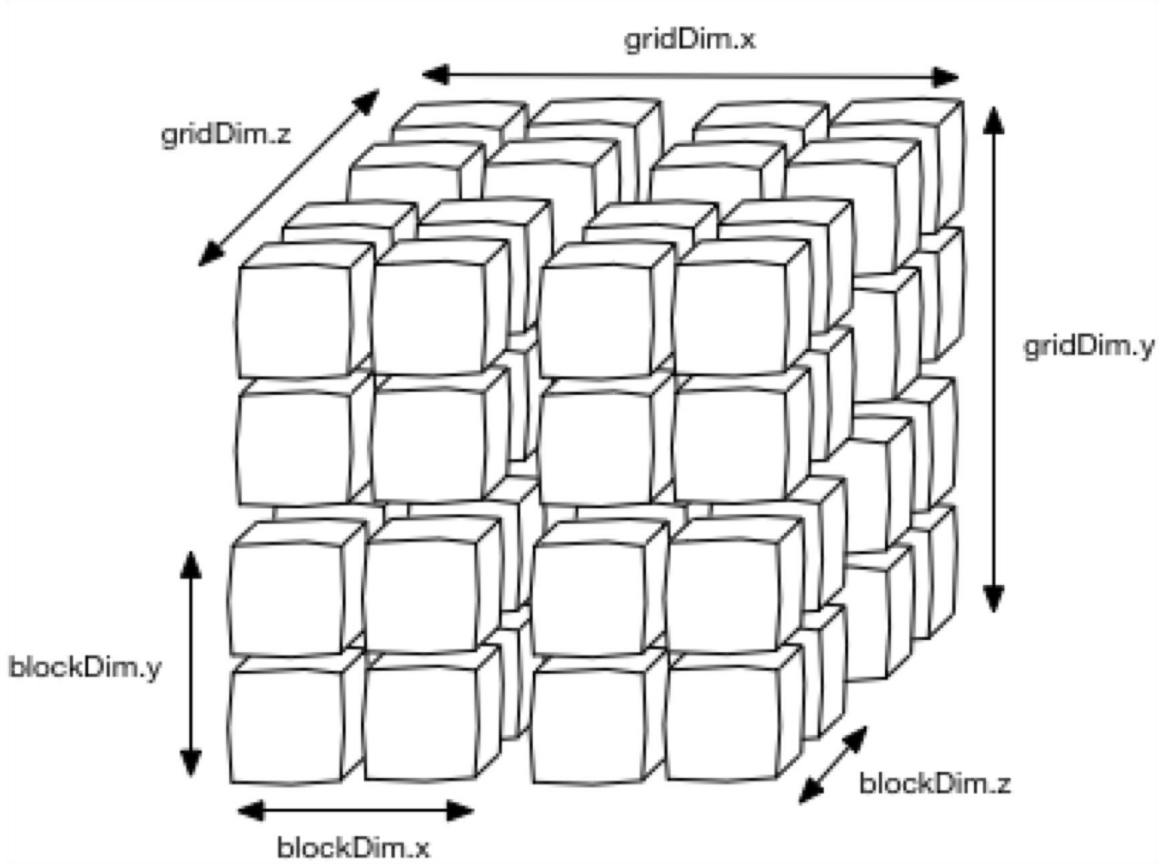
```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#if defined(__cplusplus)
    __host__ __device__ dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
    __host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
    __host__ __device__ operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
#endif /* __cplusplus */
};
```

Works for dimensions 1, 2, and 3:

- dim3 grid(256); // defines a grid of 256 x 1 x 1 blocks
- dim3 block(512, 512); // defines a block of 512 x 512 x 1 threads
- foo<<<grid, block>>>(...);

Built-in Variables

- dim3 gridDim;
  - Dimensions of the grid in blocks (gridDim.z unused below version 2.x)
- 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

```
dim3 dimGrid(5, 2, 1);
dim3 dimBlock(4, 3, 6);
KernelFunction<<<dimGrid, dimBlock>>>(…);
```

The threads created will have:

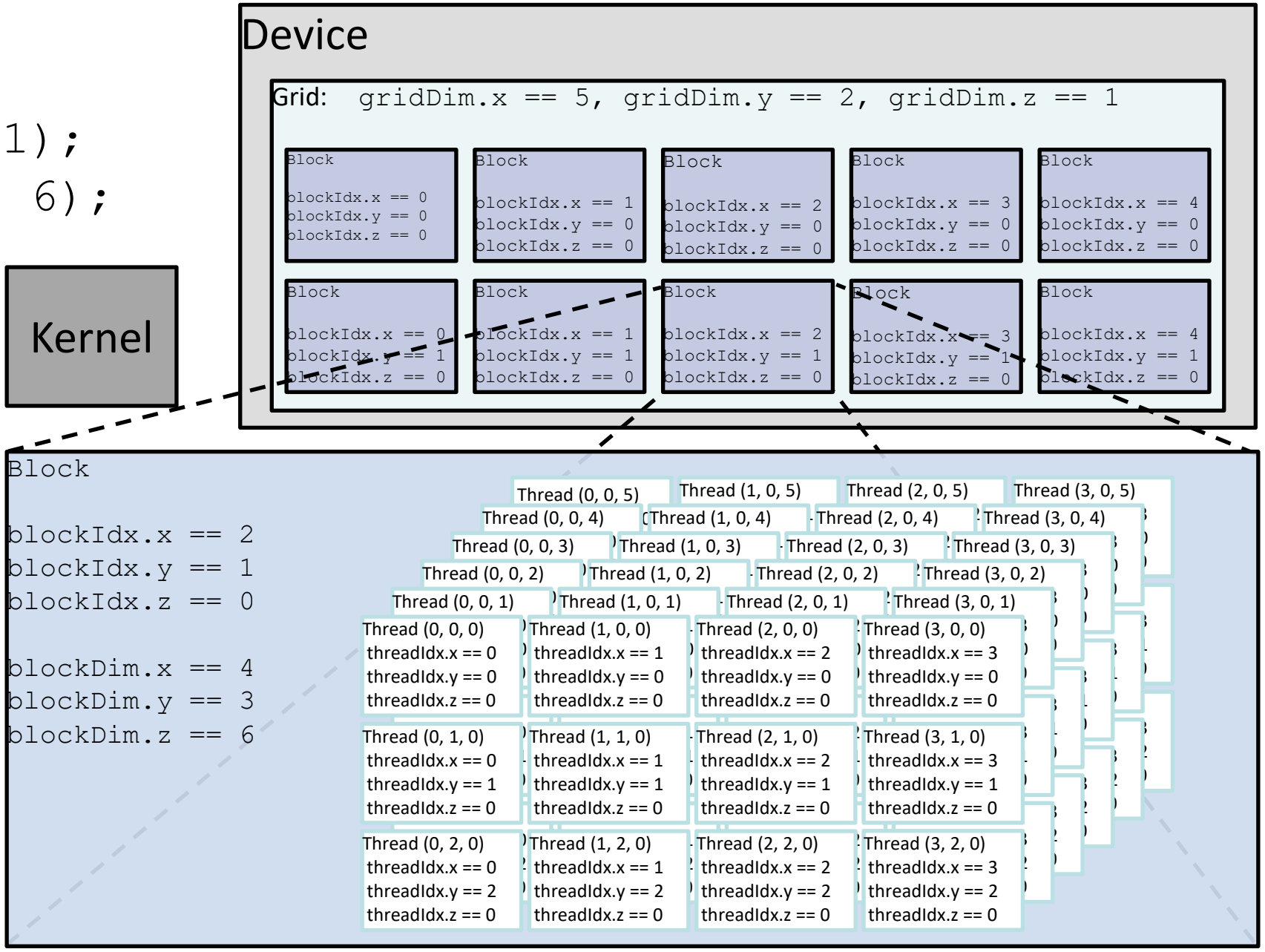
```
gridDim.x = 5, blockIdx.x = 0 … 4
gridDim.y = 2, blockIdx.y = 0 … 1
gridDim.z = 1, blockIdx.z = 0 … 0

blockDim.x = 4, threadIdx.x = 0 … 3
blockDim.y = 3, threadIdx.y = 0 … 2
blockDim.z = 6, threadIdx.z = 0 … 5
```

Therefore the total number of threads will be

```
5 * 2 * 1 * 4 * 3 * 6 = 720
```

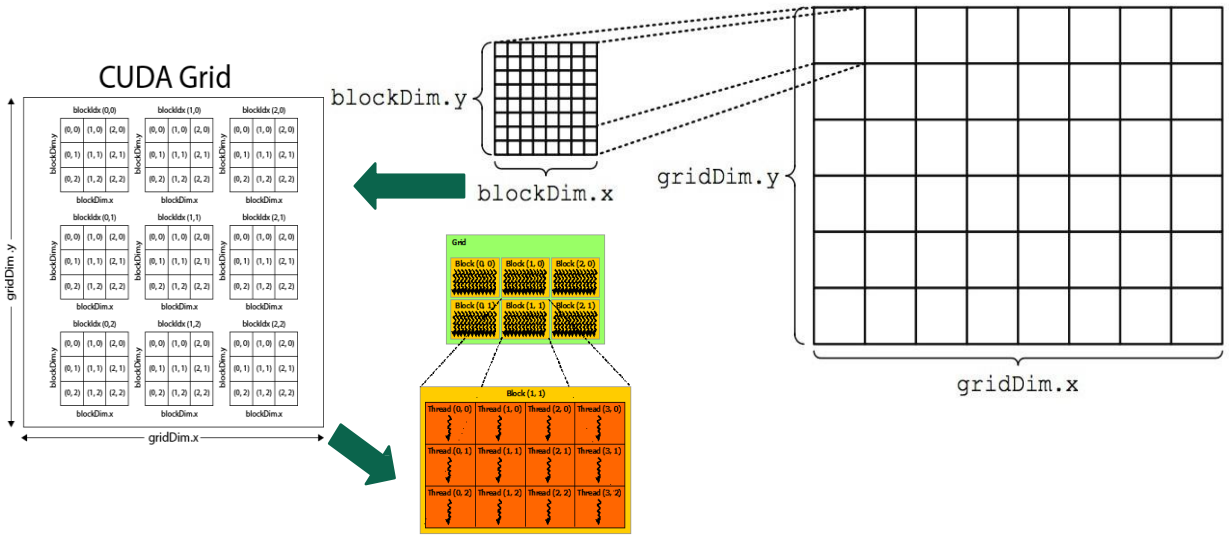
CUDA Thread Organization

```
dim3 dimGrid(5, 2, 1);
dim3 dimBlock(4, 3, 6);
```

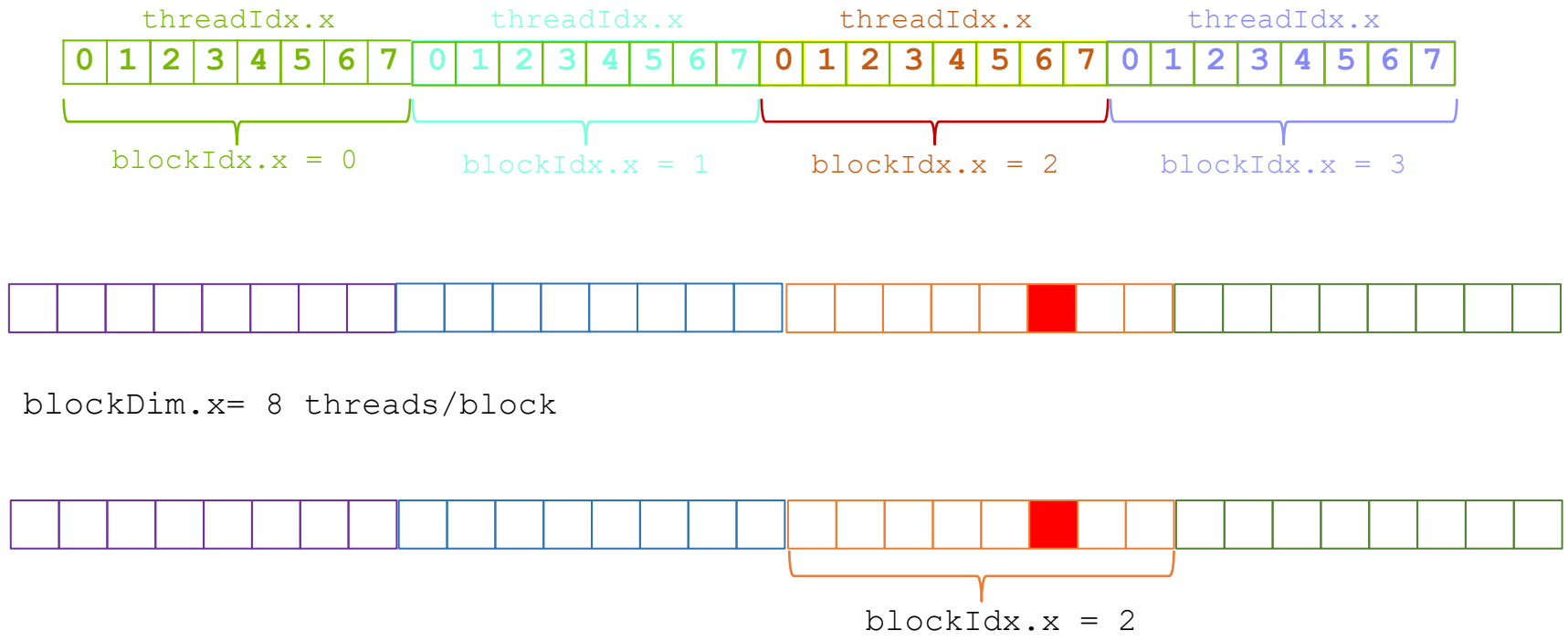


线程索引

- *Thread ID*: Scalar thread identifier
- 线程索引: threadIdx
- Grid只有一block
- 一维Block: Thread ID = Thread Index
- 二维Block (D_x, D_y)
  - Thread ID of index (x, y) = x + y D_x
- 三维Block (D_x, D_y, D_z)
  - Thread ID of index (x, y, z) = x + y D_x + z D_x D_y



Example




```
int index = threadIdx.x + blockIdx.x * blockDim.x;
          =   5     +      2          *   8;
          = 21;
```

2D grid of 3D blocks

```
__device__
int getGlobalIdx_2D_3D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x;
      int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                     + (threadIdx.z * (blockDim.x * blockDim.y))
                     + (threadIdx.y * blockDim.x) + threadIdx.x;
      return threadId;
}
```

3D grid of 1D blocks

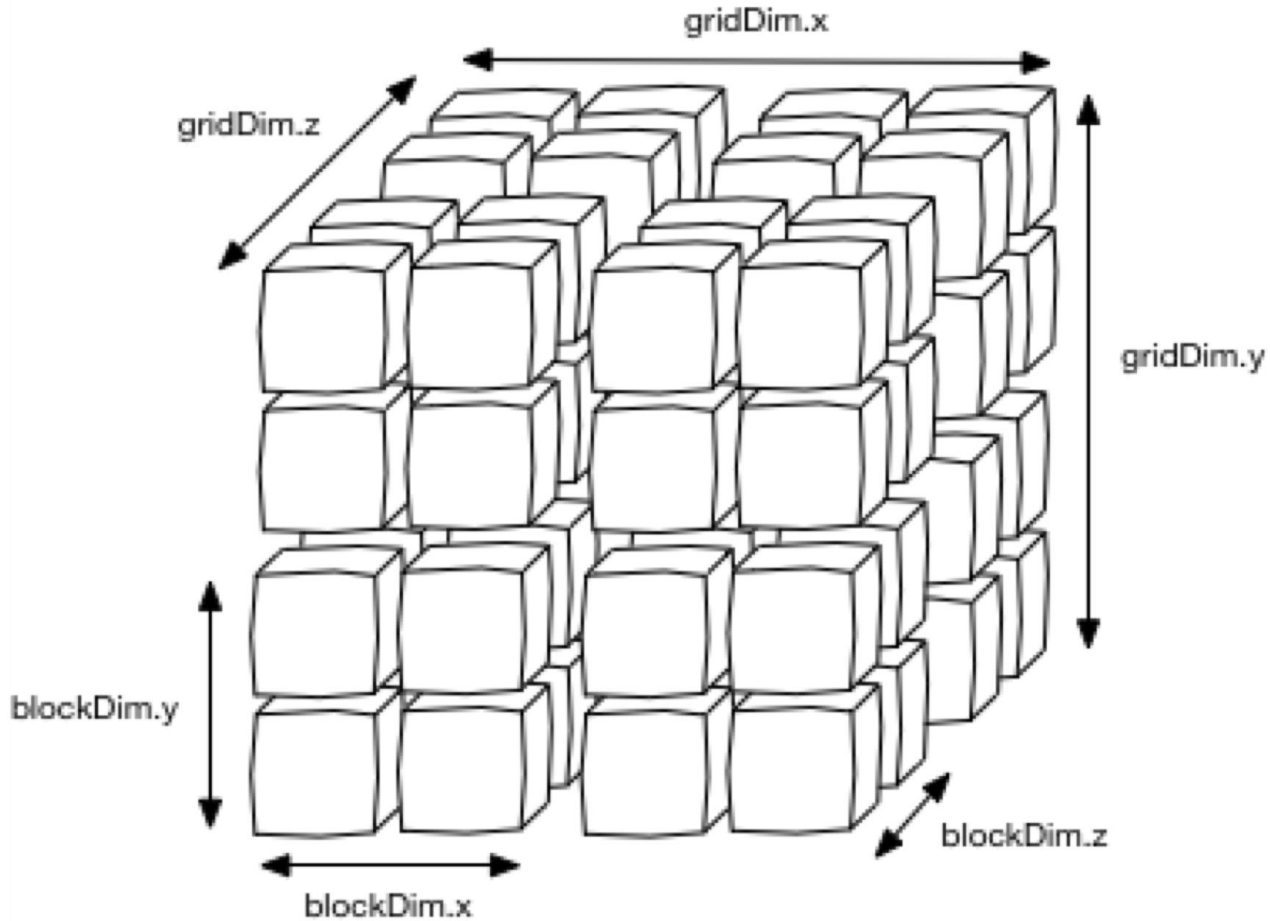
```
__device__
int getGlobalIdx_3D_1D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x
                    + gridDim.x * gridDim.y * blockIdx.z;
      int threadId = blockId * blockDim.x + threadIdx.x;
      return threadId;
}
```

3D grid of 2D blocks


```
__device__
int getGlobalIdx_3D_2D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x
                    + gridDim.x * gridDim.y * blockIdx.z;
      int threadId = blockId * (blockDim.x * blockDim.y)
                     + (threadIdx.y * blockDim.x) + threadIdx.x;
      return threadId;
}
```

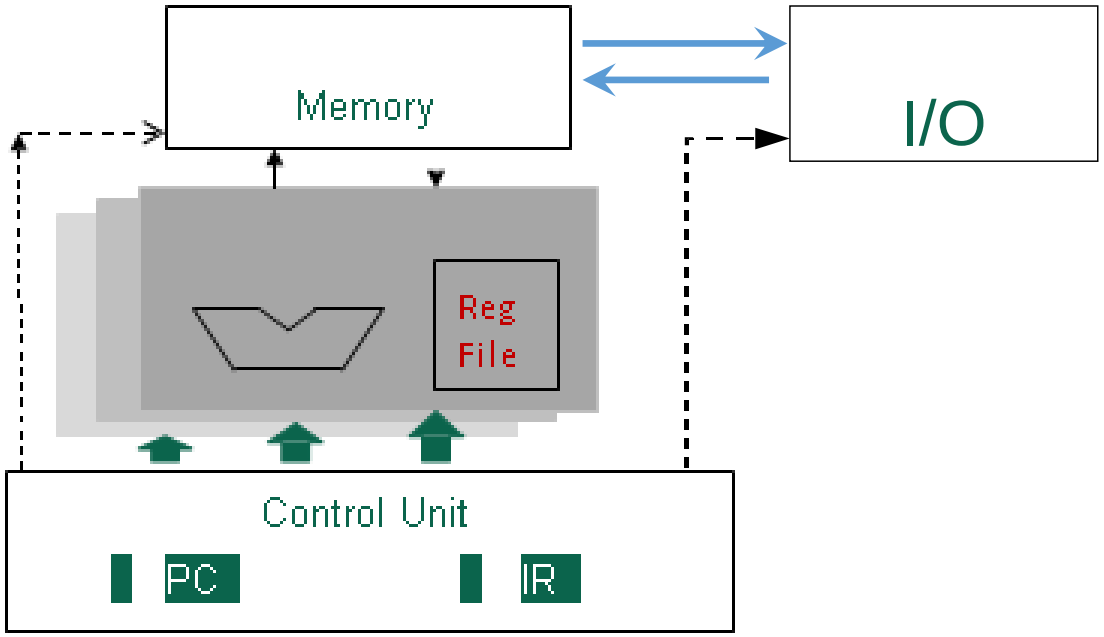
3D grid of 3D blocks

```
__device__
int getGlobalIdx_3D_3D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x
                     + gridDim.x * gridDim.y * blockIdx.z;
      int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                     + (threadIdx.z * (blockDim.x * blockDim.y))
                     + (threadIdx.y * blockDim.x) + threadIdx.x;
      return threadId;
}
```



Single Instruction, Multiple Data (SIMD)

- SIMD describes a class of instructions which perform the same operation on multiple registers simultaneously.
- Example: Add some scalar to 3 registers, storing the output for each addition in those registers.
- CPUs also have SIMD instructions and are very important for applications that need to do a lot of number crunching



Warp-based SIMD vs. Traditional SIMD

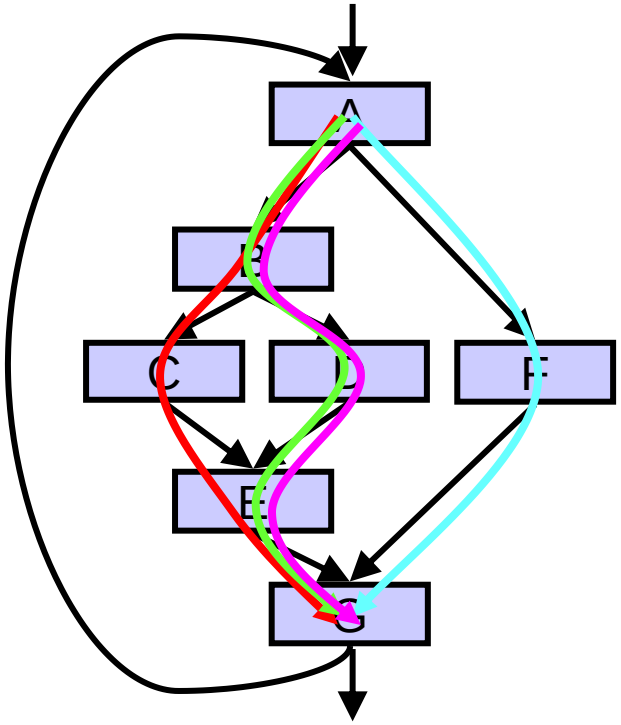
- Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions
- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

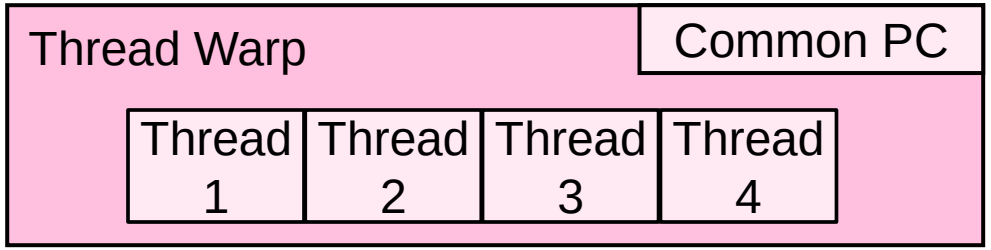
SPMD

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization
- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers
- Essentially, multiple instruction streams execute the same program
  - Each program/procedure can 1) execute a different control-flow path, 2) work on different data, at run-time
  - Modern GPUs programmed in a similar way on a SIMD computer

SIMT

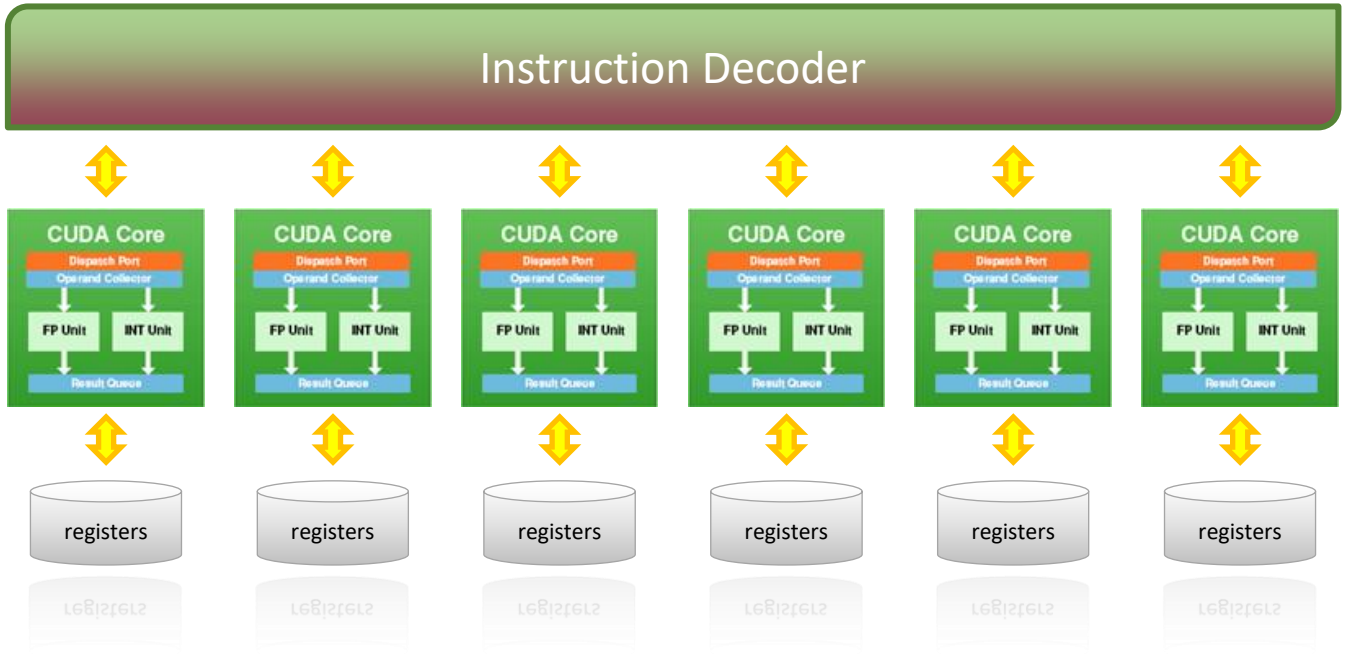
- SPMD Execution on SIMD Hardware
  - NVIDIA calls this “Single Instruction, Multiple Thread” (“SIMT”) execution





SIMT Execution

- Single Instruction Multiple Threads
  - All cores are executing the same instruction
  - Each core has its own set of registers



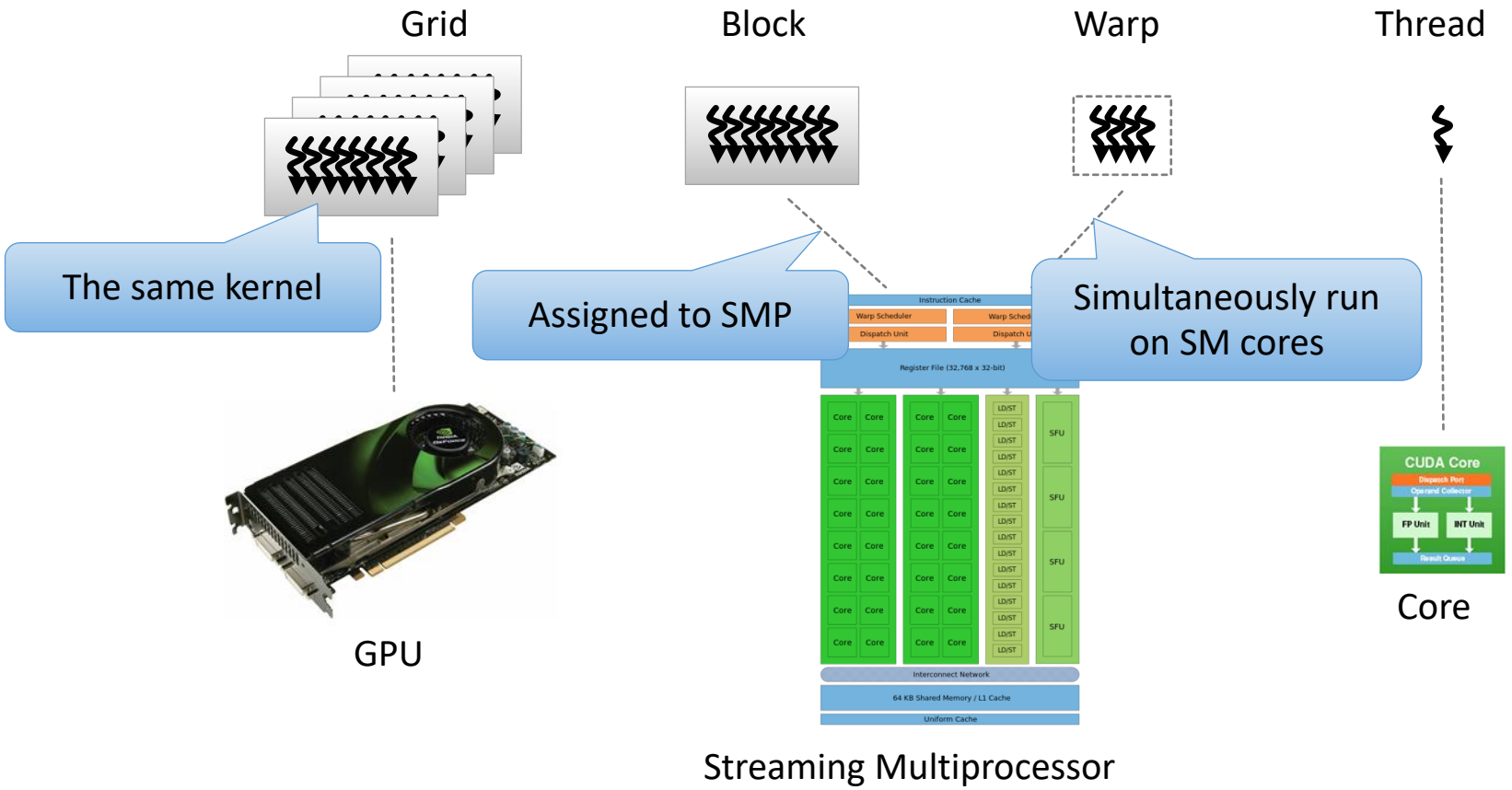
SIMT vs. SIMD

- Single Instruction Multiple Threads
  - Width-independent programming model
  - Serial-like code
  - Achieved by hardware with a little help from compiler
  - Allows code divergence

- Single Instruction Multiple Data
  - Explicitly expose the width of SIMD vector
  - Special instructions
  - Generated by compiler or directly written by programmer
  - Code divergence is usually not supported

Thread-Core Mapping

- How are threads assigned to SMPs

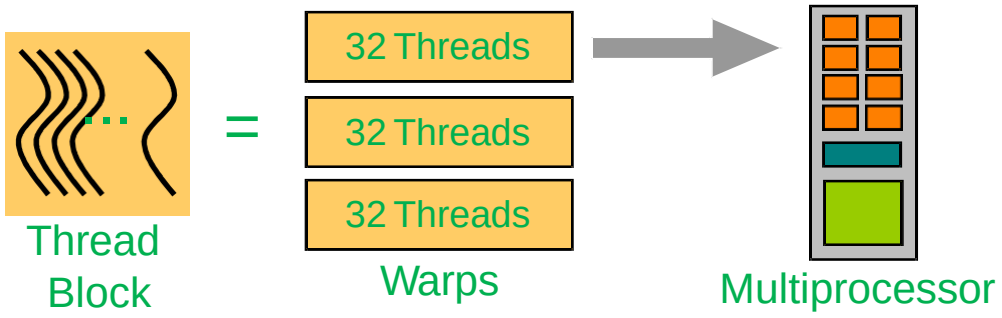


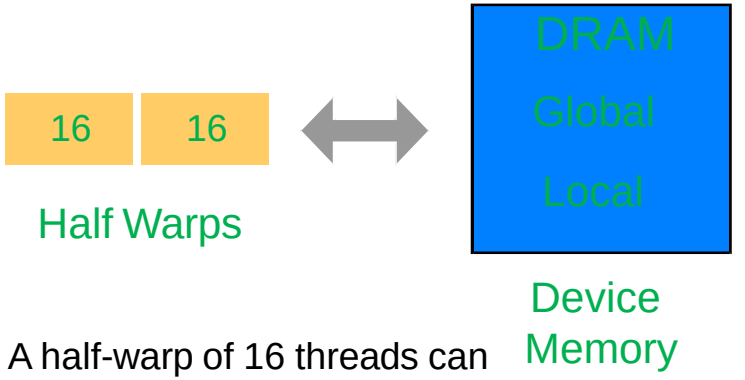
Each Block is executed as 32-thread Warps

- An implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- Threads in a warp execute in SIMD
- Future GPUs may have different number of threads in each warp

Thread-Core Mapping

- Cores in a streaming multiprocessor (SM) are SIMT cores:
  - Maximum number of threads in a block depends on the compute capability
- Decomposition
  - Each block assigned to the SMP is divided into warps and the warps are assigned to schedulers
- GPU multiprocessor creates, manages, schedules and executes threads in warps of 32*
  - minimum of 32 threads all doing the same thing at (almost) the same time (Warp executes one common instruction at a time).
  - Threads are allowed to branch, but branches are serialized





A half-warp of 16 threads can coordinate global memory accesses into a single transaction

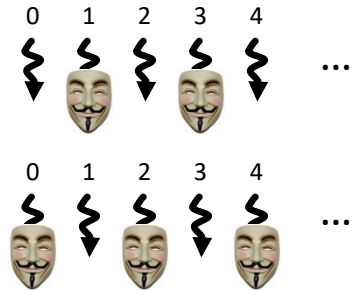
Warp

- Instructions are issued per warp
  - It takes 4 clock cycles to issue a single instruction for the whole warp
- If an operand is not ready the warp will stall
- execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data
  - Select warp that is ready at every instruction cycle
  - Fast Context Switch： When a warp gets stalled (E.g., by data load/store), scheduler switch to next active warp
- Threads in any given warp execute in lock-step, but to synchronise across warps: __syncthreads()

Branches

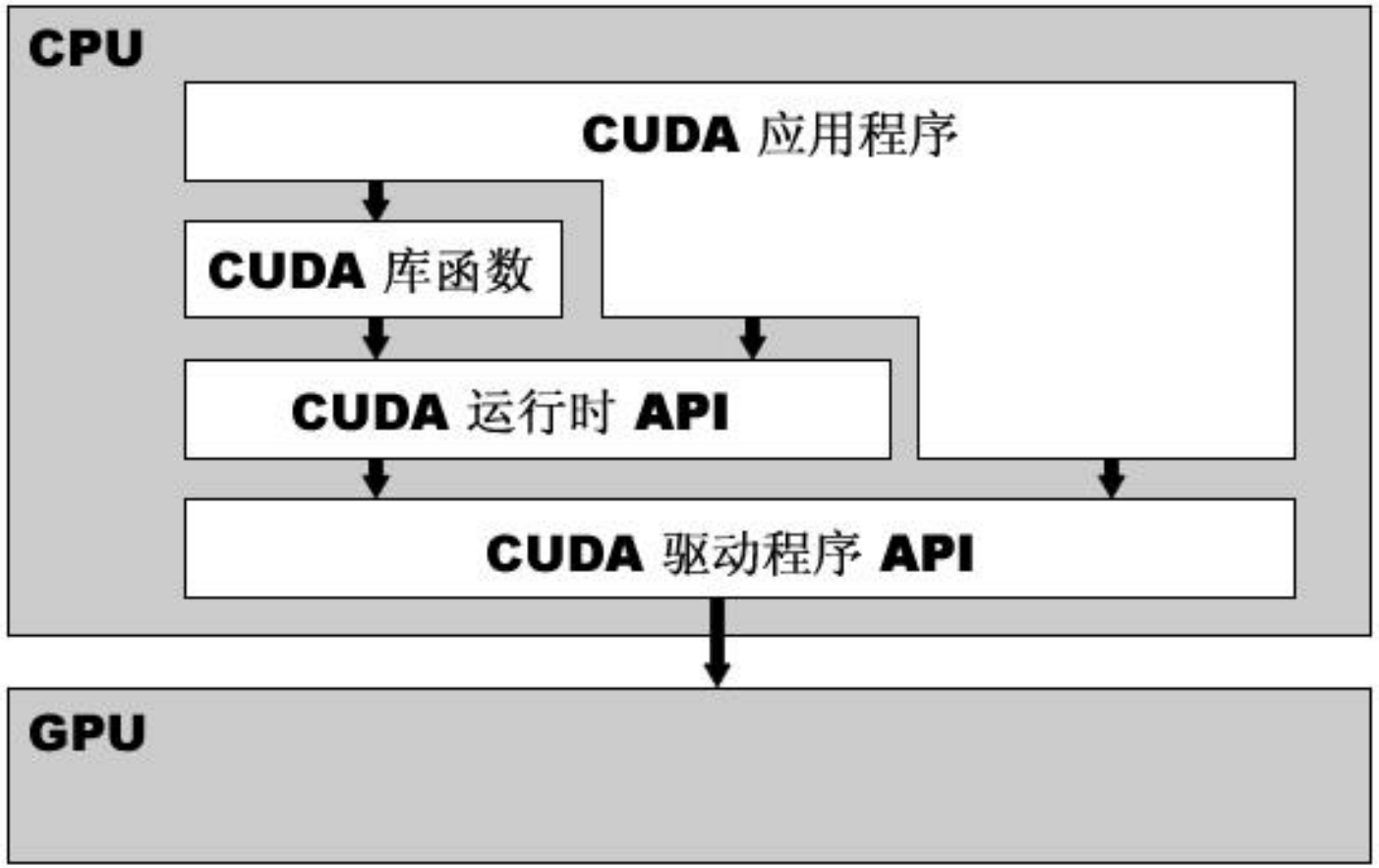
- Masking Instructions
  - In case of data-driven branches
    - if-else conditions, while loops, …
  - All branches are traversed, threads mask their execution in invalid branches

```
if (threadIdx.x % 2 == 0) {
    ... even threads code ...
} else {
    ... odd threads code ...
}
```



CUDA C/C++

CUDA API



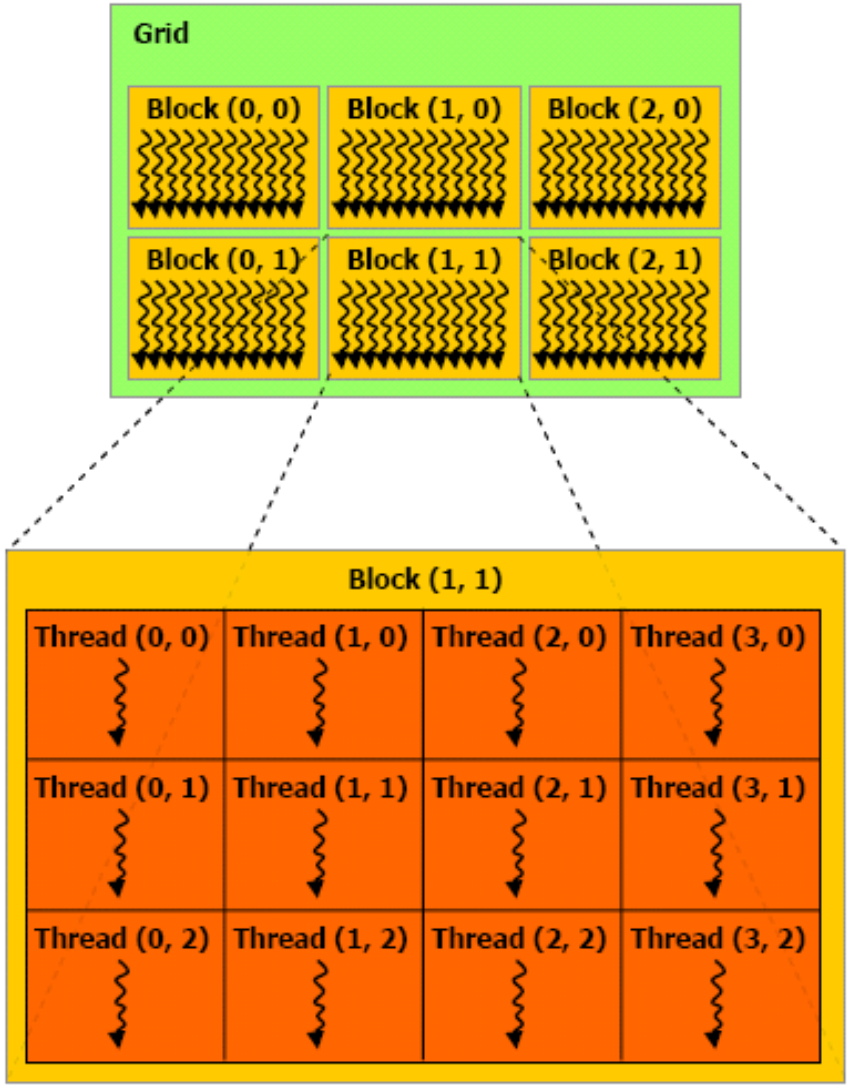
Extended C/C++

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx, …**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image){
  __shared__ float region[M];
  ...
  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}
// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA对C/C++的扩展：内建变量

- 无需定义/声明, 直接在kernel中调用
- dim3 gridDim; // Grid大小
- dim3 blockDim; // Block大小
- dim3 blockIdx; // Block ID
- dim3 threadIdx; // Thread ID



变量类型

- __device__
  - 储存于GPU上的global memory空间
  - 和应用程序具有相同的生命期(lifetime)
  - 可被grid中所有线程存取
- __constant__
  - 储存于GPU上的constant memory空间
  - 和应用程序具有相同的生命期(lifetime)
  - 可被grid中所有线程存取
- __shared__
  - 储存于GPU上thread block内的共享存储器
  - 和thread block具有相同的生命期(lifetime)
  - 只能被thread block内的线程存取
- local变量
  - 储存于SM内的寄存器和local memory
  - 和thread具有相同的生命期(lifetime)
  - Thread私有

变量类型

- `__device__`与`__local__` ,`__shared__`,或 `__constant__`一起时，可省略

Variable declaration	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

变量类型

- Pointers只能指向global memory:
  - Allocated in the host and passed to the kernel:
    - `__global__ void KernelFunc(float* ptr)`
  - Obtained as the address of a global variable: `float* ptr =&GlobalVar;`
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

CUDA C/C++:函数限定符

- 函数限定符用来规定函数是在host还是在device上执行，以及这个函数是从host调用还是从device调用。
  - __device__函数在device端执行，并且也只能从device端调用
  - __global__函数即kernel函数，它在设备上执行，但是要从host端调用
  - __host__函数在host端执行，也只能从host端调用，与一般的C函数相同

CUDA函数定义

	Executed on the:	Only callable from the:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

- __global__ 定义kernel函数
  - 必须返回void
- __device__ 函数
  - 不能用&运算符取地址, 不支持递归调用, 不支持静态变量(static variable), 不支持可变长度参数函数调用

CUDA C/C++数学函数

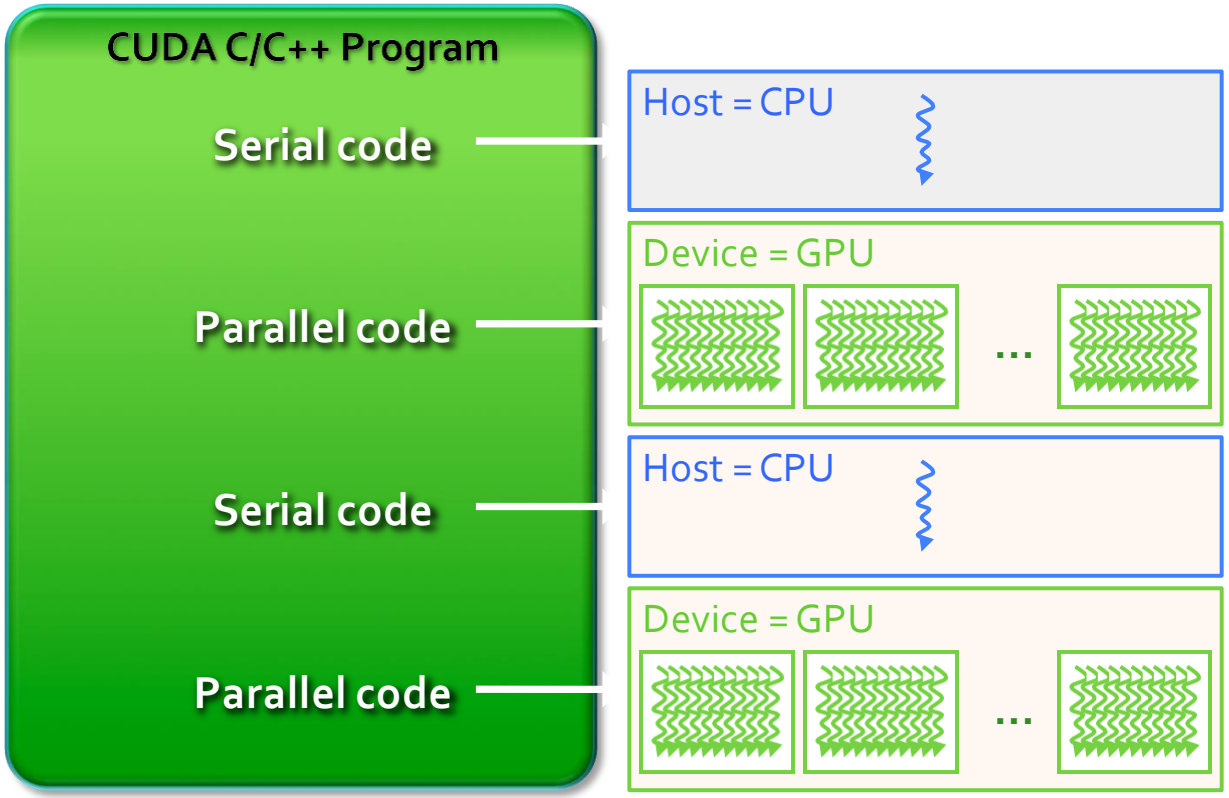
- pow, sqrt, cbrt, hypot, exp, exp2, expm1, log, log2, log10, log1p, sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, asinh, acosh, atanh, ceil, floor, trunc, round, etc.
  - 只支持标量运算
  - 许多函数有一个快速、较不精确的对应版本
    - 以”__”为前缀，如__sin()
    - 编译开关**-use_fast_math**强制生成该版本的目标码

Kernel执行参数

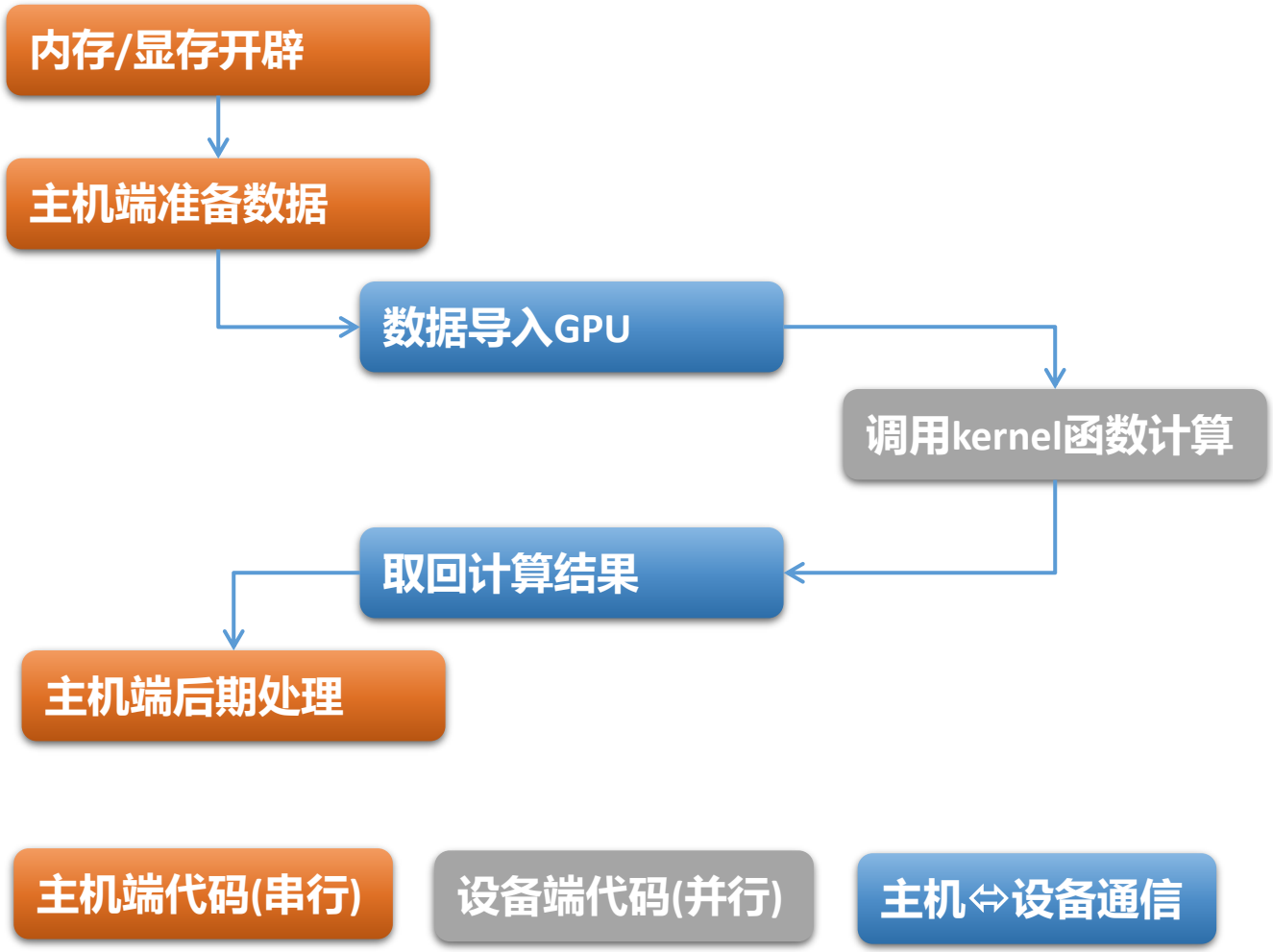
- <<< >>>运算符，用来传递一些kernel执行参数
  - Grid的大小和维度
  - Block的大小和维度
- 调用: my_kernel <<< grid_dim, block_dim >>> (...);
- <<< grid_dim, block_dim >>>实际是为gridDim和blockDim赋值
- 决定当前线程需要处理的数据和结果存储的位置

CUDA程序结构及执行

CUDA程序结构


CUDA C/C++ Program
Serial code
Parallel code
Serial code
Parallel code
Host = CPU
Device = GPU
...
Host = CPU
Device = GPU
...

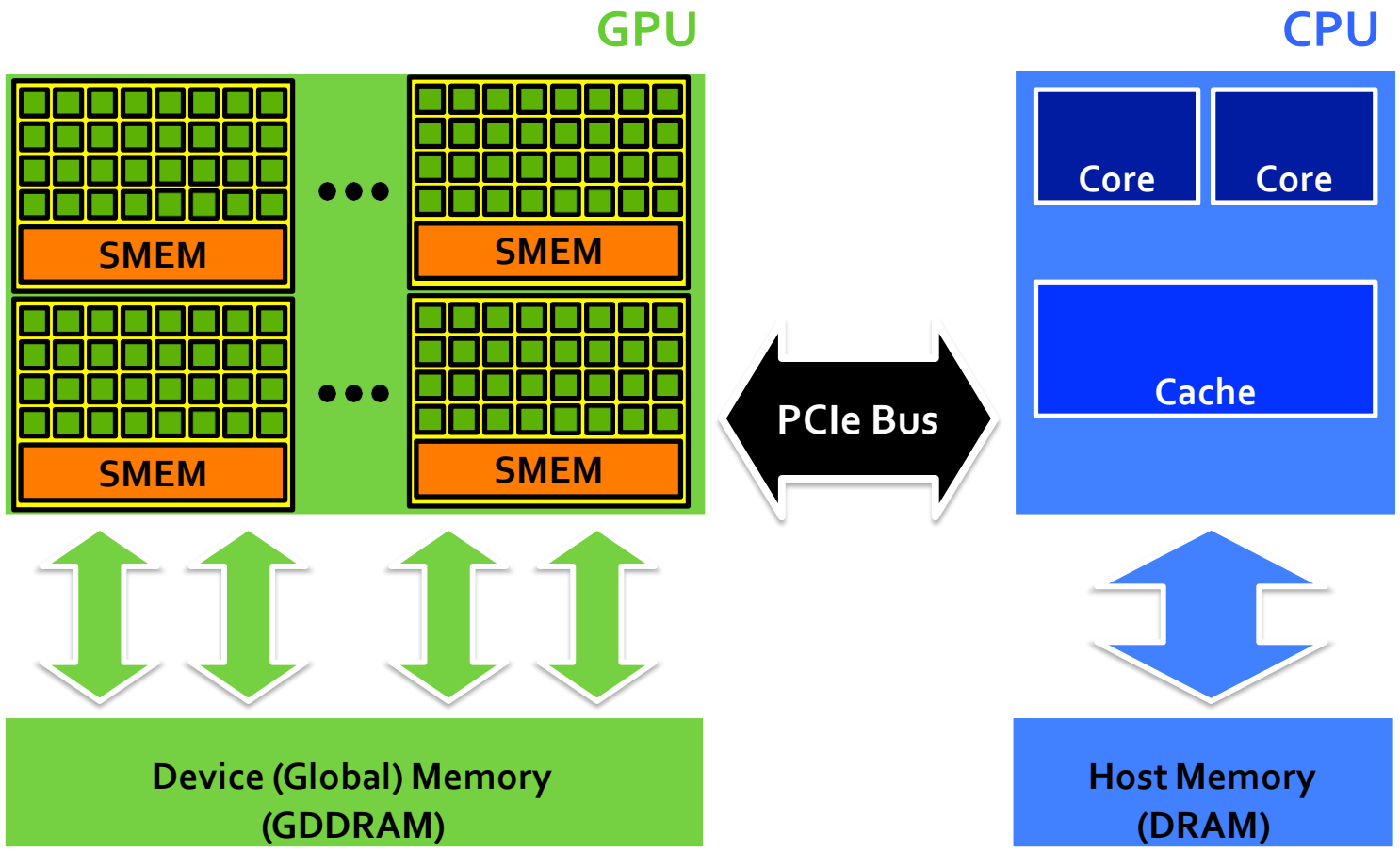
执行过程



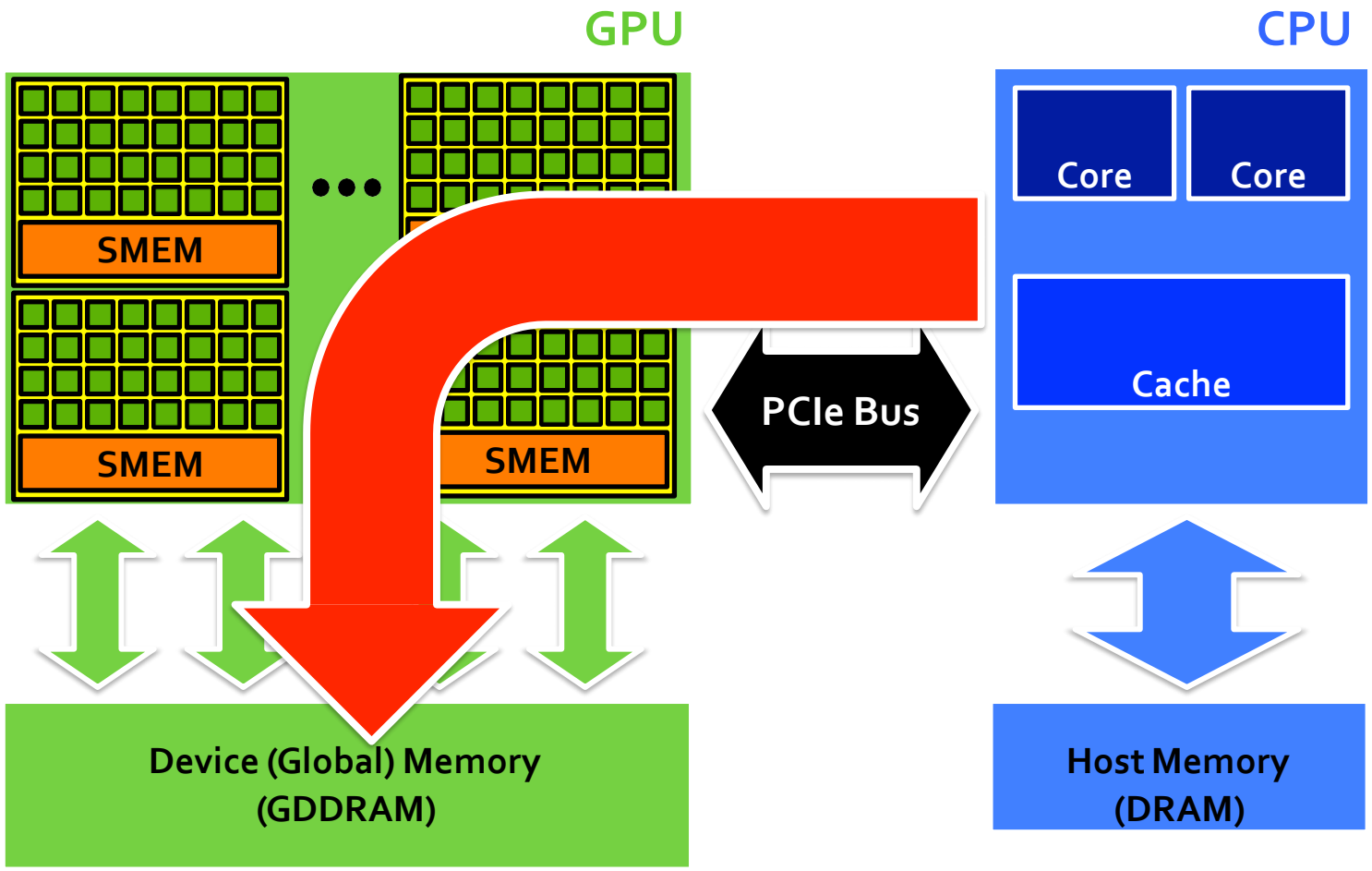
A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3    DimGrid(100, 50);
dim3    DimBlock(4, 8, 8);

// 64 bytes of shared memory
size_t SharedMemBytes = 64;
KernelFunc<<< DimGrid, DimBlock,
    SharedMemBytes >>>(...);
```

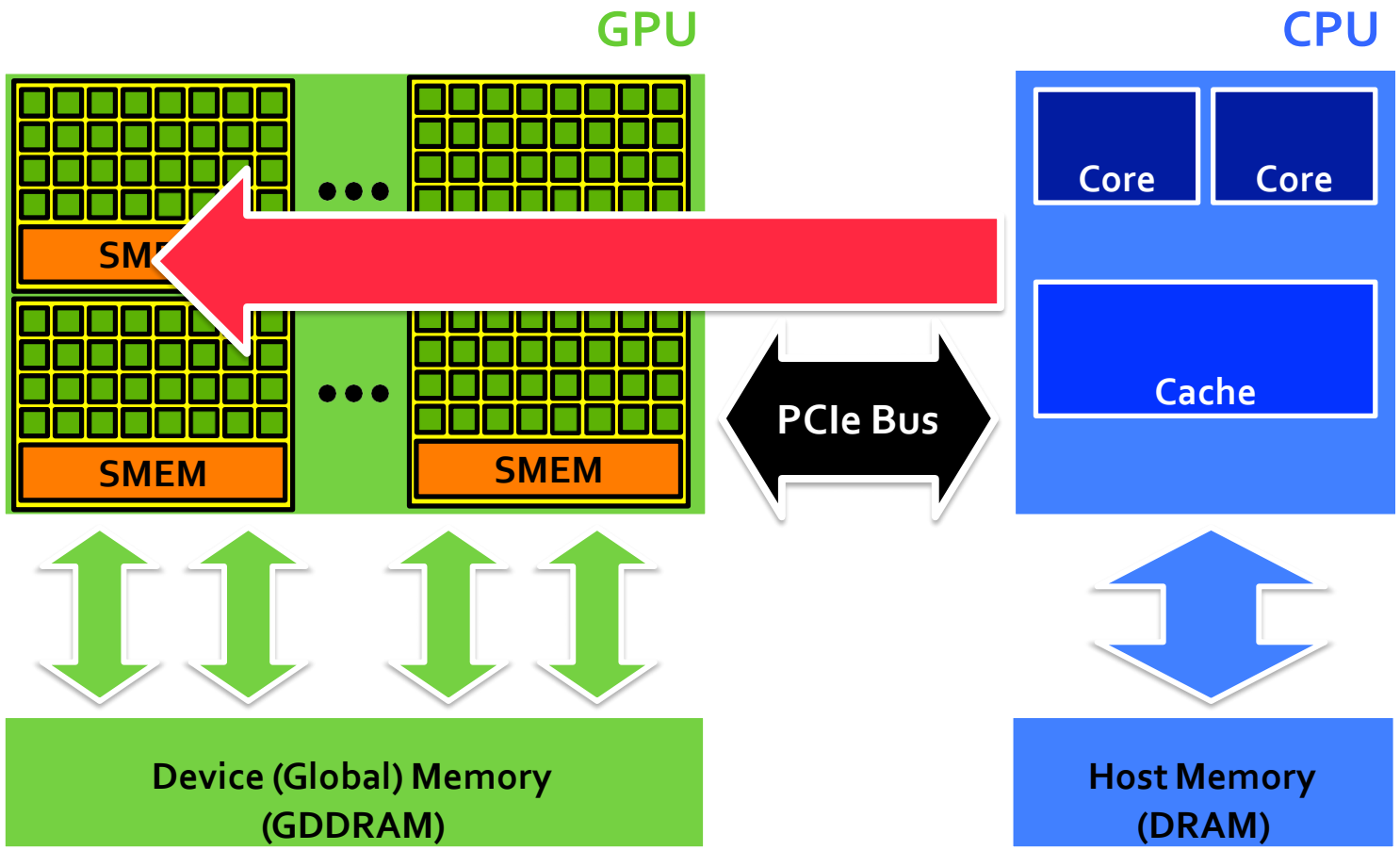
执行过程


GPU
CPU
SMEM
SMEM
SMEM
SMEM
PCIe Bus
Core
Core
Cache
Device (Global) Memory
(GDDRAM)
Host Memory
(DRAM)

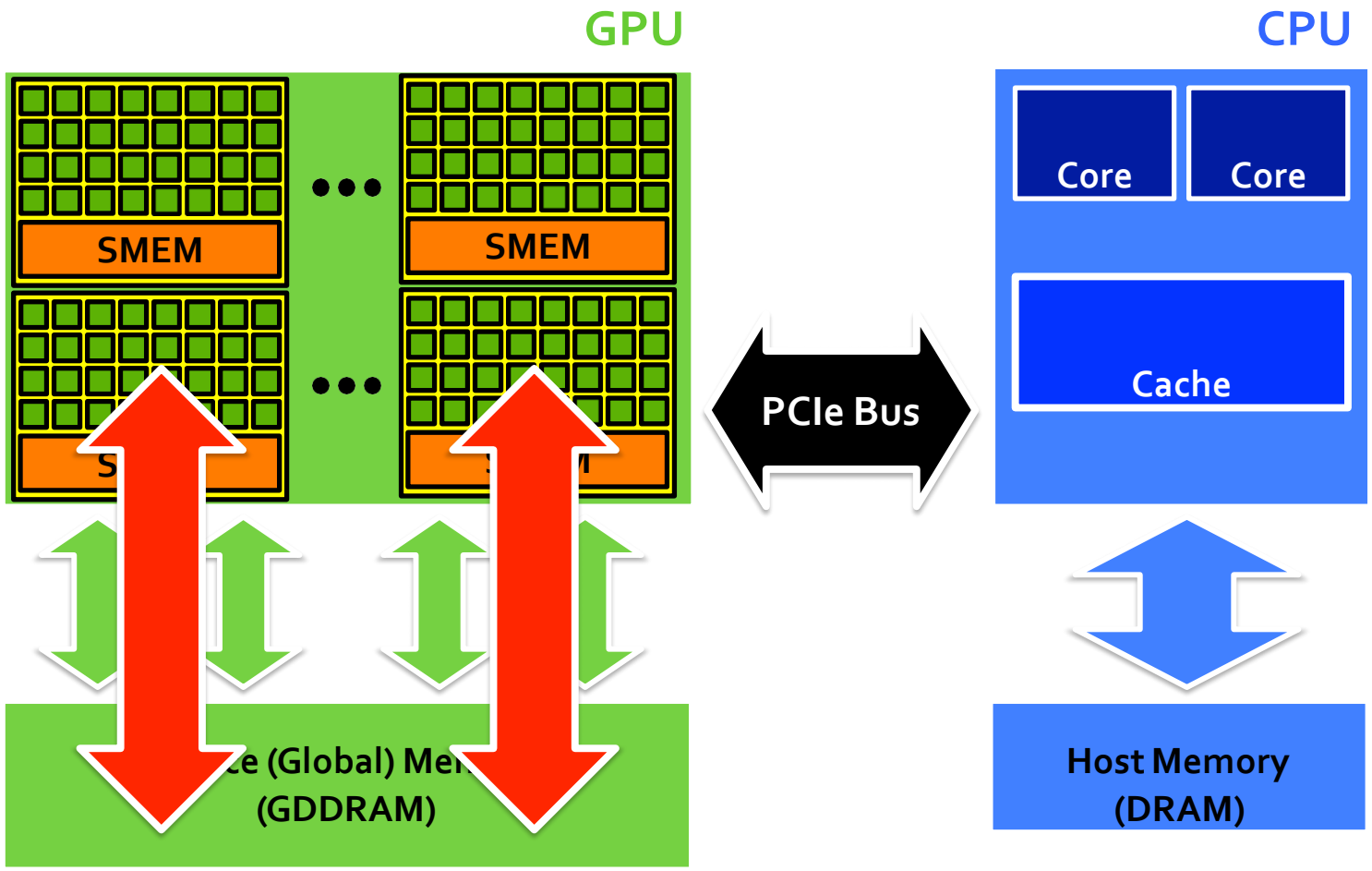
执行过程： copy data to GPU memory



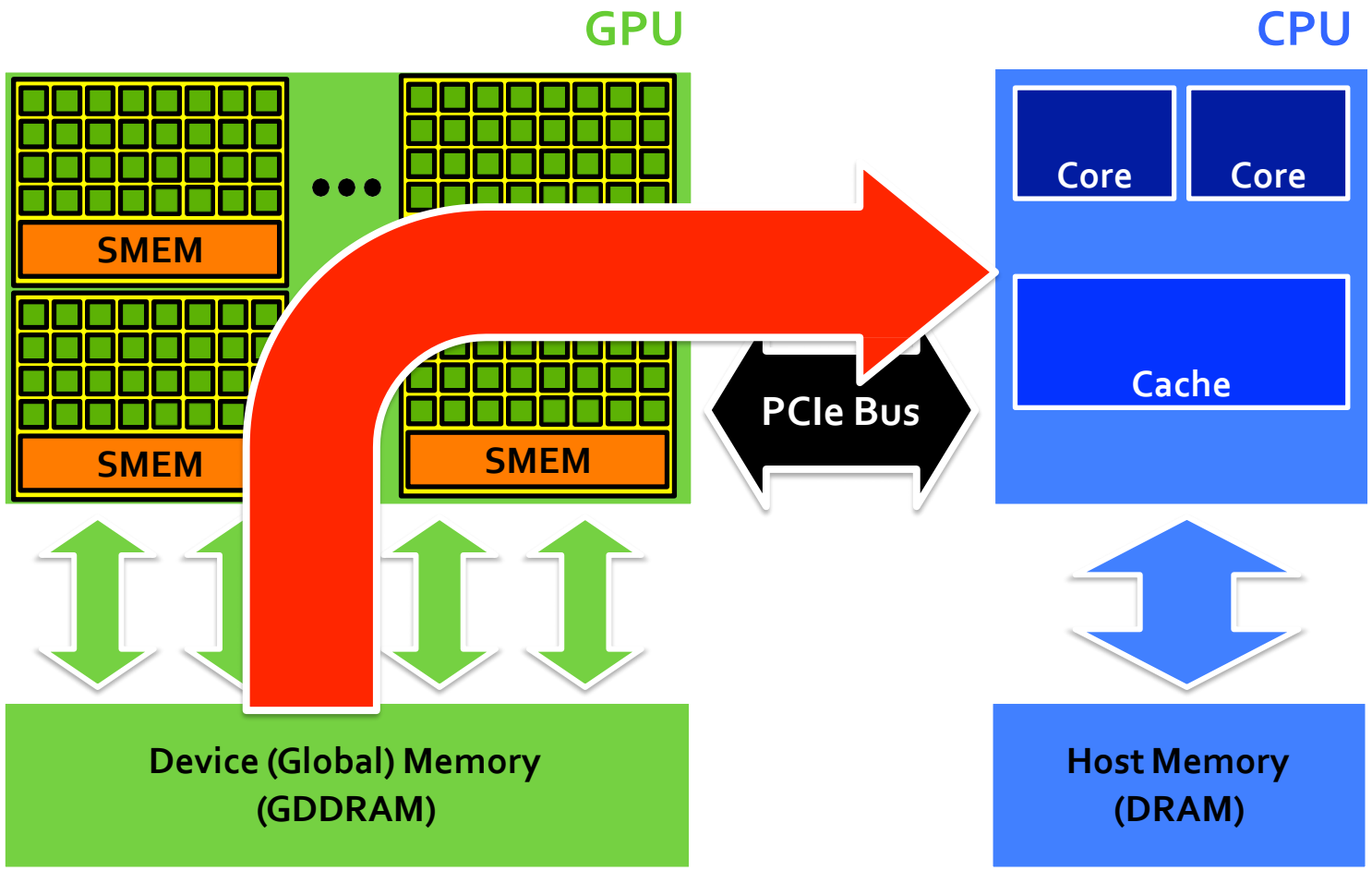
执行过程： launch kernel on GPU



执行过程： execute kernel on GPU

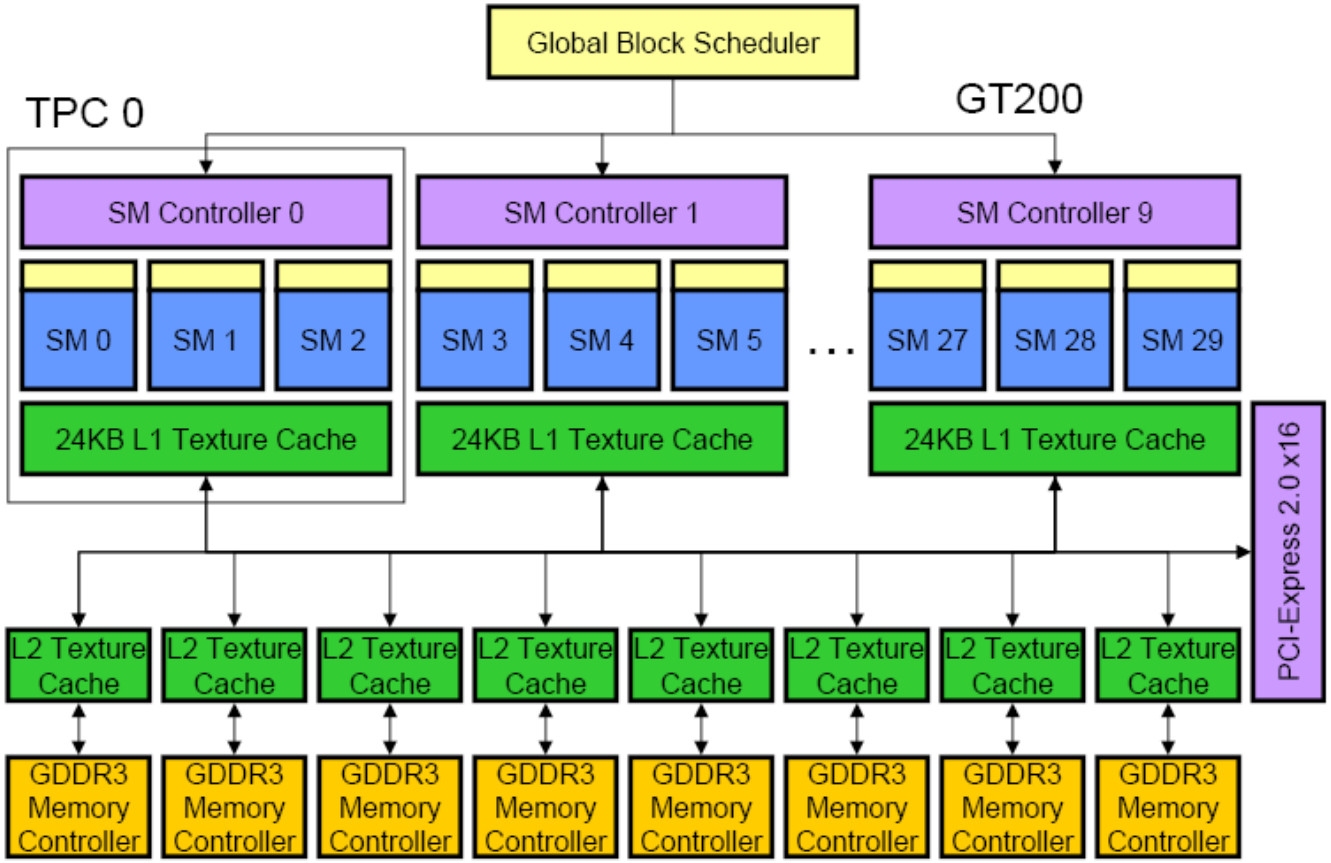


执行过程： copy data to CPU memory

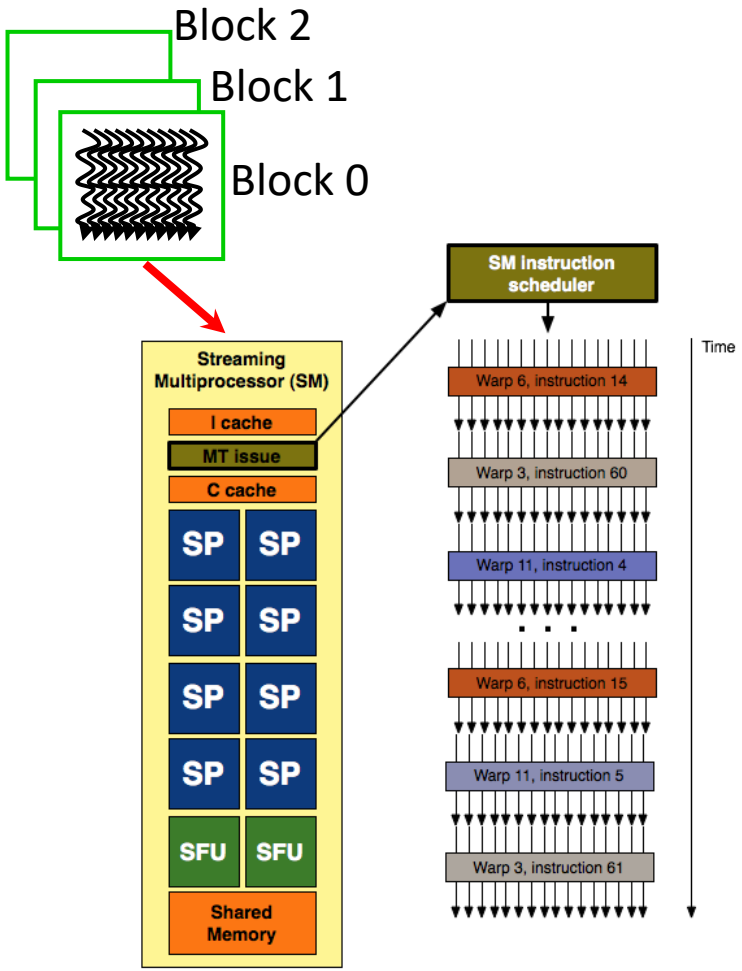


GPU负载分配

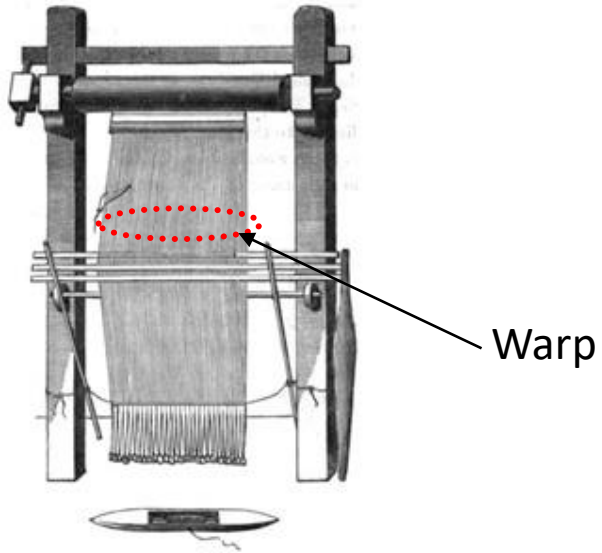
- Global block scheduler
  - 管理thread block级并行
  - 从CPU获得线程组织信息
  - 根据硬件结构分配thread block到SM



并行线程执行

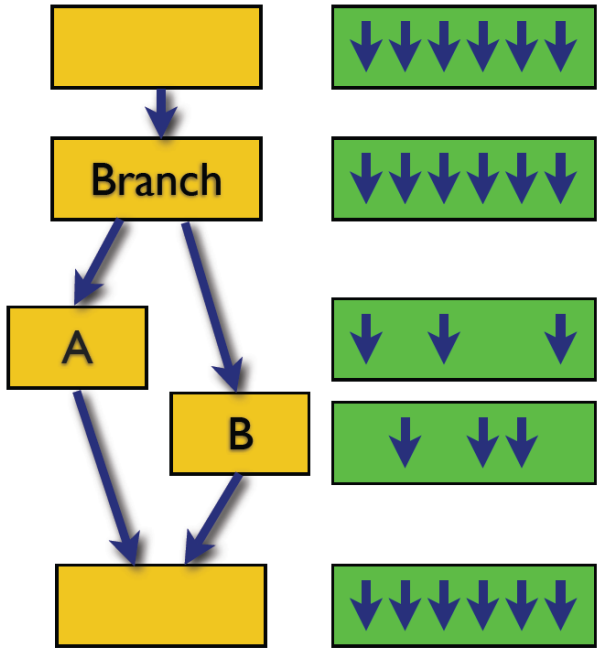


- SM内以warp (即32 threads)为单位并行执行
  - Warp内线程执行同一条指令
  - Half-warp是存储操作的基本单位



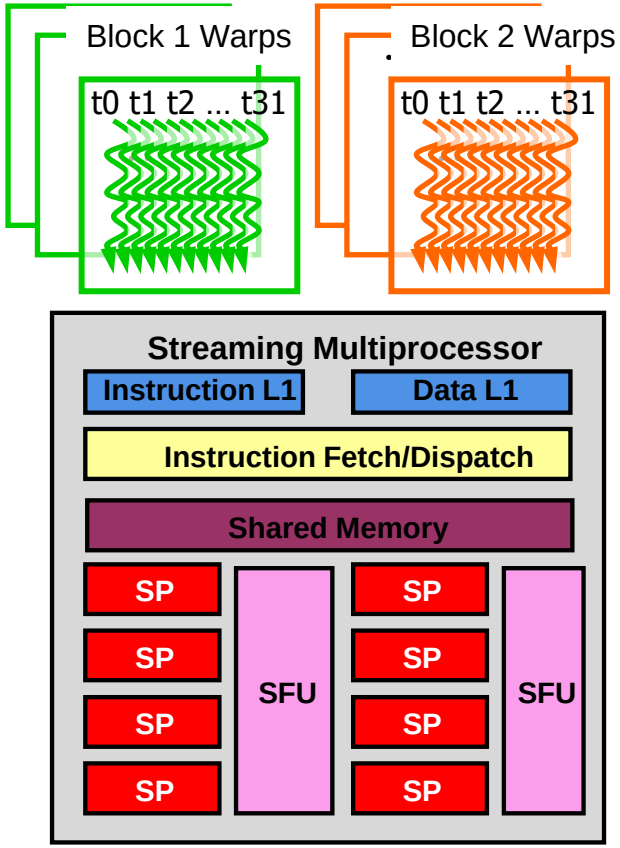
控制流

- 同一warp内的分支语句可能执行不同的指令路径
  - 不同指令路径的线程只能顺序执行
    - 每次执行warp中一条可能的路径
    - N条指令路径→1/N throughput
  - 只需要考虑同一warp即可，不同warp的不同的指令路径不具相关性



Thread Life Cycle

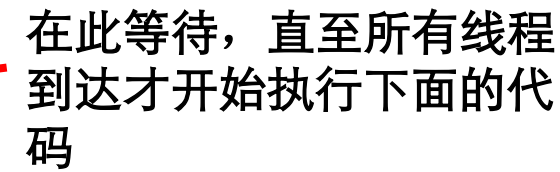
- Grid在GPU上启动
- Thread blocks顺序分配到SM’s
  - 一般SM应有>1 thread block
- SM把线程组织为warps
- SM调度并执行就绪的warp
- Warps和thread blocks 执行结束后释放资源
- GPU继续分发thread blocks



线程同步

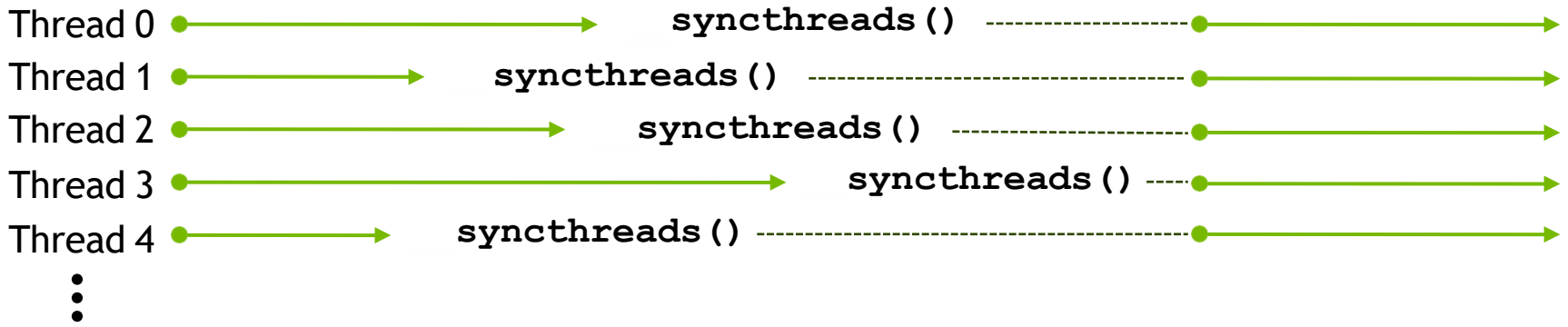
- void __syncthreads();
  - Barrier synchronization
  - 同步thread block之内的所有线程

```
__shared__ float scratch[256];
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID -1];
```



线程同步

- __syncthreads()
- Threads in the block wait until all threads have hit the syncthreads()
- Threads are *only* synchronized within a block



Parallel Dot Product: `dot()`

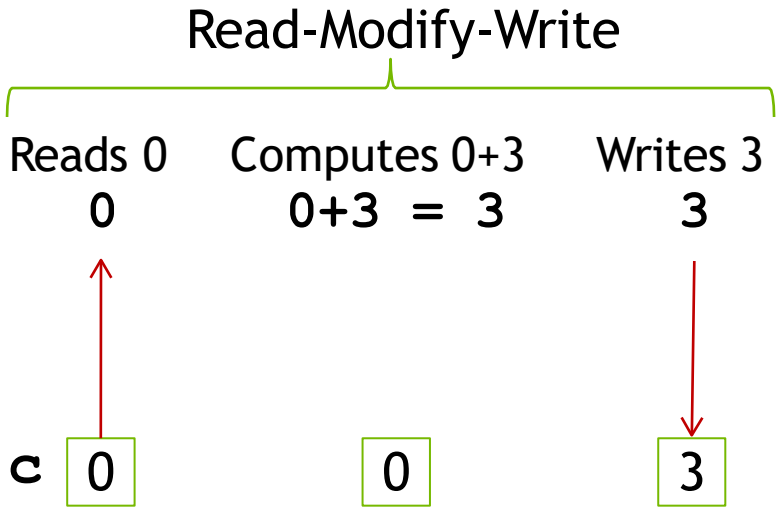
```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    __syncthreads();

    if( 0 == threadIdx.x )
    {   int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```

Global Memory Contention

Block 0
`sum = 3`

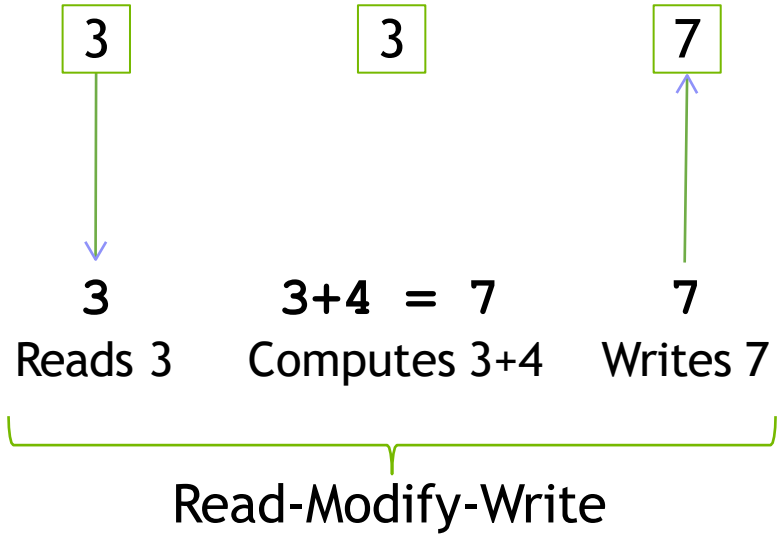
- Read value at address `c`
- Add `sum` to value
- Write result to address `c`

`*c += sum`

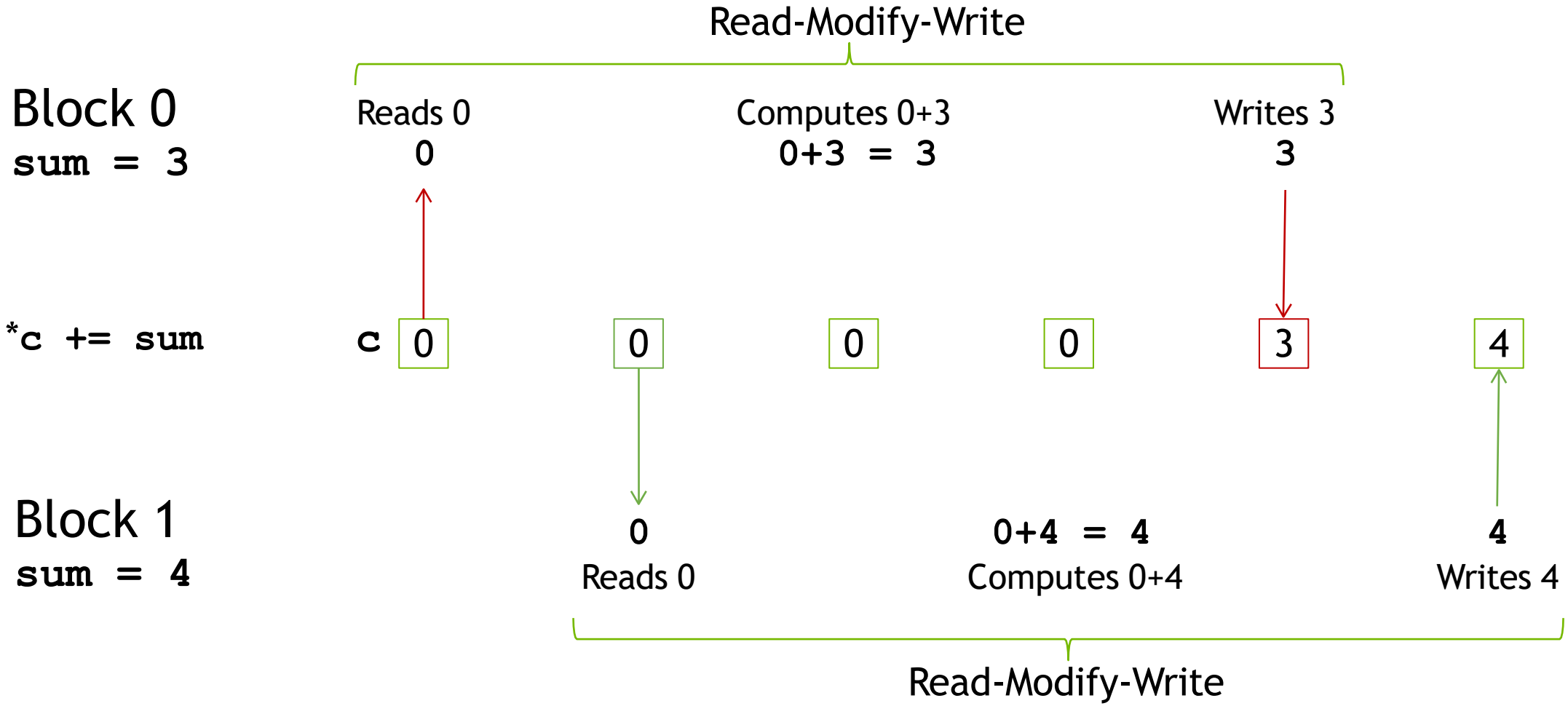


Block 1
`sum = 4`

- Read value at address `c`
- Add `sum` to value
- Write result to address `c`



Global Memory Contention



Atomic Operations

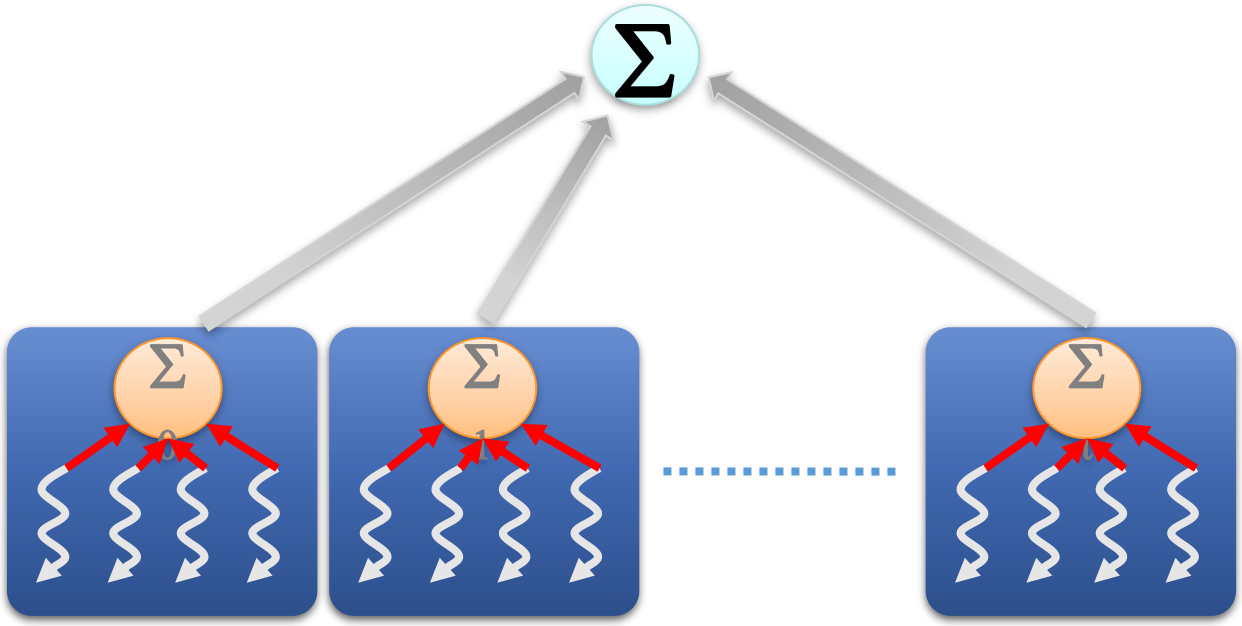
- Terminology: Read-modify-write uninterruptible when *atomic*
- Many *atomic operations* on memory available with CUDA C
  - `atomicAdd()`
  - `atomicSub()`
  - `atomicMin()`
  - `atomicMax()`
  - `atomicInc()`
  - `atomicDec()`
  - `atomicExch()`
  - `atomicCAS()`

Resource Contention

- Atomic operations can create bottlenecks which collapse your parallel program to a serial program and significantly degrade performance
- They imply serialized access to a variable
- Atomics to __shared__ variables are much faster than atomics to global variables
- Don’t synchronize or serialize unnecessarily

```
__global__ void sum(int *input, int
   *result)
{
   atomicAdd(result, input[threadIdx.x]);
}
...
// how many threads will contend
// for exclusive access to result?
sum<<<B,N/B>>>(input,result);
```

Hierarchical Atomics




```
__global__ void sum(int *input, int *result)
{
  ...
  // each thread updates the partial sum
  atomicAdd(&partial_sum,
            input[threadIdx.x]);
  __syncthreads();

  // thread 0 updates the total sum
  if(threadIdx.x == 0)
    atomicAdd(result, partial_sum);
}
```

- Divide & Conquer
  - Per-thread `atomicAdd` to a `__shared__` partial sum
  - Per-block `atomicAdd` to the total sum

Synchronization

- Memory Fences
  - Weak-ordered memory model
    - The order of writes into shared/global/host/peer memory is not necessarily the same as the order of reads made by another thread
    - The order of read operations is not necessarily the same as the order of the read instructions in the code
  - Example
    - Let us have variables `__device__ int X = 1, Y = 2;`

```
__device__ void write() {
    X = 10;
    Y = 20;
}
```

```
__device__ void read() {
    int A = X;
    int B = Y;
}
```

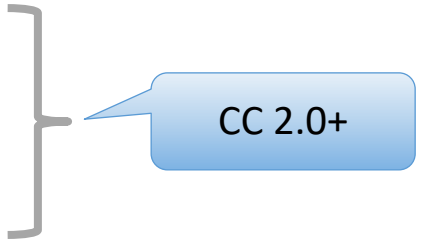
Synchronization

- Memory Fences

```
__threadfence();
__threadfence_block()
__threadfence_system();
```

- Barrier
  - Synchronization between warps in block

```
__syncthreads();
__syncthreads_count(predicate);
__syncthreads_and(predicate);
__syncthreads_or(predicate);
```



Explicit Synchronization

- Synchronize everything
  - cudaDeviceSynchronize ()
  - Blocks host until all issued CUDA calls are complete

- Synchronize w.r.t. a specific stream
  - cudaStreamSynchronize (streamid)
  - Blocks host until all CUDA calls in streamid are complete

- Synchronize using Events
  - Create specific 'Events', within streams, to use for synchronization cudaEventRecord (event, streamid)
  - cudaEventSynchronize (event) cudaStreamWaitEvent (stream, event) cudaEventQuery (event)

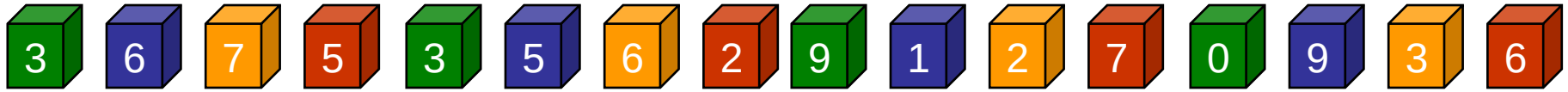
Explicit Synchronization Example

```
Resolve using an event
{
    cudaEvent_t event;
    cudaEventCreate (&event); // create event

    cudaMemcpyAsync ( d_in, in, size, H2D, stream1 ); // 1) H2D copy of new input
    cudaEventRecord (event, stream1); // record event

    cudaMemcpyAsync ( out, d_out, size, D2H, stream2 ); // 2) D2H copy of previous result

    cudaStreamWaitEvent ( stream2, event ); // wait for event in stream1
    kernel <<< , , , stream2 >>> ( d_in, d_out ); // 3) must wait for 1 and 2

    asynchronousCPUmethod ( … ) // Async GPU method

}
```

示例

统计数据

- 统计数组中数字6出现的次数
- 假设输入数据存在数组，共16个元素，每一线程负责4数组元素，1 block in grid, 1 grid



threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

Known as a cyclic data distribution

CUDA伪代码

MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with "6"

Compute local result

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Synchronize after completion

Copy ***device*** output to host

DEVICE FUNCTION:

Compare current element and "6"

Return 1 if same, else 0

主程序

MAIN PROGRAM:

Initialization (OMIT)

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute (int *in_arr, int *out_arr);
int main(int argc, char **argv){
        int *in_array, *out_array;
        int sum = 0;
        /* initialization */ …
        outer_compute(in_array, out_array);
        for (int i=0; i<BLOCKSIZE; i++) {
          sum+=out_array[i];
        }
        printf ("Result = %d\n",sum);
}
```

Host函数

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Synchronize after completion

Copy ***device*** output to host

```
__host__ void outer_compute (int *h_in_array, int
  *h_out_array) {
  int *d_in_array, *d_out_array;

  cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));
  cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));
  cudaMemcpy(d_in_array, h_in_array,
        SIZE*sizeof(int),      cudaMemcpyHostToDevice);
  … do computation …
  cudaMemcpy(h_out_array,d_out_array,
        BLOCKSIZE*sizeof(int),
        cudaMemcpyDeviceToHost);
```

Host函数:调用Global Function

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Synchronize after completion

Copy ***device*** output to host

```
__host__ void outer_compute (int *h_in_array, int *h_out_array) {
  int *d_in_array, *d_out_array;
   cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));
   cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));
   cudaMemcpy(d_in_array, h_in_array,        SIZE*sizeof(int),
          cudaMemcpyHostToDevice);
  compute<<<(1,BLOCKSIZE)>>> (d_in_array, d_out_array);
  cudaThreadSynchronize();
  cudaMemcpy(h_out_array, d_out_array,
          BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}
```

Global Function

GLOBAL FUNCTION:

Thread scans subset of array elements

Call *device* function to compare with "6"

Compute local result

```
__global__ void compute(int *d_in,int *d_out){
  d_out[threadIdx.x] = 0;
  for (int i=0; i<SIZE/BLOCKSIZE; i++)   {
      int val = d_in[i*BLOCKSIZE + threadIdx.x];
     d_out[threadIdx.x] += compare(val, 6);
   }
}
```

Device Function

DEVICE FUNCTION:

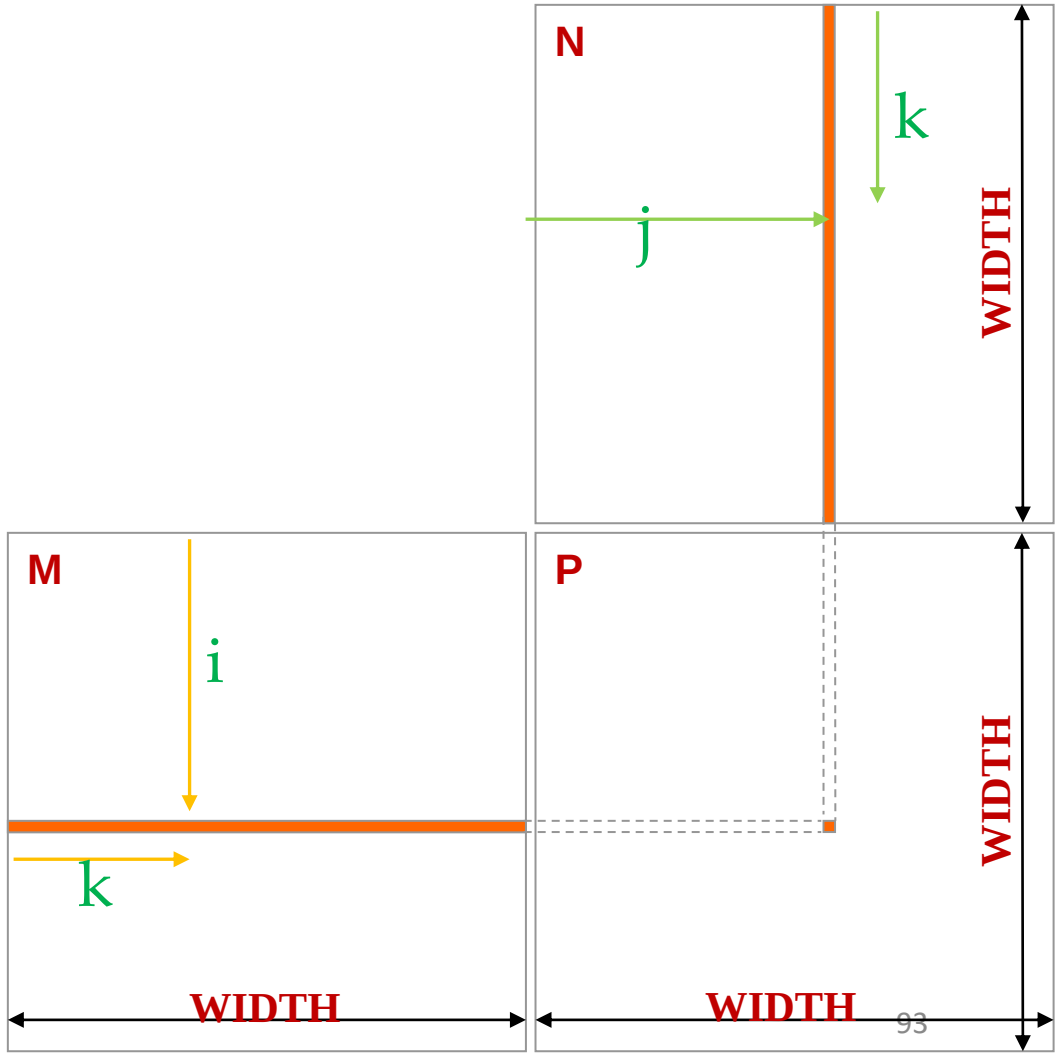
Compare current element and "6"

Return 1 if same, else 0

```
__device__ int compare(int a, int b) {
        if (a == b) return 1;
        return 0;
}
```

矩阵相乘：CPU实现

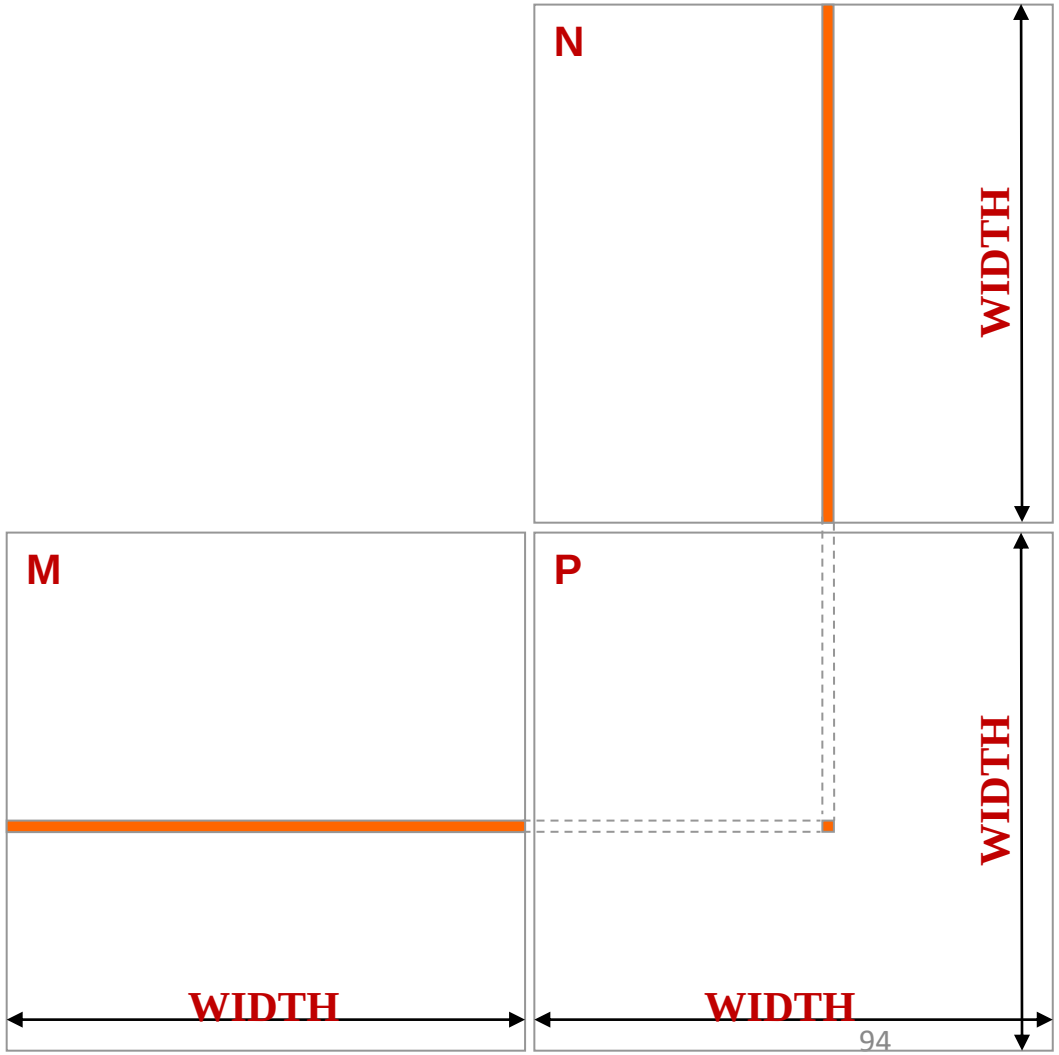
```
// Matrix multiplication on the (CPU) host
   in double precision
void MatrixMulOnHost(float* M, float* N,
   float* P, int Width)
{
     for (int i = 0; i < Width; ++i)
         for (int j = 0; j < Width; ++j) {
             double sum = 0;
             for (int k = 0; k < Width; ++k)
   {
                 double a = M[i * width + k];
                 double b = N[k * width + j];
                 sum += a * b;
             }
             P[i * Width + j] = sum;
         }
}
```



矩阵相乘- CUDA程序实现

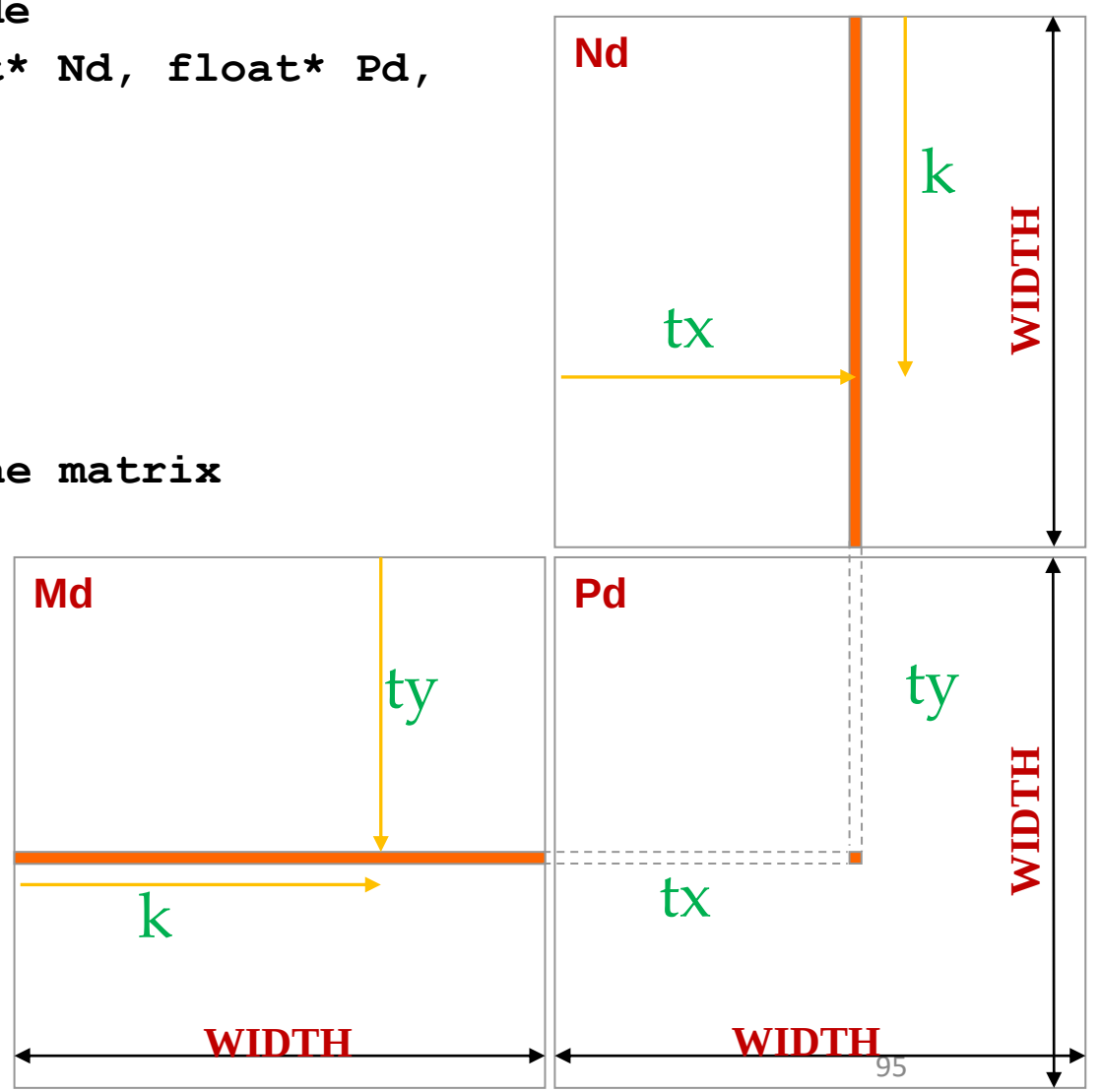
- P = M * N （长宽均为WIDTH）
- 计算策略

  每个线程计算矩阵P中的一个元素



kernel函数

```
// Matrix multiplication kernel – per thread code
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd,
   int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)  {
         float Melement = Md[ty * Width + k];
         float Nelement = Nd[k * Width + tx];
         Pvalue += Melement * Nelement;
    }
    Pd[ty * Width + tx] = Pvalue;
}
```



主函数

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width){
     int size = Width * Width * sizeof(float);
     float* Md, Nd, Pd;
    …
1. // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
    cudaMalloc(&Pd, size); // Allocate P on the device

2.  // Setup the execution configuration
    dim3 dimBlock(Width, Width);
    dim3 dimGrid(1, 1);
    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

3.  // Read P from the device
     cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
     // Free device matrices
     cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

Thrust Parallel Algorithms Library

What is Thrust?

- Thrust is the CUDA analog of the Standard Template Library (STL) of C++.
- High-Level Parallel Algorithms Library
- Parallel Analog of the C++ Standard Template Library (STL)
- Performance-Portable Abstraction Layer
- Productive way to program CUDA

Thrust

- It comes with any installation of CUDA 4.2 and above and features:
  - Dynamic data structures
  - An encapsulation of GPU/CPU communication, memory management, and other low-level tasks.
  - High-performance GPU-accelerated algorithms such as sorting and reduction

Example

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());
    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Productivity

- Containers
  - `host_vector`
  - `device_vector`
- Memory Mangement
  - Allocation
  - Transfers
- Algorithm Selection
  - Location is implicit

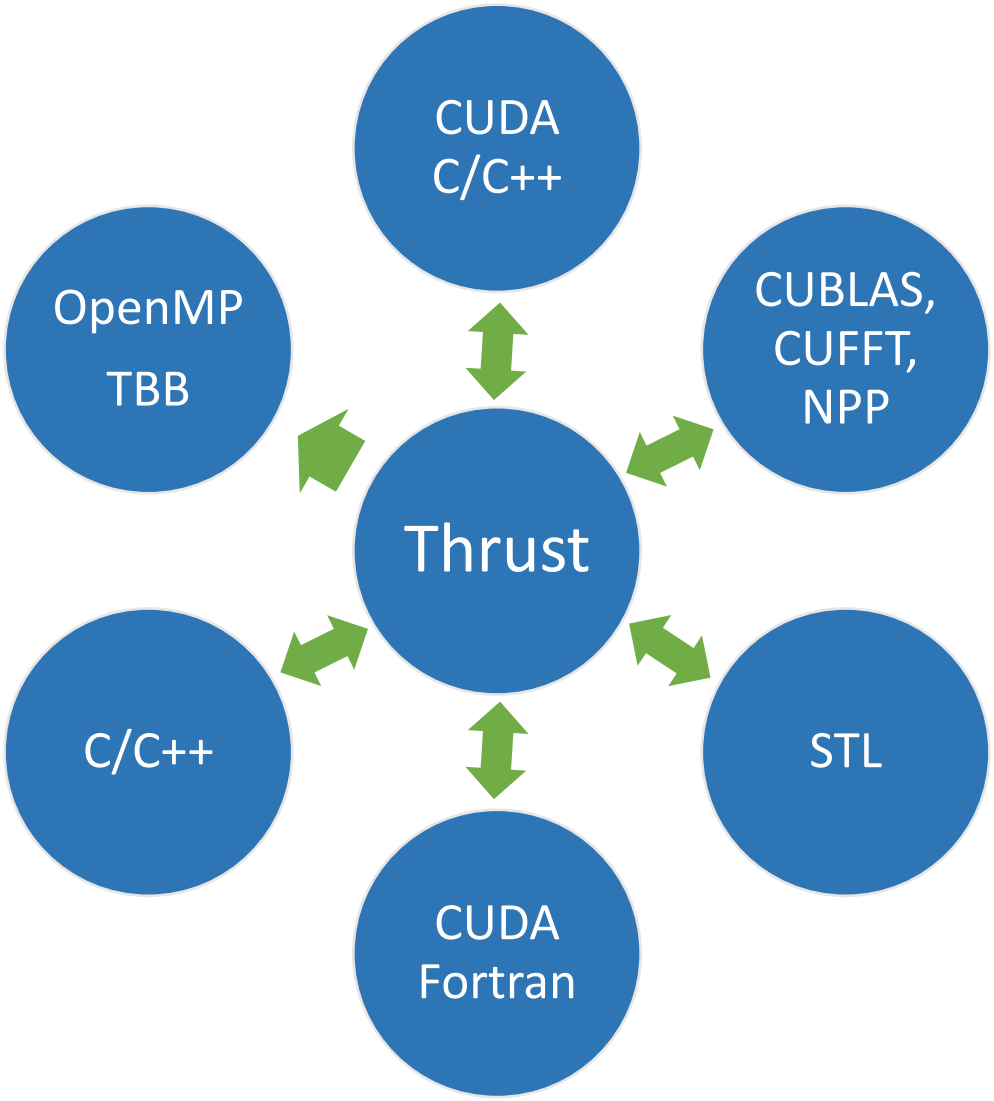
```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host data to device memory
thrust::device_vector<int> d_vec = h_vec;

// write device values from the host
d_vec[0] = 27;
d_vec[1] = 13;

// read device values from the host
int sum = d_vec[0] + d_vec[1];

// invoke algorithm on device
thrust::sort(d_vec.begin(), d_vec.end());

// memory automatically released
```

Productivity

- Large set of algorithms
  - ~75 functions
  - ~125 variations
- Flexible
  - User-defined types
  - User-defined operators

Algorithm	Description
`reduce`	Sum of a sequence
`find`	First position of a value in a sequence
`mismatch`	First position where two sequences differ
`inner_product`	Dot product of two sequences
`equal`	Whether two sequences are equal
`min_element`	Position of the smallest value
`count`	Number of instances of a value
`is_sorted`	Whether sequence is in sorted order
`transform_reduce`	Sum of transformed sequence

Interoperability



Multiple Backend Systems

Host Systems
THRUST_HOST_SYSTEM_CPP
THRUST_HOST_SYSTEM_OMP
THRUST_HOST_SYSTEM_TBB

Device Systems
THRUST_DEVICE_SYSTEM_CUDA
THRUST_DEVICE_SYSTEM_OMP
THRUST_DEVICE_SYSTEM_TBB

Backend System Options

```
thrust::omp::vector<float> my_omp_vec(100);
thrust::cuda::vector<float> my_cuda_vec(100);

...

// reduce in parallel on the CPU
thrust::reduce(my_omp_vec.begin(), my_omp_vec.end());

// sort in parallel on the GPU
thrust::sort(my_cuda_vec.begin(), my_cuda_vec.end());
```

Mix different backends freely within the same app

Containers

- Examples of containers include: vector、deque、list、tack、queue、priority queue、set、multiset、map、multimap、biset
- Thrust only implements vectors, but it’s still compatible with the rest of STL’s template classes.
- Declaring vectors:
  - thrust::device_vector<T> D; creates a vector D with entries of data type T on the device.
  - The analogous declaration for host vectors is thrust:: host_vector<T> H;.
- An object D of the vector template class includes the following features:
  - A dynamic linear array of elements of type T.
  - Two iterators:
    - D.begin()
    - D.end()

Vector

```
int main(void){
  thrust::device_vector <int> D(10, 1); // initialize all ten integers of a device_vector to 1

  thrust::fill(D.begin(), D.begin() + 7, 9); //set the first seven elements of a vector to 9

  thrust::host_vector<int> H(D.begin(), D.begin() + 5); //initialize a host_vector with the first
five elements of D

  thrust::sequence(H.begin(), H.end()); //set the elements of H to 0, 1, 2, 3, ...

  thrust::copy(H.begin(), H.end(), D.begin()); //copy all of H back to the beginning of D

  for(int i = 0; i < D.size(); i++)
    std::cout << "D[" << i << "] = " << D[i] << std::endl;

  return 0;
}
```

Basic iterators

- An iterator is a pointer with a C++ wrapper around it. The wrapper contains additional information, such as whether the vector is stored on the host or the device.

```
// allocate device vector
thrust::device_vector<int> d_vec(4);
thrust::device_vector <int > :: iterator begin = d_vec.begin ( ) ;
thrust::device_vector <int > :: iterator end = d_vec.end ( ) ;
// [begin, end) pair defines a sequence of 4 elements
int length=end-begin; // compute the length of the vector
end= d_vec.begin()+ 3; //define a sequence of 3 elements
```

Wrap pointers to make iterators

```
#include <thrust/fill.h>
#include <cuda.h>
#include <iostream>
int main(){
    int N = 10;

    // raw pointer to device memory
    int* rawptr;
    cudaMalloc((void **) &rawptr, N*sizeof(int));

    //wrap raw pointer with a device ptr
    thrust::device_ptr<int> devptr(rawptr);

    //use device ptr in thrust algorithms
    thrust::fill(devptr, devptr + N, (int) 0);
    devptr[0] = 1;
    cudaFree(rawptr);
    return 0;
}
```

Unwrap iterators to extract pointers

```
#include <thrust/device_vector.h>
#include <iostream>
#include <cuda.h>

void __global__ myKernel(int N, int *ptr){
}
int main(){
  int N = 512;
  thrust::device_vector<int> d_vec(4); //allocate device vector

  int *ptr=thrust::raw_pointer_cast(&d_vec[0]); //obtain raw pointer to device vector's
memory

  myKernel<<<N/256, 256>>>(N, ptr); //use pointer in a CUDA C kernel
  // Note: ptr cannot be dereferenced on the host!
  return 0;
}
```

Array of Structures vs. Structure of Arrays

Naïve approach, Array of Structures:

```
struct record {
   int key;
   int value;
   int flag;
};
record *d_records;
cudaMalloc((void** ) &d_records, ...);
```

Better idea, Structure of Arrays:

```
struct SoA {
   int * keys;
   int * values;
   int * flags;
};
SoA d_SoA_data;
cudaMalloc((void** ) &d_SoA_data.keys, ...);
cudaMalloc((void** ) &d_SoA_data.values, ...);
cudaMalloc((void** ) &d_SoA_data.flags, ...);
```

Example: SoA vs. AoS

```
__global__ void bar(record *AoS_data, SoA SoA_data)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;

   int key = AoS_data[i].key;
   // waste of bandwidth (uses only 1/3 of copied data)

   int better_key = SoA_data.keys[i];
   // more efficient, since keys are in a contiguous array
}
```

Reminder: all threads execute kernel code at the same time

zip iterator

- A zip iterator is a pointer associated with a vector of tuples.

```
int main(){
  thrust::device_vector<int> int_v(3);
  int_v[0] = 0; int_v[1] = 1; int_v[2] = 2;
  thrust::device_vector<float> float_v(3);
  float_v[0] = 0.0; float_v[1] = 1.0; float_v[2] = 2.0;
  thrust::device_vector<char> char_v(3);
  char_v[0] = 'a'; char_v[1] = 'b'; char_v[2] = 'c';

  // typedef these iterators for shorthand
  typedef thrust::device_vector<int>::iterator
IntIterator;
  typedef thrust::device_vector<float>::iterator
FloatIterator;
  typedef thrust::device_vector<char>::iterator
CharIterator;
  // typedef a tuple of these iterators
  typedef thrust::tuple<IntIterator, FloatIterator,
CharIterator> IteratorTuple;
  // typedef the zip_iterator of this tuple
  typedef thrust::zip_iterator<IteratorTuple> ZipIterator;
  // finally, create the zip_iterator
  ZipIterator iter(thrust::make_tuple(int_v.begin(),
float_v.begin(), char_v.begin()));

  *iter;   // returns (0, 0.0, 'a')
  iter[0]; // returns (0, 0.0, 'a')
  iter[1]; // returns (1, 1.0, 'b')
  iter[2]; // returns (2, 2.0, 'c')

  thrust::get<0>(iter[2]); // returns 2
  thrust::get<1>(iter[0]); // returns 0.0
  thrust::get<2>(iter[1]); // returns 'b'
  // iter[3] is an out-of-bounds error
}
```

Algorithms: Transformations

- A transformation is the application of a function to each element within a range of elements in a vector. The results are stored as a range of elements in another vector.
- Examples:
  - thrust::fill()
  - thrust::sequence()
  - thrust::replace()
  - thrust::transform()

Transformations

```
int main(void) {
  // allocate three device_vectors with 10 elements
  thrust::device_vector <int> X(10);
  thrust::device_vector <int> Y(10);
  thrust::device_vector <int> Z(10);

  thrust::sequence(X.begin(), X.end()); // initialize X to 0,1,2,3, ....

  thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>()); // compute Y = -X

  thrust::fill(Z.begin(), Z.end(), 2); // fill Z with twos

  thrust::replace(Y.begin(), Y.end(), 1, 10); // replace all the ones in Y with tens

  thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout, "\n")); // print Y
  return 0;
}
```

Reductions

```
int main(){
  int sum;
  thrust::device_vector <int> D(10, 1);
  //The third argument is the starting value of the reduction. The 4th argument is the binary operation that defines
the kind of reduction
  sum = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());
  sum = thrust::reduce(D.begin(), D.end(), (int) 0);
  sum = thrust::reduce(D.begin(), D.end());

  std::cout << "sum = " << sum << "\n";

  thrust::device_vector <int> vec(5,0);
  vec[1] = 1; vec[3] = 1; vec[4] = 1; // put three 1s in a device vector
  int result = thrust::count(vec.begin(), vec.end() , 1);
  std::cout << "result = " << result << "\n"; // result is three
  return 0;
}
```

Scans

A scan, also called a prefix-sum, applies a function to multiple sub-ranges of a vector and returns the result in a vector of the same size. The default function is addition

```
#include <thrust/scan.h>
#include <thrust/device_vector.h>
#include <iostream>
int main(){
  int data[6] = {1, 0, 2, 2, 1, 3};
  thrust::inclusive_scan(data, data + 6, data);

 /* data[0] = data[0]
  * data[1] = data[0] + data[1]
  * data[2] = data[0] + data[1] + data[2]
  * ...
  * data[5] = data[0] + data[1] + ... + data[5]
  */

  // data is now {1, 1, 3, 5, 6, 9}
  thrust::exclusivescan(data.begin( ), data.end() , data.end()); //in- place scan

  // data is now {0 , 1 , 1 , 3 , 5 , 6} 16
  /* data [ 0 ] = 0
   * data [ 1 ] = data [ 0 ]
   * data [ 2 ] = data [ 0 ] + data [ 1 ]
   * . . .
   * data [ 5 ]= data [ 0 ] + data [ 1 ] + . . . + data [ 4 ]
}
```

Reordering

- The “Reordering” utilities provides subsetting and partitioning tools:
  - thrust::copy if(): copy the elements that make some logical function return true.
  - thrust::partition(); reorder a vector such that values returning true precede values returning false.
  - thrust::remove() and remove if(): remove elements that return false.
  - thrust::unique(): remove duplicates in a vector.

Partitions

```
#include <thrust/partition.h>
#include <iostream>
struct is_even{
  __host__ __device__ bool operator()(const int &x){
    return (x % 2) == 0;
  }
};
int main(){
  int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  const int N = sizeof(A)/sizeof(int);
  thrust::partition(A, A + N, is_even());
  // A is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
  for(int i = 0; i < 10; i++){ printf("A[%d] = %d\n", i, A[i]); }

  return 0;
}
```

Sorting

```
#include <thrust/functional.h>
#include <thrust/sort.h>
#include <iostream>

int main(){
  const int N = 6;
  int A[N] = {1, 4, 2, 8, 5, 7};
  thrust::sort(A, A+N); // A is now {1, 2, 4, 5, 7, 8}
  int keys[N] = { 1, 4, 2, 8, 5, 7};
  char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
  thrust::sort_by_key(keys , keys + N, values);
  // keys is now { 1, 2, 4, 5, 7, 8}
  // values is now {'a', 'c', 'b', 'e', 'f', 'd'}

  int A2[N] = {1, 4, 2, 8, 5, 7};
  thrust::stable_sort(A2, A2 + N, thrust::greater<int>());  // A2 is now {8, 7, 5, 4, 2, 1}

  return 0;
}
```