Parallel Programming Principle and Practice

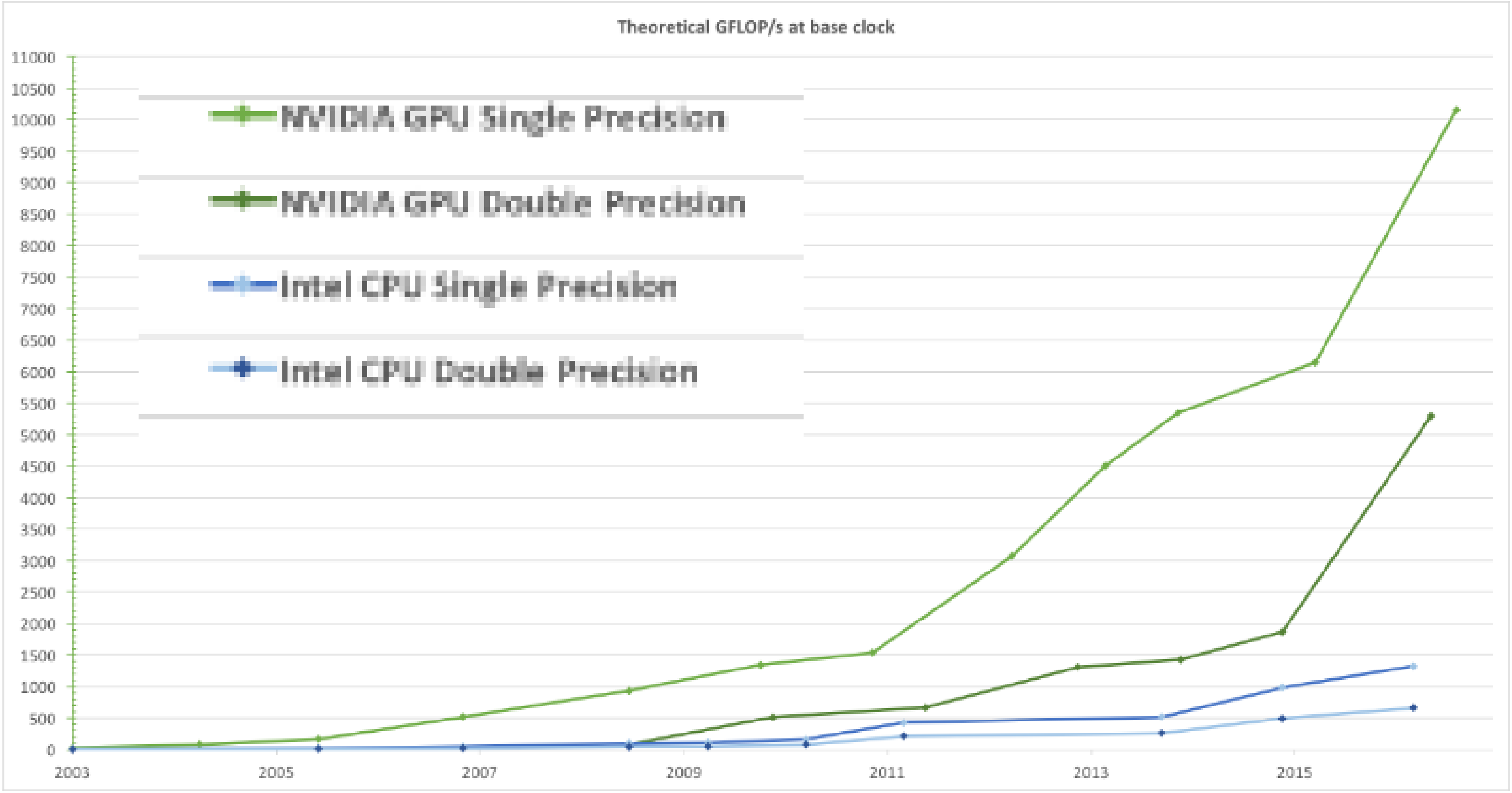
Lecture 8 — Introduction to GPGPUs and CUDA Programming Model

Outline

Graphic Processor Unit

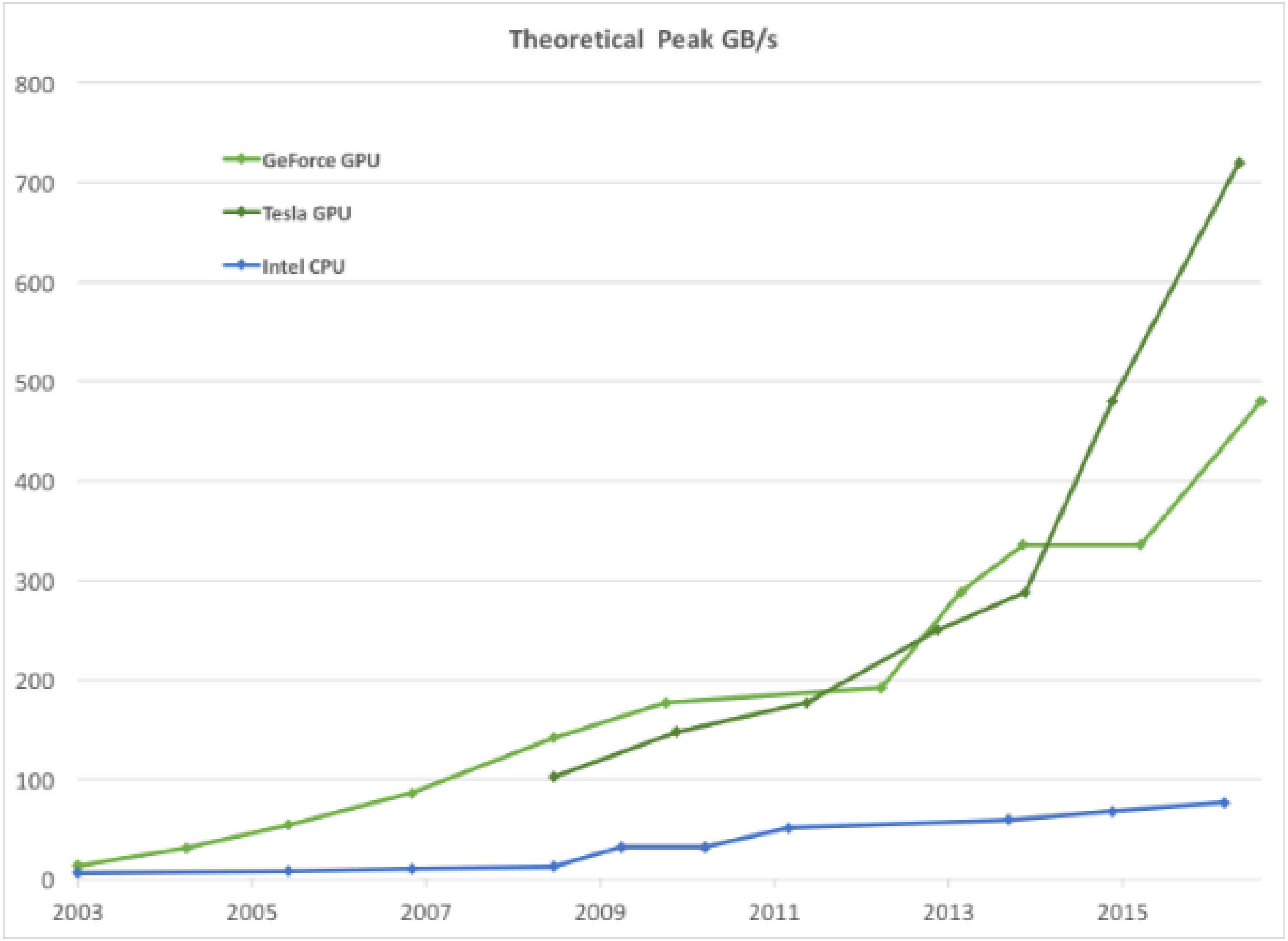
- Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable GPU has evolved into
  - a highly parallel, multithreaded, manycore processor with tremendous computational horsepower
  - very high memory bandwidth
- the GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control

Floating-Point Operations per Second for the CPU and GPU



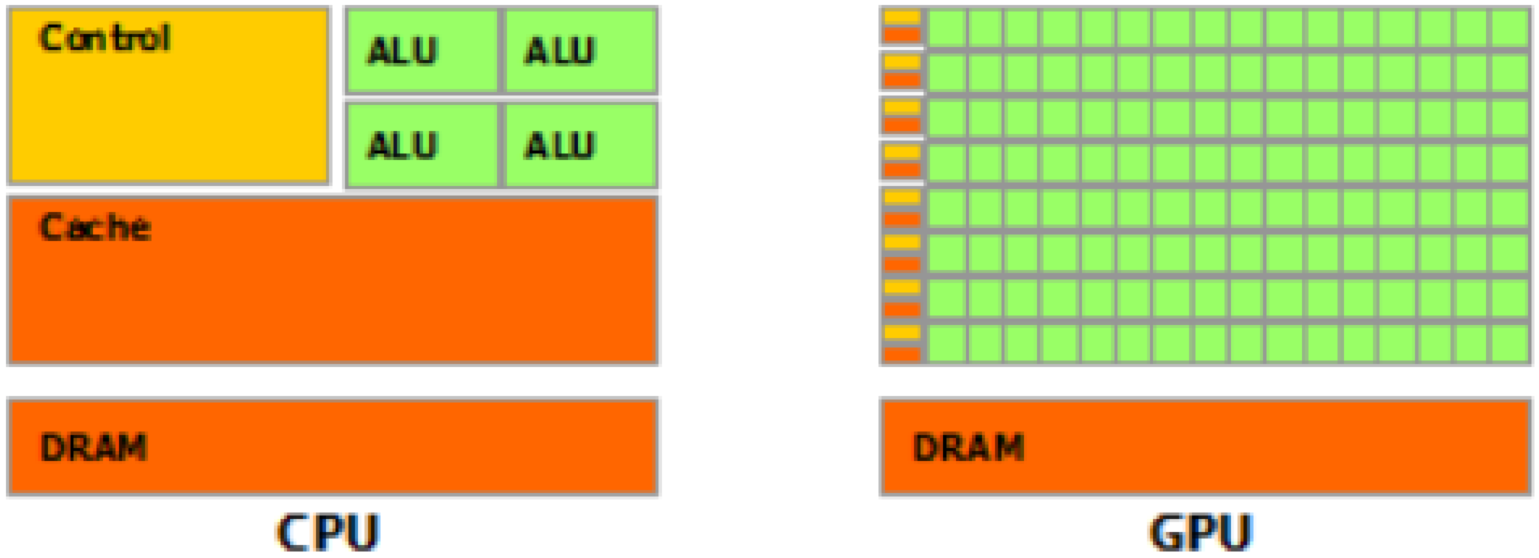
華中科技大學

Memory Bandwidth for the CPU and GPU



The GPU Devotes More Transistors to Data Processing

- data-parallel computations - the same program is executed on many data elements in parallel
- high arithmetic intensity - the ratio of arithmetic operations to memory operations
- same program is executed for each data element
- a lower requirement for sophisticated flow control

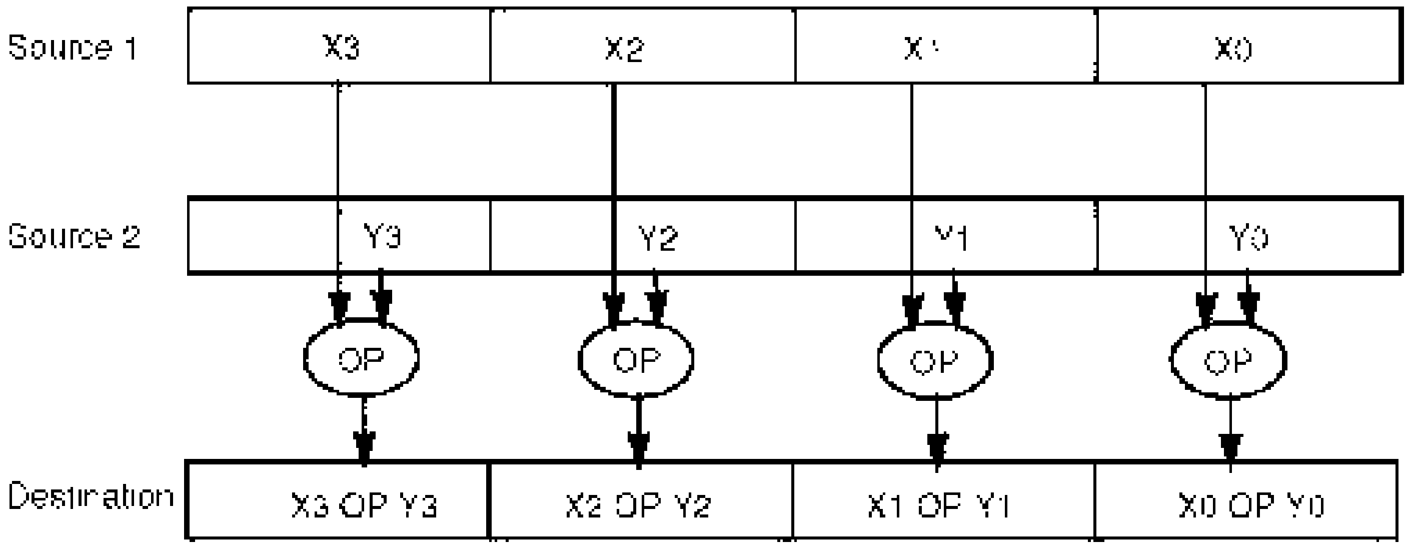


CUDA

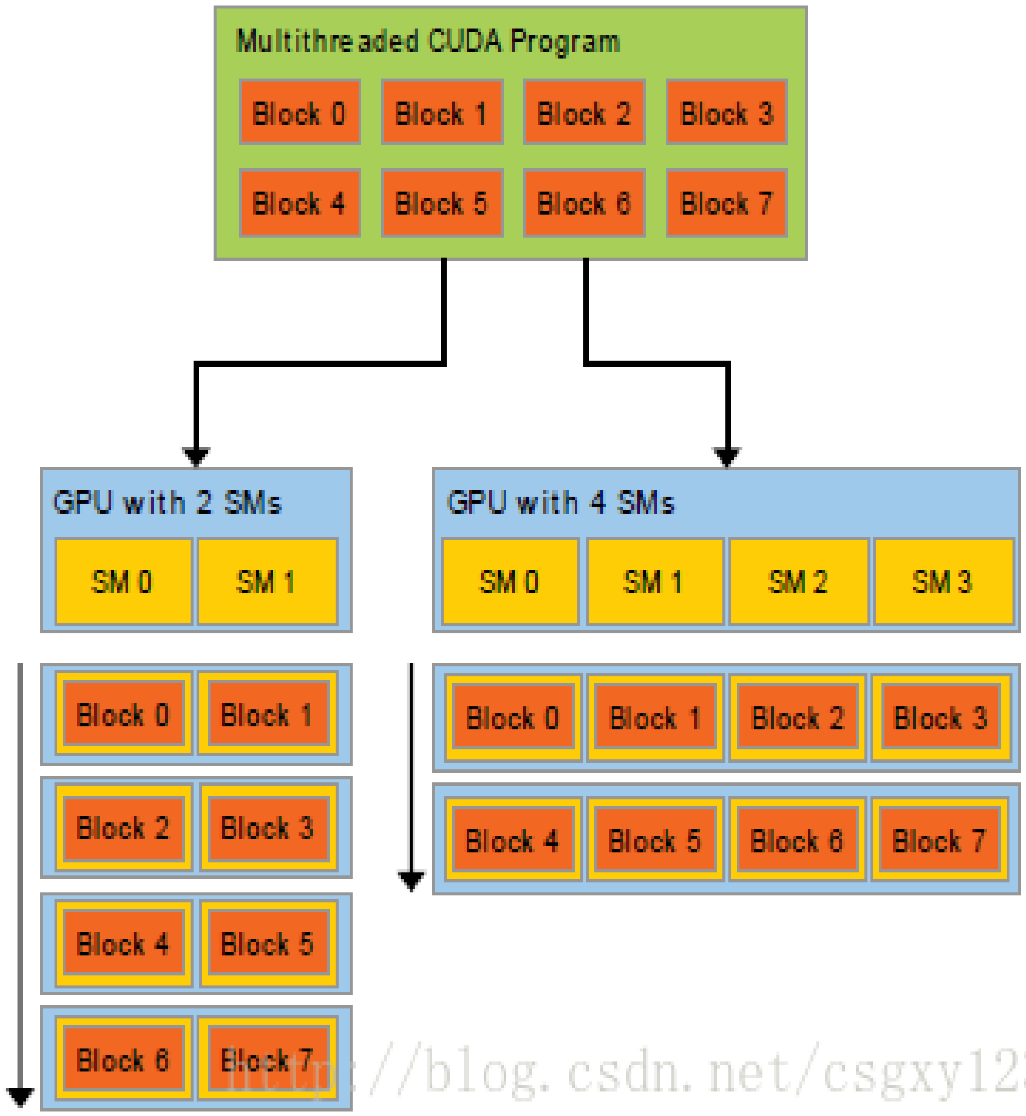
- In November 2006, NVIDIA introduced CUDA®, a general purpose parallel computing platform and programming model
  - leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU
- CUDA comes with a software environment that allows developers to use C as a high-level programming language
- other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, DirectCompute, OpenACC

Cuda Goals: SIMD Programming

- Hardware architects love SIMD, since it permits a very space and energy-efficient implementation
- However, standard SIMD instructions on CPUs are inflexible, and difficult to use, difficult for a compiler to target
- The Cuda Thread abstraction will provide programmability at the cost of additional hardware



Cuda Goals: Scalability



- A GPU is built around an array of Streaming Multiprocessors (SMs)
- A multithreaded program is partitioned into blocks of threads that execute independently from each other
- a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors

Cuda Goals: Scalability

- three key abstractions
  - a hierarchy of thread groups
  - shared memories
  - barrier synchronization
- partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads
- each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block

Outline

- Introduction to GPGPUs and Cuda Programming Model
- Programming Model
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compute Capability
- Mapping Cuda to Nvidia GPUs

Programming Model

- CUDA C define C functions, called **kernels**,
  - when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions
- A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>> execution configuration syntax
- Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in **threadIdx** variable

Kernels

- adds two vectors A and B of size N and stores the result into vector C

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

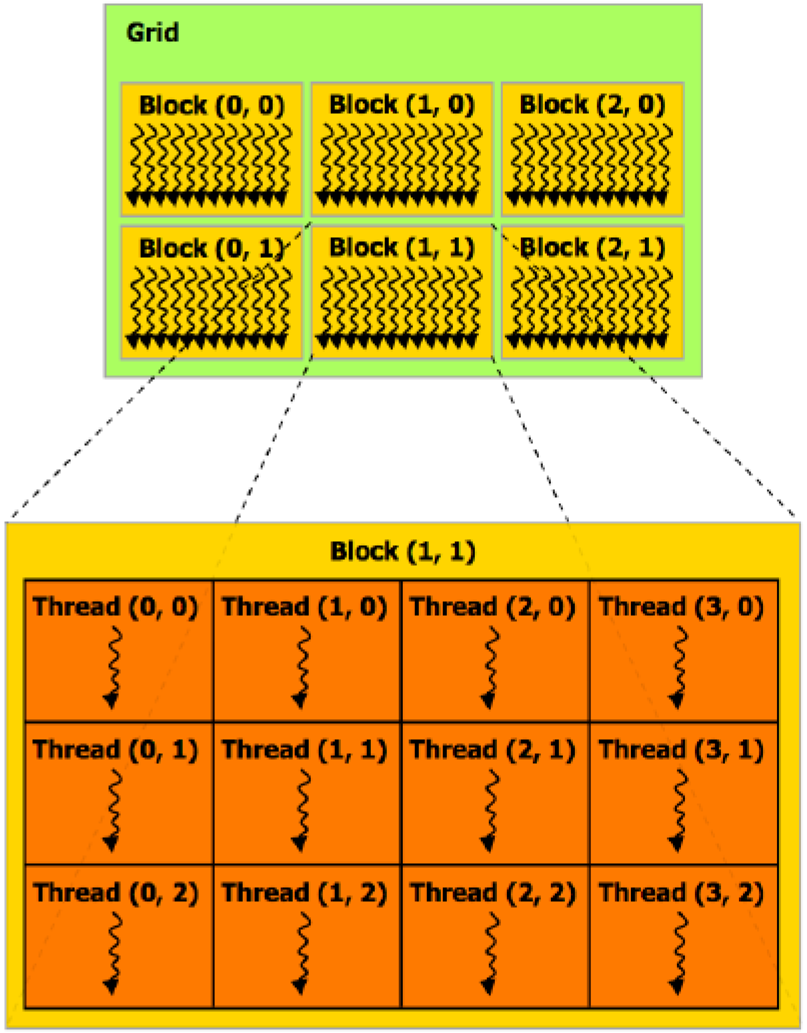
- each of the N threads that execute VecAdd() performs one pair-wise addition

Outline

- Introduction to GPGPUs and Cuda Programming Model
- Programming Model
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compute Capability
- Mapping Cuda to Nvidia GPUs

Cuda Thread Hierarchy

- Parallelism in the Cuda Programming Model is expressed as a 4-level Hierarchy



- A ***Stream*** is a list of ***Grids*** that execute in-order. Fermi GPUs execute multiple Streams in parallel
- A ***Grid*** is a set of up to 2^{32} ***Thread Blocks*** executing the same kernel
- A ***Thread Block*** is a set of up to 1024 ***Cuda Threads***
- Each ***Cuda Thread*** is an independent, lightweight, scalar execution context
- Groups of 32 threads form ***Warps*** that execute in lockstep SIMD

Thread Hierarchy

- **threadIdx** is a 3-component vector
  - threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index
  - forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a **thread block**
- The index of a thread and its thread ID relate to each other:
  - for a one-dimensional block, they are the same
  - for a two-dimensional block of size (Dx, Dy),the thread ID of a thread of index (x, y) is (x + y Dx)
  - for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy)

An example

- the following code adds two matrices A and B of size NxN and stores the result into matrix C

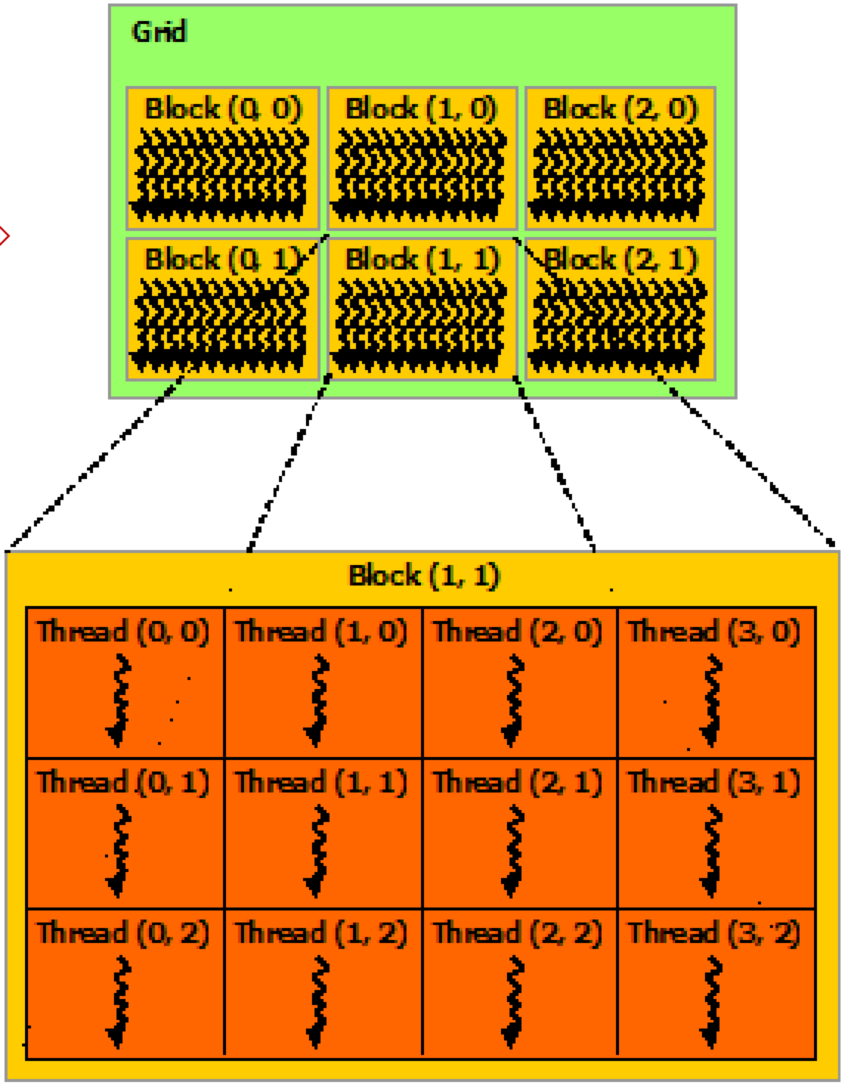
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Number of threads

- There is **a limit to the number of threads per block**
  - since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core
  - On current GPUs, a thread block may contain up to 1024 threads
- a kernel can be executed by **multiple equally-shaped thread blocks**,
  - so that the total number of threads is equal to the number of threads per block times the number of blocks

Grid of Thread Blocks

- Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread
- The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed



Number of blocks

- The number of threads per block and the number of blocks per grid specified in the <<<...>>> syntax can be of type int or dim3
- Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in **blockIdx** variable
- The dimension of the thread block is accessible within the kernel through the built-in blockDim variable

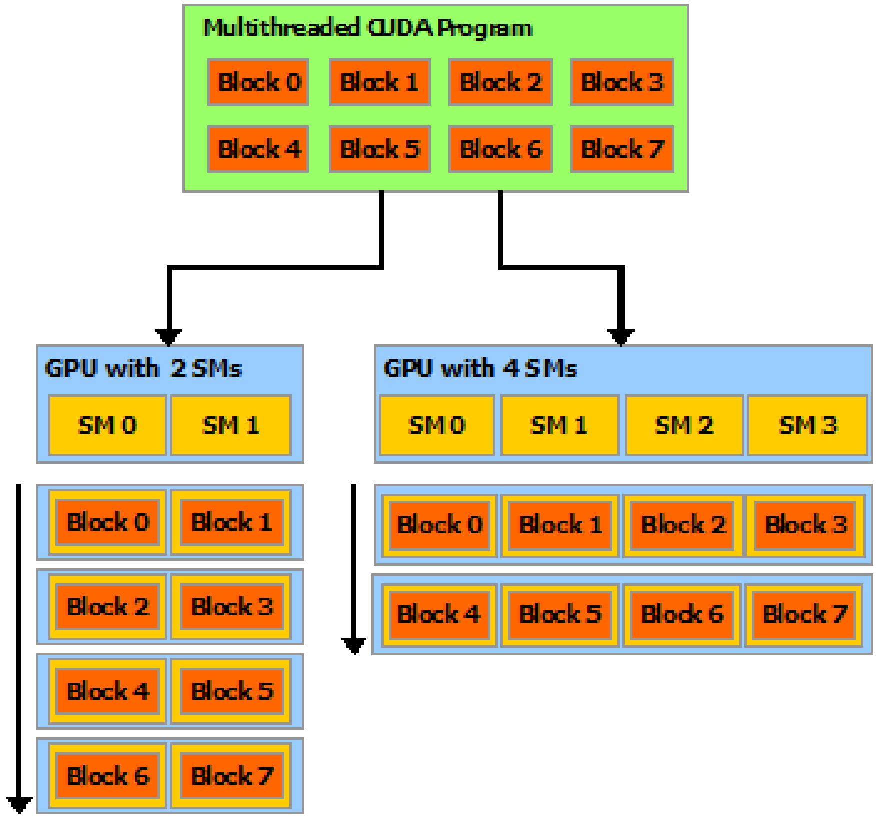
Number of blocks

- Extending the previous MatAdd() example to handle multiple blocks, the code becomes as follows

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread block

- Thread blocks are required to execute independently
  - It must be possible to execute them in any order, in parallel or in series
  - This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores



Cooperate in a block

- Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses
  - one can specify synchronization points in the kernel by calling the __**syncthreads() intrinsic function**
  - __syncthreads() acts as a barrier at which all threads in the block must wait before any is allowed to proceed
- the Cooperative Groups API provides a rich set of thread-synchronization primitives
- the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache)

Thread-Block Synchronization

- Intra-block barrier instruction `__syncthreads()` for synchronizing accesses to `__shared__` and global memory
  - To guarantee correctness, must `__syncthreads()` before reading values written by other threads
  - All threads in a block must execute the same `__syncthreads()`, or the GPU will hang (not just the same number of barriers !)
- Additional intrinsics worth mentioning here:
  - `int __syncthreads_count(int), int __syncthreads_and(int), int __syncthreads_or(int)`

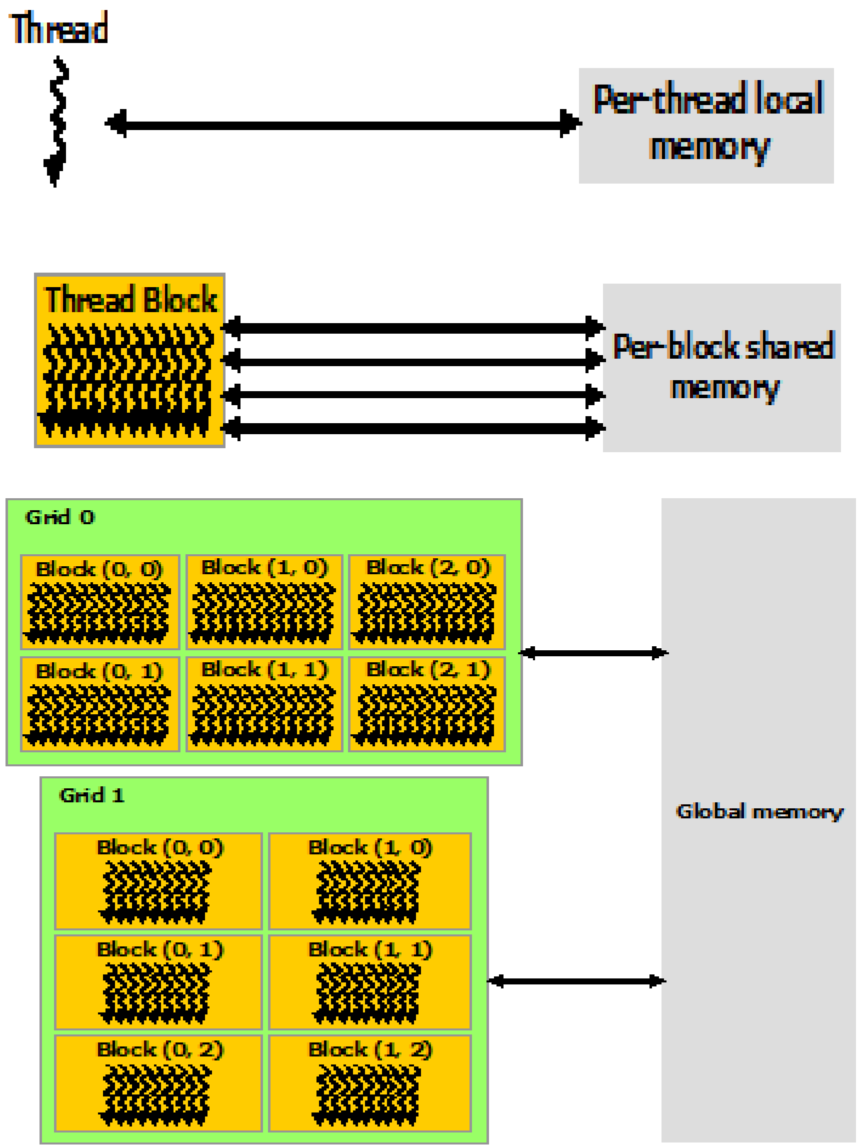
```
extern __shared__ float T[];
__device__ void
transpose (float* a, int lda){
    int i = threadIdx.x, j = threadIdx.y;
    T[i + lda*j] = a[i + lda*j];
    __syncthreads();
    a[i + lda*j] = T[j + lda*i];
}
```

Outline

- Introduction to GPGPUs and Cuda Programming Model
- Programming Model
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compute Capability
- Mapping Cuda to Nvidia GPUs

Memory Hierarchy

- CUDA threads may access data from multiple memory spaces during their execution
  - Each thread has private local memory
  - Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block
  - All threads have access to the same global memory



Using per-block shared memory

- The per-block shared memory / L1 cache is a crucial resource: without it, the performance of most Cuda programs would be hopelessly DRAM-bound
- Block-shared variables can be declared statically:

```
        __shared__ int begin, end;
```

- Software-managed scratchpad is allocated statically:

```
        __shared__ int scratch[128];
        scratch[threadIdx.x] = ... ;
```

- ... or dynamically:

```
    extern __shared__ int scratch[];

    kernel_call <<< grid_dim, block_dim, scratch_size >>> ( ... );
```

- Most intra-block communication is via shared scratchpad:

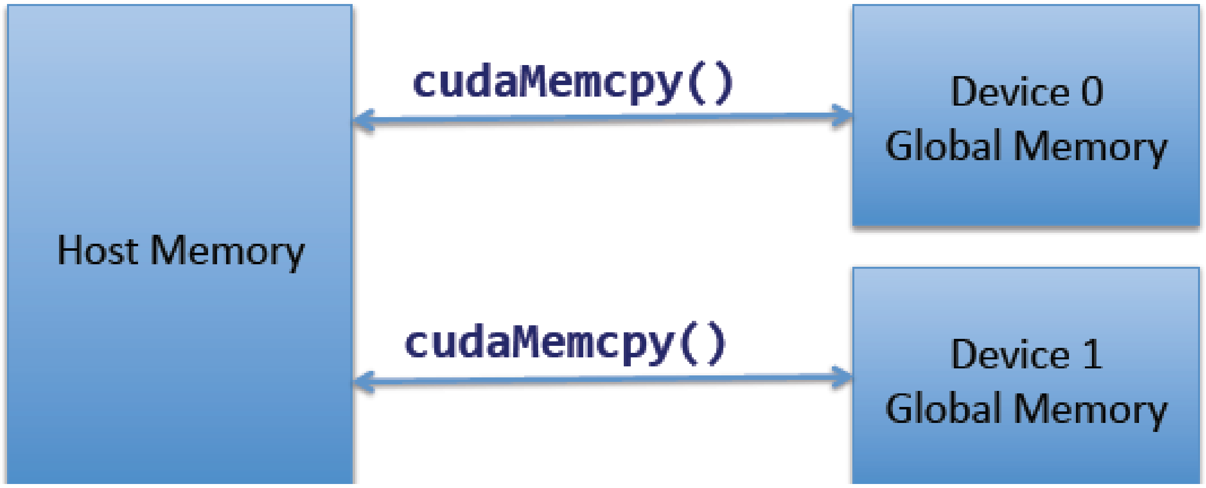
```
        scratch[threadIdx.x] = ...;
        __syncthreads();
        int left = scratch[threadIdx.x - 1];
```

Memory Hierarchy

- There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces
- The global, constant, and texture memory spaces are optimized for different memory usages
- Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats
- The global, constant, and texture memory spaces are persistent across kernel launches by the same application

Cuda Memory Hierarchy

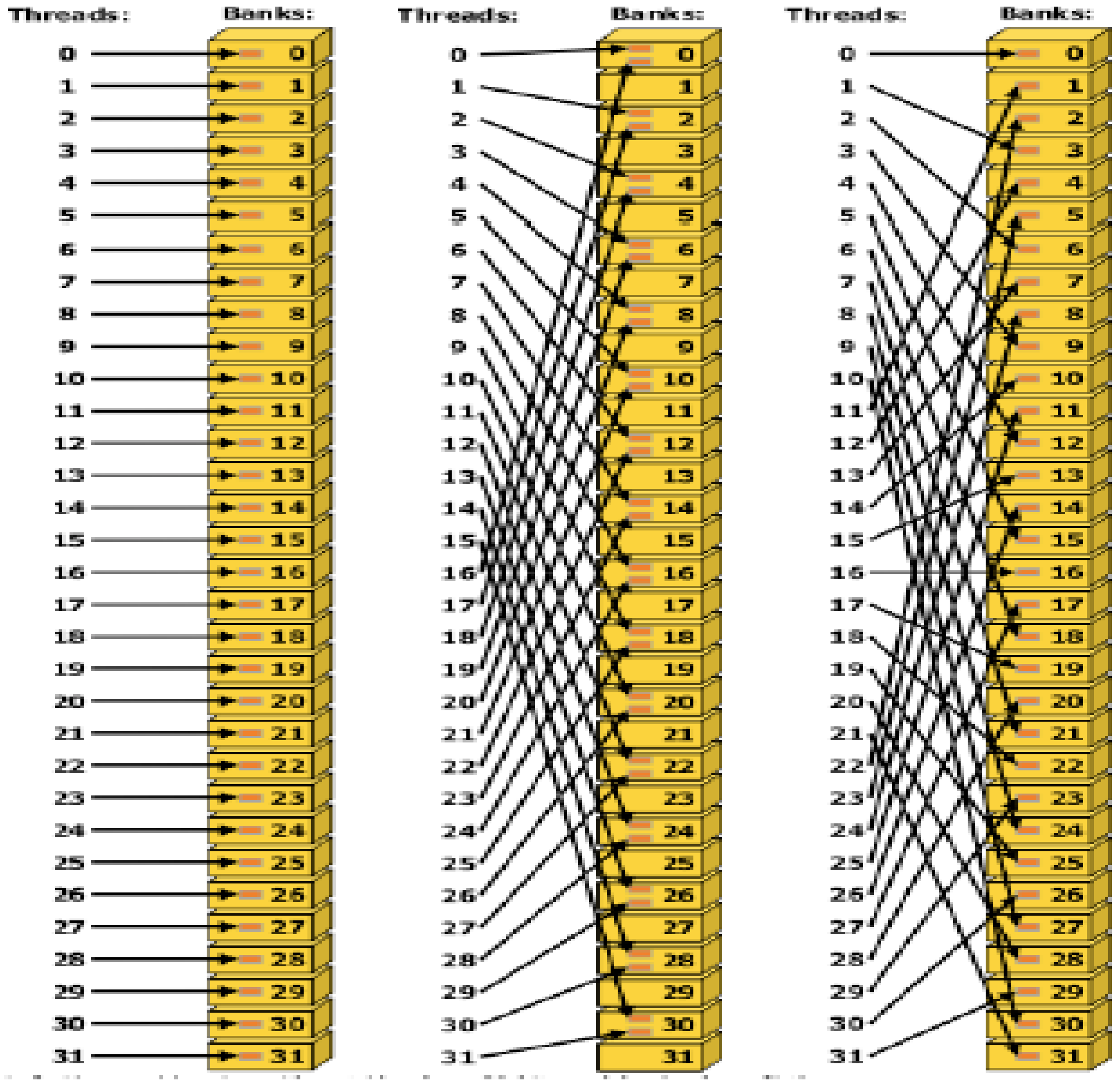
- Each Cuda device in the system has its own Global memory, separate from the Host CPU memory
  - Allocated via cudaMalloc()/cudaFree() and friends
- Host ⟺ Device memory transfers are via cudaMemcpy() over PCI-E, and are extremely expensive
  - microsecond latency, ~GB/s bandwidth
- Multiple Devices managed via multiple CPU threads



Shared Memory Bank Conflicts

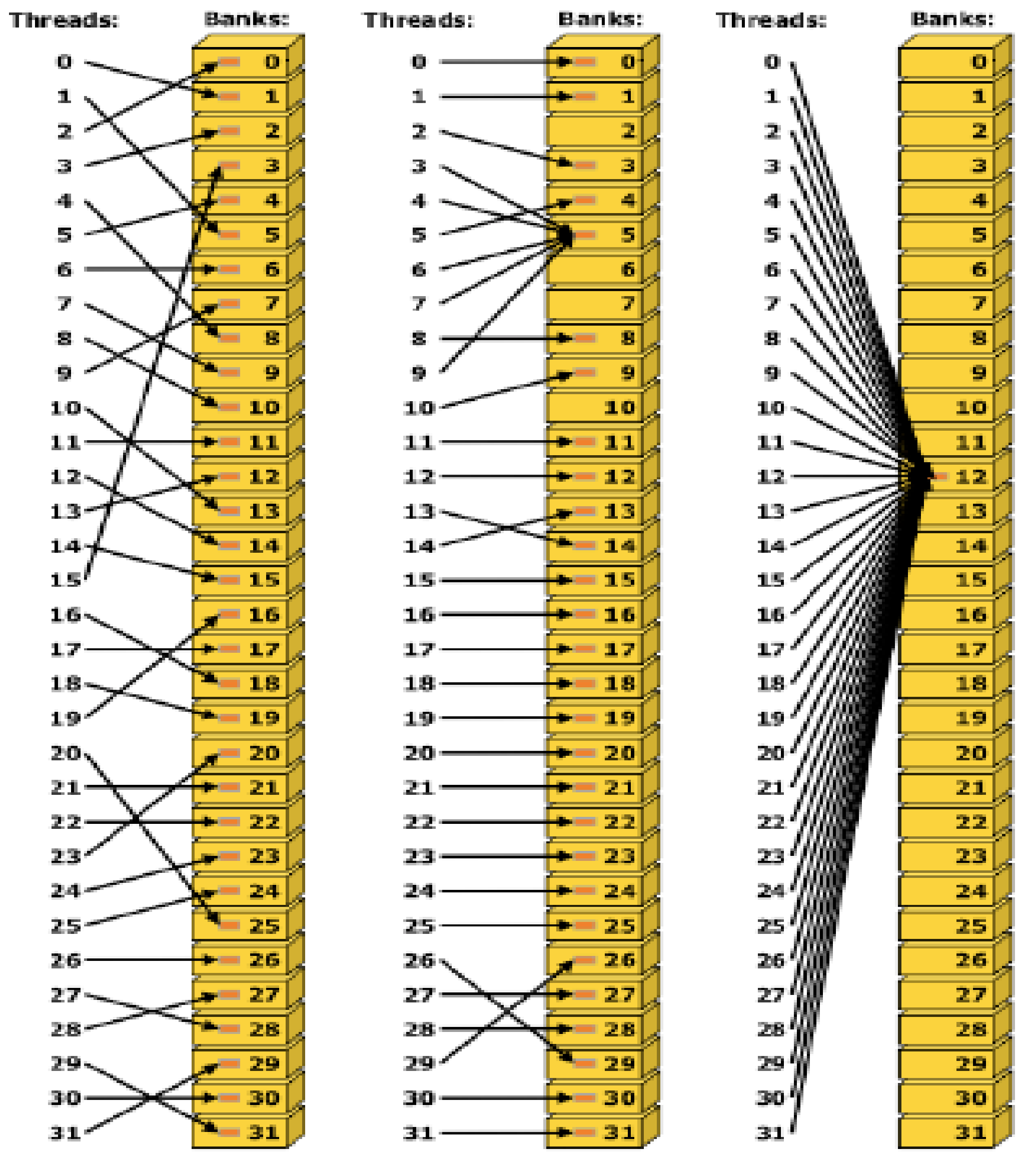
- Shared memory is ***banked:*** it consists of 32 (16, pre-Fermi) independently addressable 4-byte wide memories
  - Addresses interleave: **float *p** points to a float in bank *k, p+1* points to a float in bank ***(k+1) mod 32***
- Each bank can satisfy a single 4-byte access per cycle
  - A ***bank conflict*** occurs when two threads (in the same warp) try to access the same bank in a given cycle
  - The GPU hardware will execute the two accesses serially, and the warp's instruction will take an extra cycle to execute
- Bank conflicts are a second-order performance effect: even serialized accesses to on-chip shared memory is faster than accesses to off-chip DRAM

Shared Memory Bank Conflicts



- Unit-Stride access is **conflict-free**
- Stride-2 access: thread ***n conflicts*** with thread ***16+n***
- Stride-3 access is **conflict-free**

Shared Memory Bank Conflicts



- Three more cases of conflict-free access
- Permuations within a 32-float block are OK
- Multiple threads reading the **same memory address**
- **All threads** reading the same memory address is a ***broadcast***

Atomic Memory Operations

- Cuda provides a set of instructions which execute atomically with respect to each other
  - Allow non-read-only access to variables shared between threads in shared or global memory
  - Substantially more expensive than standard load/stores
  - With voluntary consistency, can implement e.g. spin locks!

```
int atomicAdd (int*,int), float atomicAdd (float*, float), ...
...
int atomicMin (int*,int),
...
int atomicExch (int*,int), float atomicExch (float*,float), ...
int atomicCAS (int*, int compare, int val), ...
```

Voluntary Memory Consistency

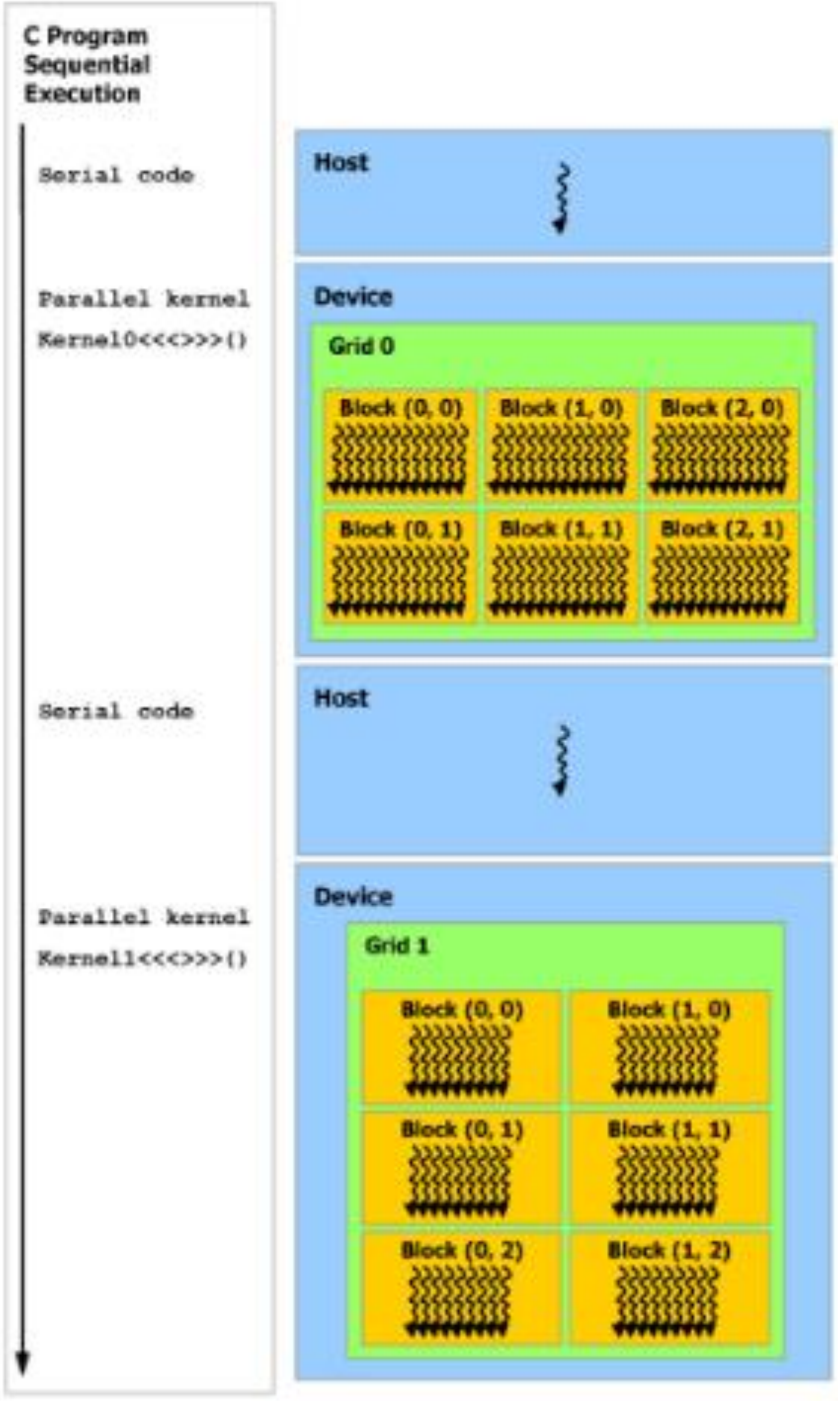
- By default, you cannot assume memory accesses are occur in the same order specified by the program
  - Although a thread's ***own*** accesses appear to that thread to occur in program order
- To enforce ordering, use ***memory fence*** instructions
  - __threadfence_block(): make all previous memory accesses visible to all other threads ***within the thread block***
  - __threadfence(): make previous ***global*** memory accesses visible to all other threads ***on the device***
- Frequently must also use the **volatile** type qualifier
  - Has same behavior as CPU C/C++: the compiler is forbidden from register-promoting values in volatile memory
  - Ensures that pointer dereferences produce load/store instructions
  - Declared as **volatile** float *p; *p must produce a memory ref.

Outline

- Introduction to GPGPUs and Cuda Programming Model
- Programming Model
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compute Capability
- Mapping Cuda to Nvidia GPUs

Heterogeneous Programming

- the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program
- when the kernels execute on a GPU and the rest of the C program executes on a CPU



Heterogeneous Programming

- The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively
  - a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime
  - his includes device memory allocation and deallocation as well as data transfer between host and device memory

Unified Memory

- **Unified Memory** provides managed memory to bridge the host and device memory spaces
  - Managed memory is accessible from all CPUs and GPUs in the system as a single, coherent memory image with a common address space
  - This capability enables oversubscription of device memory and can greatly simplify the task of porting applications by eliminating the need to explicitly mirror data on host and device

Outline

- Introduction to GPGPUs and Cuda Programming Model
- Programming Model
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compute Capability
- Mapping Cuda to Nvidia GPUs

compute capability

- The compute capability of a device is represented by a version number, also sometimes called its "SM version“
  - This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU
  - The compute capability comprises a major revision number X and a minor revision number Y and is denoted by X.Y
  - Devices with the same major revision number are of the same core architecture

Outline

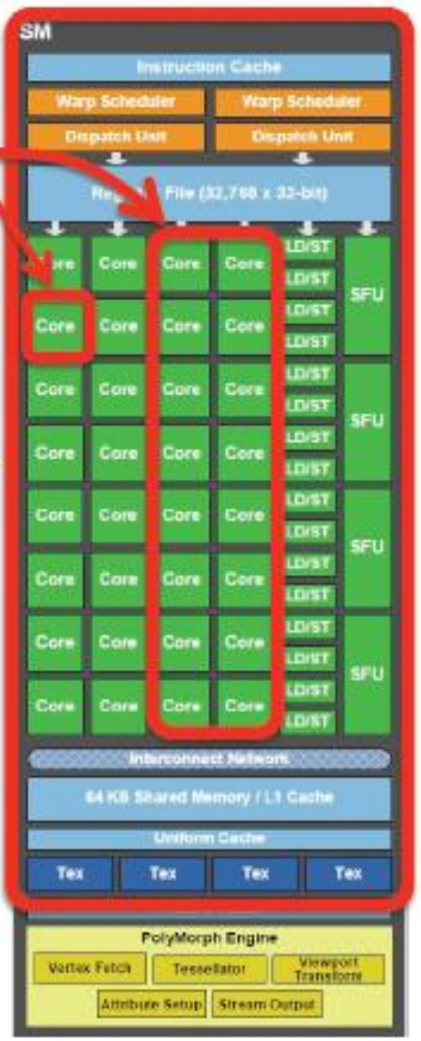
- Introduction to GPGPUs and Cuda Programming Model
- Programming Model
  - Kernels
  - Thread Hierarchy
  - Memory Hierarchy
  - Heterogeneous Programming
  - Compute Capability
- Mapping Cuda to Nvidia GPUs

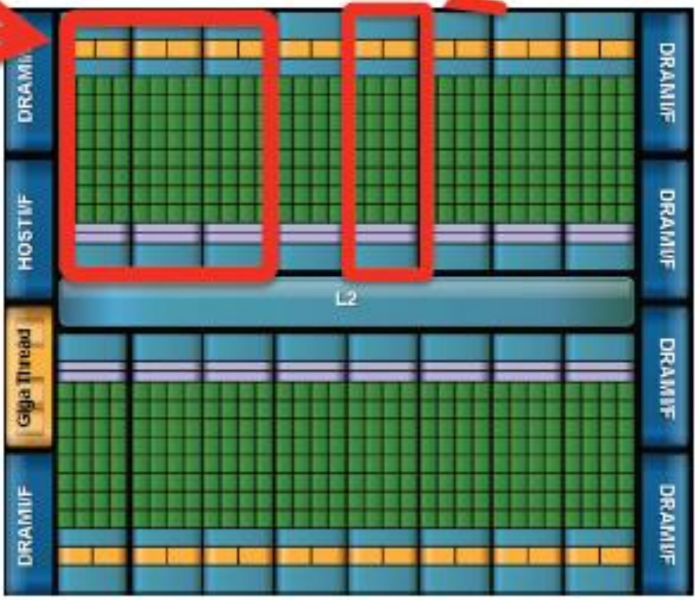
Mapping Cuda to Nvidia GPUs

- Cuda is designed to be "functionally forgiving”
  - Easy to get correct programs running
  - The more time you invest in optimizing your code, the more performance you will get
- Speedup is possible with a simple "Homogeneous SPMD“ approach to writing Cuda programs
- Achieving performance requires an understanding of the hardware implementation of Cuda

Mapping Cuda to Nvidia GPUs

- Scalar Thread ⇔ SIMD Lane
- Warp ⇔ SIMD execution granularity
- Thread Block ⇔ Streaming Multiprocessor
- Grid ⇔ Multiple SMs
- Set of Streams ⇔ Whole GPU



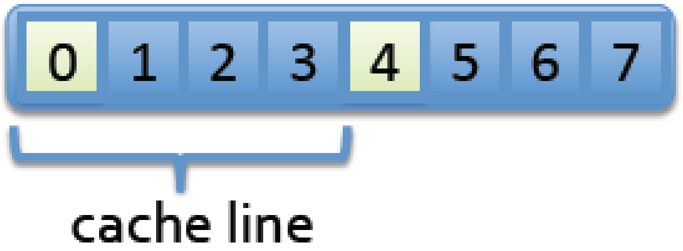


Mapping Cuda to Nvidia GPUs

- Each level of the GPU's processor hierarchy is associated with a memory resource
  - Scalar Threads / Warps: Subset of register file
  - Thread Block / SM: shared memory (l1 Cache)
  - Multiple SMs / Whole GPU: Global DRAM
- Massive multi-threading is used to hide latencies: DRAM access, functional unit execution, PCI-E transfers
- A highly performing Cuda program must carefully trade resource usage for concurrency
  - More registers per thread ⟺ fewer threads
  - More shared memory per block ⟺ fewer blocks

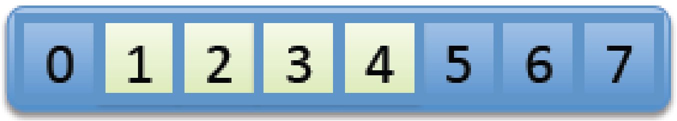
Memory, Memory, Memory

- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem
  - **Memory concerns dominate performance tuning!**
- Memory is SIMD too! The memory systems of CPUs and GPUs alike require memory to be accessed in aligned blocks
  - **Sparse accesses waste bandwidth!**



2 words used, 8 words loaded:
¼ effective bandwidth

  - **Unaligned accesses waste bandwidth!**



4 words used, 8 words loaded:
½ effective bandwidth

Cuda Summary

- The Cuda Programming Model provides a general approach to organizing Data Parallel programs for heterogeneous, hierarchical platforms
  - Currently, the only production-quality implementation is Cuda for C/C++ on Nvidia's GPUs
  - But Cuda notions of "Scalar Threads", "Warps", "Blocks", and "Grids" can be mapped to other platforms as well!
- A simple "Homogenous SPMD" approach to Cuda programming is useful, especially in early stages of implementation and debugging
  - But achieving high efficiency requires careful consideration of the mapping from computations to processors, data to memories, and data access patterns

References

- The content expressed in this chapter is come from
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
  - berkeley university open course (http://parlab.eecs.berkeley.edu/2010bootcampagenda, Slides-Cuda & Slides-OpenCL)