CS 698L: GPGPU Architectures and CUDA C

Swarnendu Biswas

Semester 2019-2020-I

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Rise of GPU Computing

- Popularity of graphical OS in late 80s created a market for a new compute device
  - 2D display accelerators offered hardware-assisted bitmap operations
- Silicon Graphics popularized use of 3D graphics
  - Released OpenGL as a programming interface to its hardware
- Popularity of first-person games in mid-90s was the final push

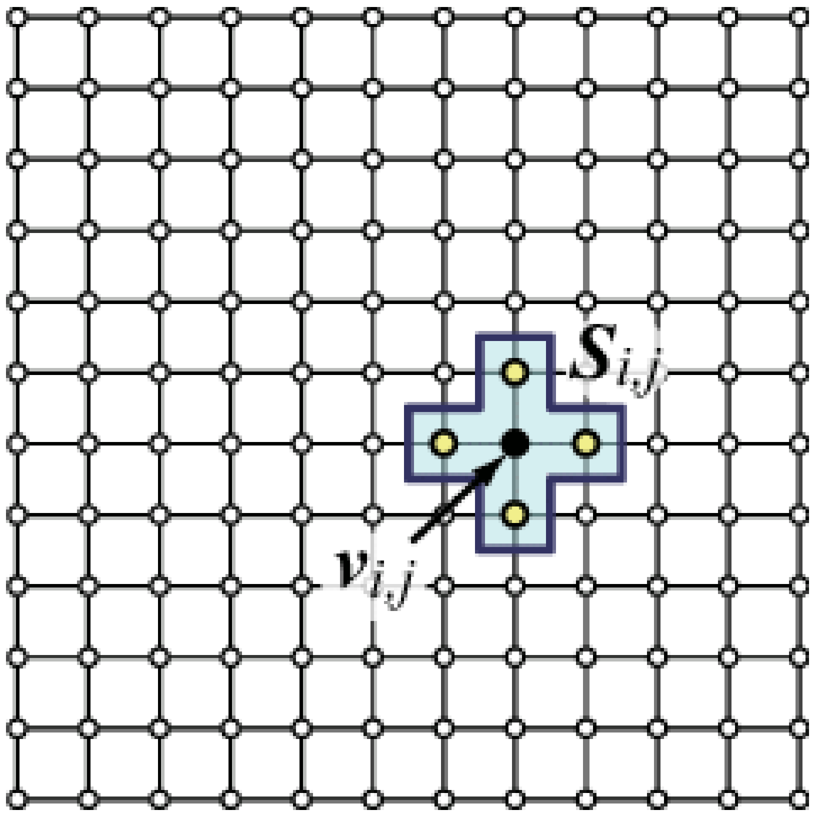
Rise of GPU Computing

- Pixel shaders were used to produce a color for a pixel on screen
  - It uses the (x,y) coordinates, input colors, texture coordinates and other attributes as inputs

- NVIDIA's GeForce 3 series in 2001 implemented the DirectX 8.0 standard from Microsoft

Need for GPGPU Computing Support

- Many real-world applications are compute-intensive and data-parallel
  - They need to process a lot of data, mostly floating-point operations
  - For example, real-time high-definition graphics applications such as your favorite video games
  - Iterative kernels which update elements according to some **fixed pattern called a stencil**



Rise of GPGPU Computing

- Researchers tricked GPUs to perform non-rendering computations
- Programming initial GPU devices for other purposes was very convoluted
- Programming model was very restrictive
  - Limited input colors and texture units, writes to arbitrary locations, floating-point computations
- This spurred the need for a highly-parallel computational device with high computational power and memory bandwidth
  - CPUs are more complex devices catering to a wider audience

Enter NVIDIA and CUDA

- NVIDIA released GeForce 8800 GTX in 2006 with CUDA architecture
  - General-purpose ALU and instruction set for general-purpose computation
  - IEEE compliance for single-precision floating-point arithmetic
  - Allowed arbitrary reads and writes to shared memory
- Introduced CUDA C and the toolchain for ease of development with the CUDA architecture

Rise of GPGPU Computing

- GPUs are now used in different applications
  - Game effects, computational science simulations, image processing and machine learning, linear algebra
- Several GPU vendors like NVIDIA, AMD, Intel, QualComm, and ARM

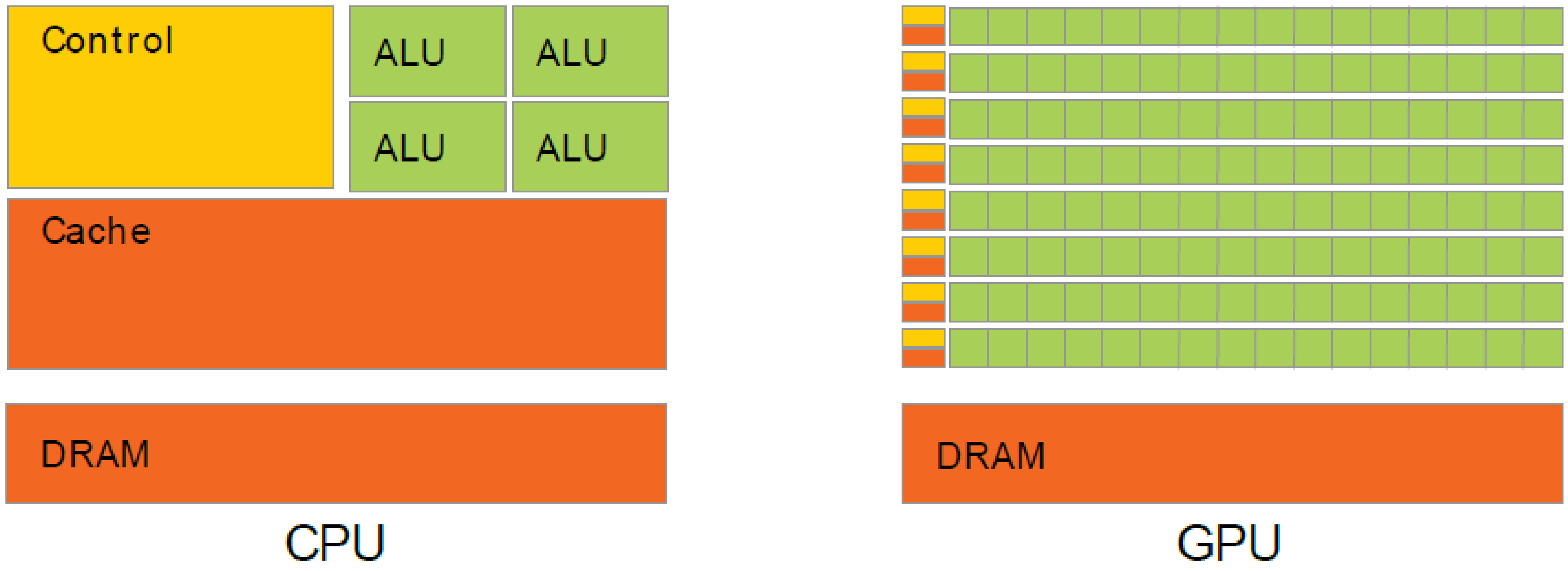
GPGPU Architecture

Key Insights in GPGPU Architecture

- GPUs are suited for compute-intensive data-parallel applications
  - The same program is executed for each data element
  - Less complex control flow
- Multi-core chip
  - SIMD execution within a single core (many ALUs performing the same instruction)
  - Multi-threaded execution on a single core (multiple threads executed concurrently by a core)

Key Insights in GPGPU Architecture

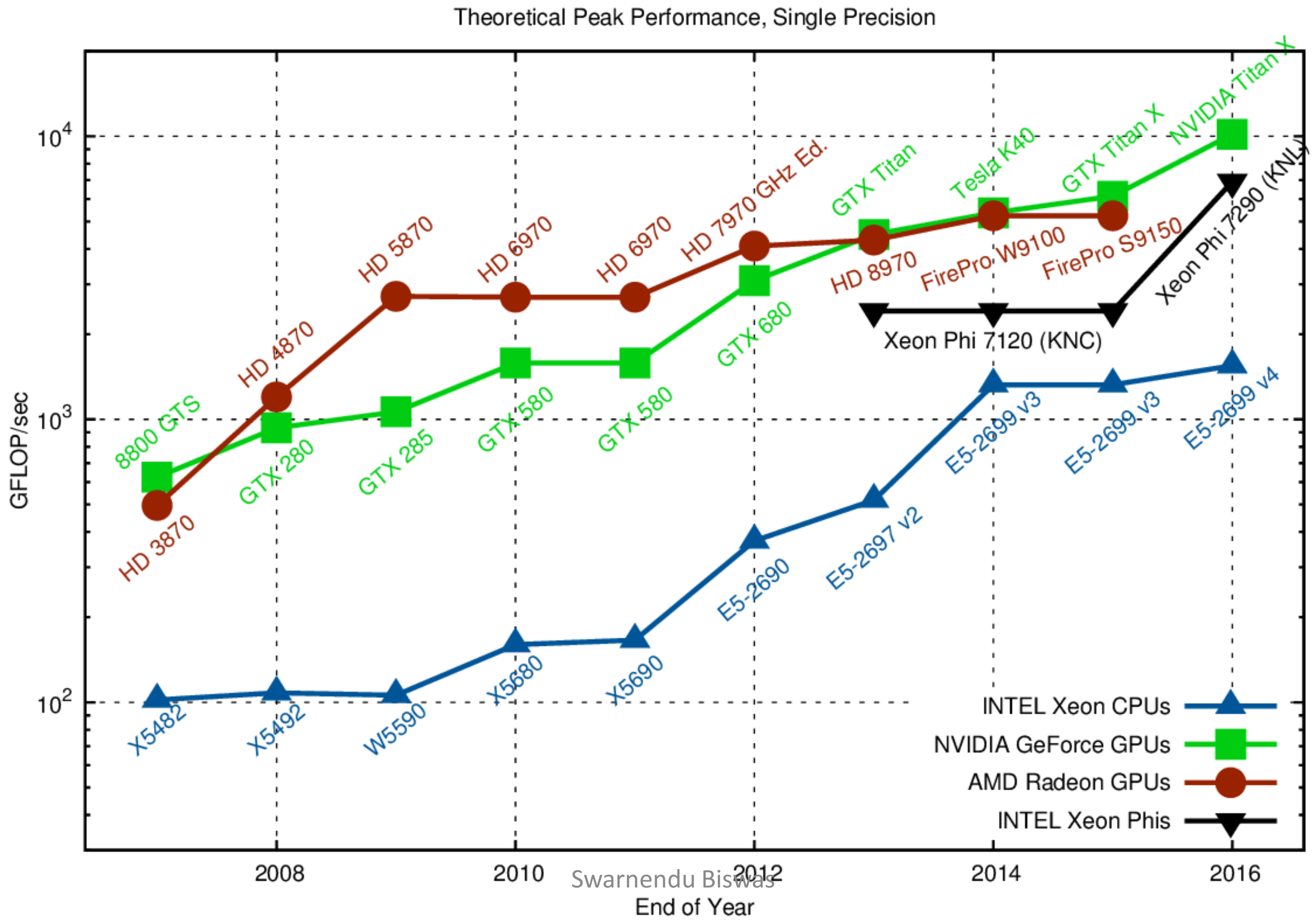
- Much more transistors or real-estate is devoted to computation rather than data caching and control flow



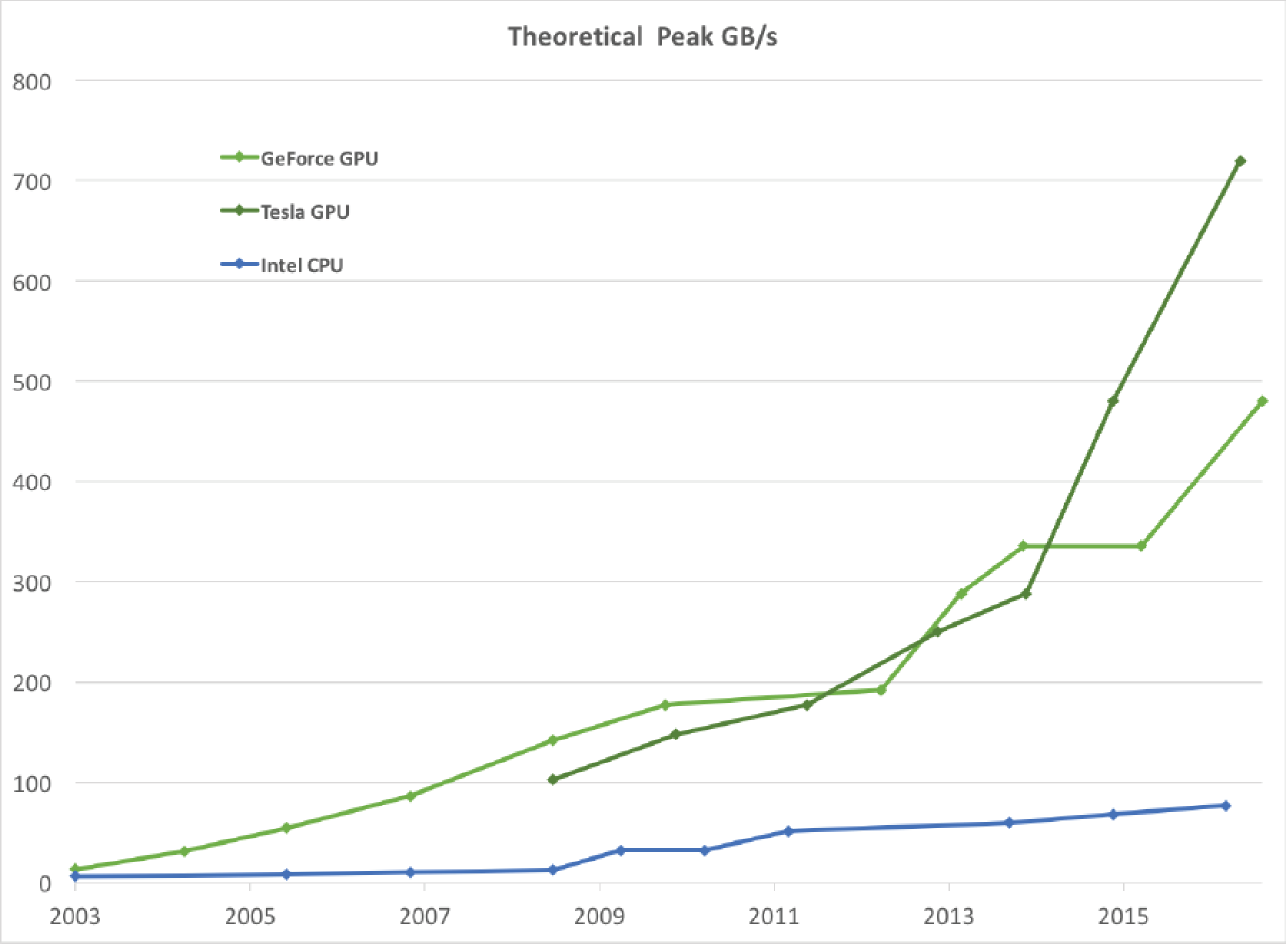
Key Insights in GPGPU Architecture

- GPUs do not reduce latency, they aim to hide latency
- The focus is on overall computing throughput rather than on the speed of an individual core
  - High arithmetic intensity to hide latency of memory accesses
  - Large number of schedulable units

Floating-Point Operations per Second for the CPU and GPU



Memory Bandwidth for CPU and GPU



High-end CPU-GPU Comparison

	Xeon 8180M	**Titan V**
Cores	28	5120 (+ 640)
Active threads	2 per core	32 per core
Frequency	2.5 (3.8) GHz	1.2 (1.45) GHz
Peak performance (SP)	4.1 TFlop/s	13.8 TFlop/s
Peak mem. bandwidth	119 GB/s	653 GB/s
Maximum power	205 W	250 W
Launch price	$13,000	$3000

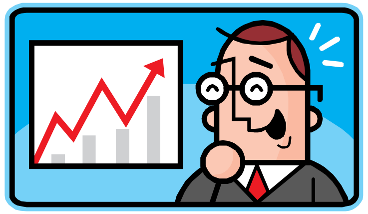
Release dates

- Xeon: Q3’17
- Titan V: Q4’17

Compare GPU to CPU Architecture

- Aims to reduce memory latency with increasingly large and complex memory hierarchy
- Disadvantages
  - The Intel I7-920 processor has some 8 MB of internal L3 cache, almost 30% of the size of the chip
  - Larger cache structures increases the physical size of the processor
  - Implies more expensive manufacturing costs and increases likelihood of manufacturing defects
- Effect of larger, progressively more inefficient caches ultimately results in higher costs to the end user

Advantages of a GPU



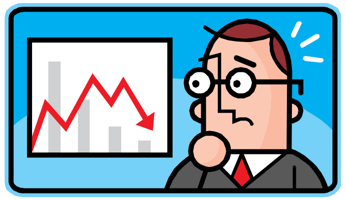
- Performance of Xeon 8180M and Titan V (based on peak values)
  - 3.4x as many operations executed per second
- Main memory bandwidth
  - 5.5x as many bytes transferred per second
- Cost- and energy-efficiency
  - 15x as much performance per dollar
  - 2.8x as much performance per watt

- GPU's higher performance and energy efficiency are due to different allocation of chip area
  - High degree of SIMD parallelism, simple in-order cores, less control/sync. logic, lower cache/scratchpad capacity

From FLOPS to FLOPS/Watt

- Exploiting hardware specialization can improve energy efficiency
- Moving to vector hardware, such as that found in GPUs, may yield up to 10X gain in efficiency by eliminating overheads of instruction processing
- For example, Apple A8 application processor devotes more die area to its integrated GPU than to central processor unit (CPU) cores
- Most energy-efficient supercomputers are now based on GPUs instead of only-CPUs

GPU Disadvantages



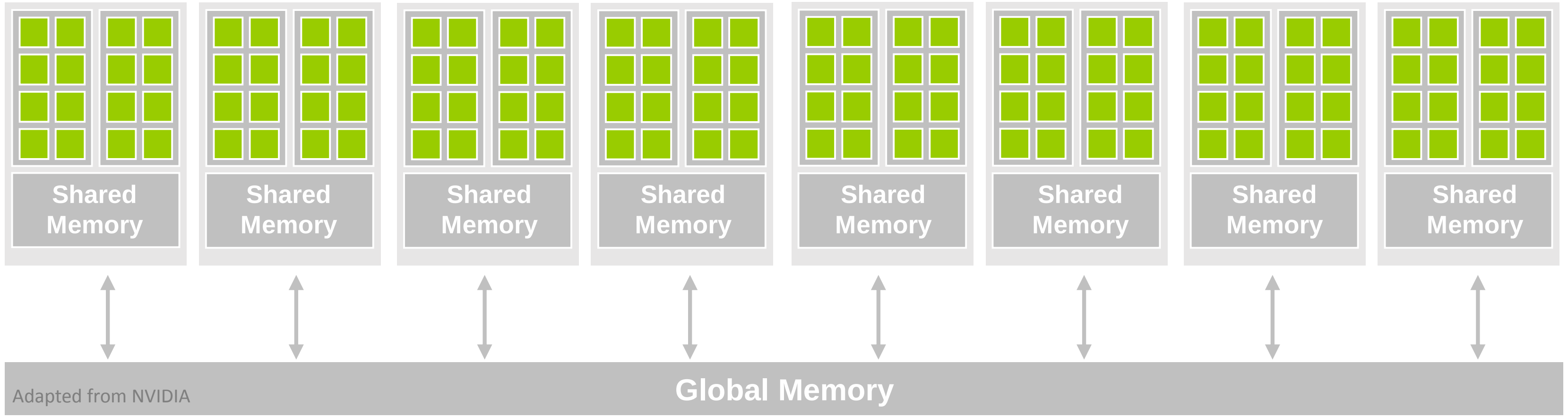
- Clearly, we should be using GPUs all the time

- GPUs can only execute some types of code fast
  - SIMD parallelism is not well suited for all algorithms
  - Need lots of data parallelism, data reuse, & regularity

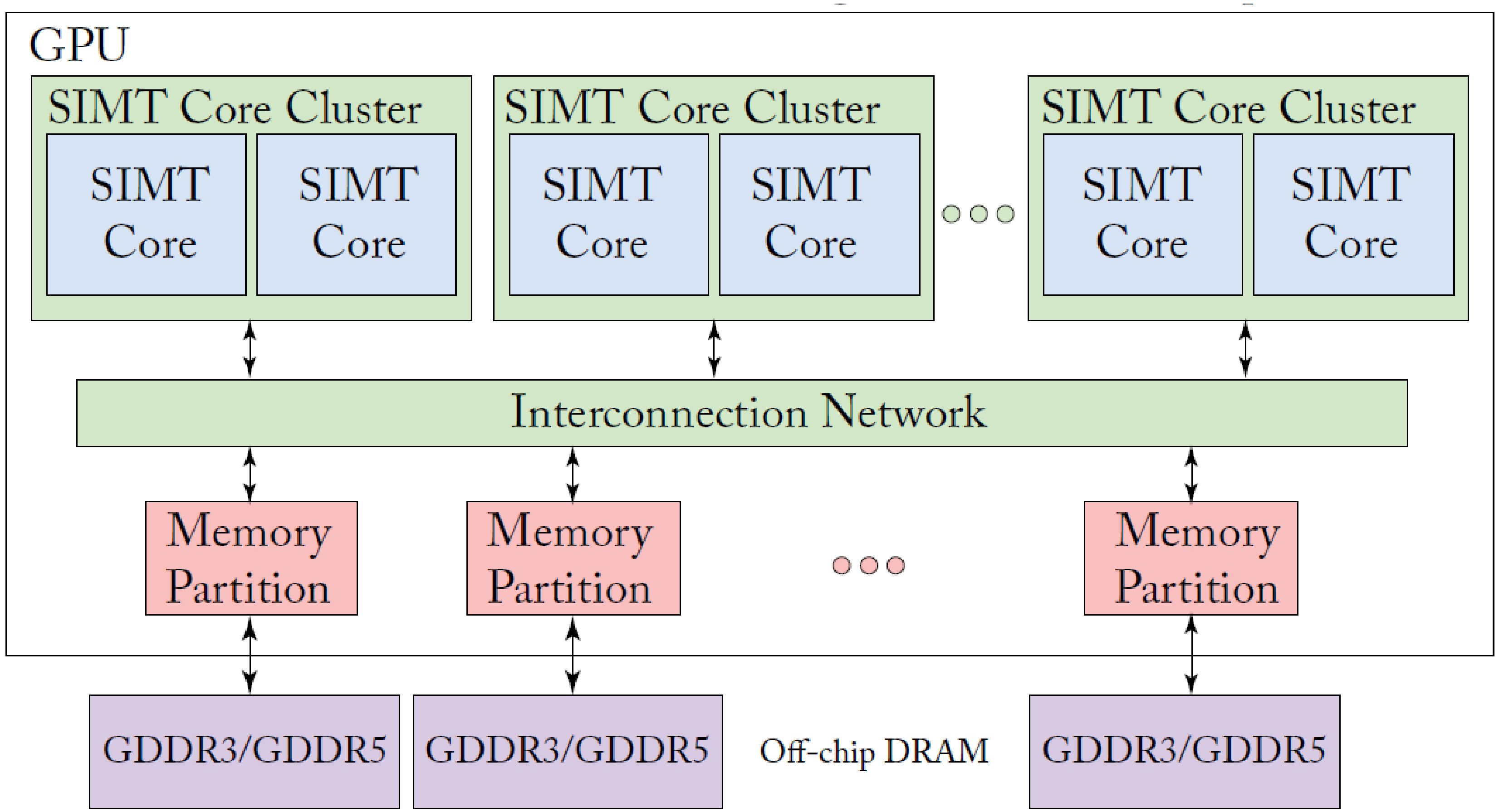
- GPUs are harder to program and tune than CPUs
  - Mostly because of their architecture
  - Fewer tools and libraries exist

GPU Architecture

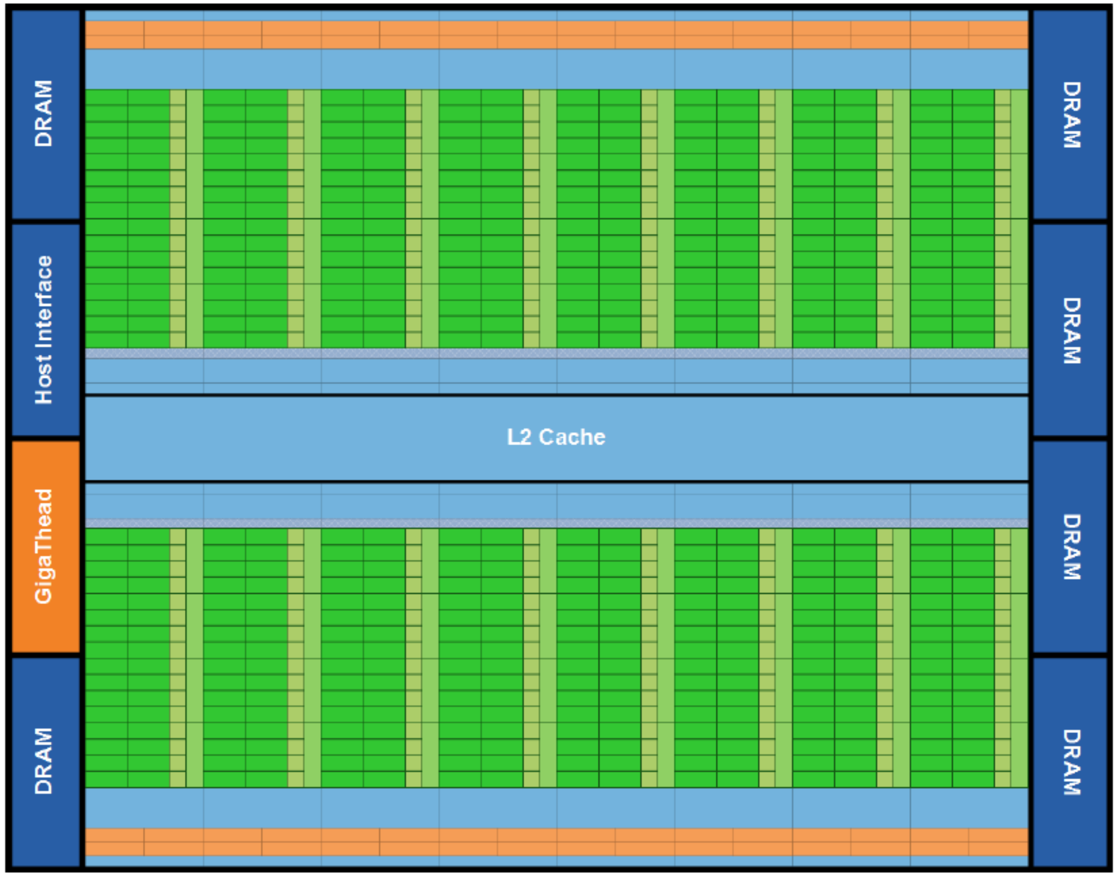
- GPUs consist of Streaming Multiprocessors (SMs)
  - NVIDIA calls these streaming multiprocessors and AMD calls them compute units
- SMs contain Streaming Processors (SPs) or Processing Elements (PEs)
  - Each core contains one or more ALUs and FPUs
- GPU can be thought of as a multi-multicore system



A Generic Modern GPU Architecture



Fermi Architecture



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

Looking into the SM

- A FP32 core is the execution unit that performs single precision floating point arithmetic (floats).
- A FP64 core performs double precision arithmetic (doubles).
- A INT32 Core performs integer arithmetic.
- The warp scheduler selects which warp (group of 32 threads) to send to which execution units (more to come).
- 64 KB Register File –lots of transistors used for very fast memory.
- 128KB of configurable L1 (data) cache or shared memory
- *Shared memory is a used managed cache (more to come).*
- LD/ST units load and store data to/from cores.
- SFU–special function units compute things like transcendentals.

NVIDIA GPU Microarchitecture	Release Year	Remarks
Tesla	2006	Unified shader model
Fermi	2010	Improved double precision performance, support for FMA
Kepler	2012	Focused on energy efficiency, shuffle instructions, dynamic parallelism
Maxwell	2014	Focused on energy efficiency, larger L2 cache
Pascal	2016	Unified memory, half-precision floating-point
Volta	2017	Features tensor cores for deep learning workloads
Turing	2018	Features tensor cores for deep learning workloads and real-time ray tracing. Gaming version of Volta.
Ampere	2020?	

CUDA-Enabled NVIDIA GPUs

	Embedded	Consumer desktop/laptop	Professional Workstation	Data Center
Turing (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series
Volta (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series
Pascal (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler (Compute capabilities 3.x)	Tegra K1	GeForce 600/700 Series	Quadro K Series	Tesla K Series

Compute Capability

- When programming with CUDA, it is very important to be aware of the differences among different versions of hardware
- In CUDA, compute capability refers to architecture features
  - For example, number of registers and cores, cache and memory size, supported arithmetic instructions
- For example, compute capability 1.x devices have 16KB local memory per thread, and 2.x and 3.x devices have 512KB local memory per thread

---

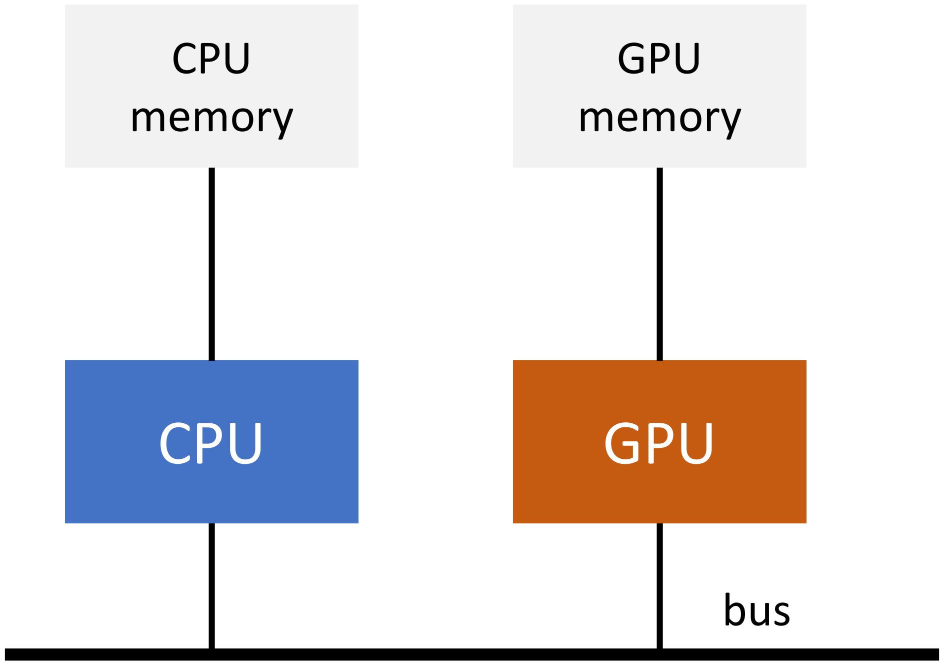
https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications

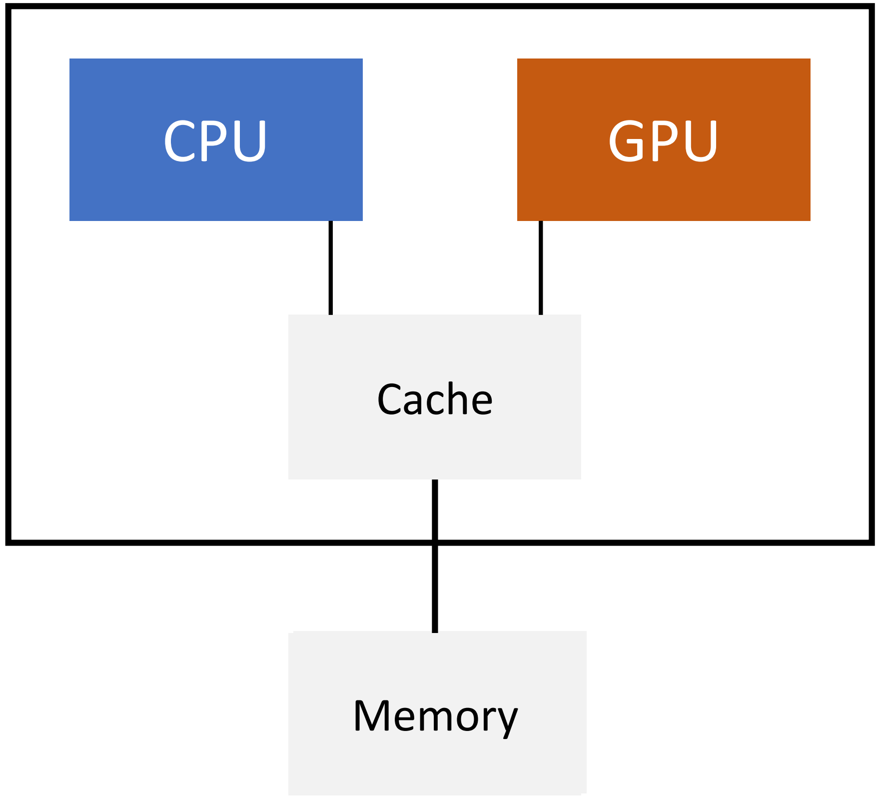
Role of CPUs

- CPU is responsible for initiating computation on the GPU and transferring data to and from the GPU
- Beginning and end of the computation typically require access to input/output (I/O) devices
- There are ongoing efforts to develop APIs providing I/O services directly on the GPU
  - GPUs are not standalone yet, assumes the existence of a CPU

Discrete and Integrated GPUs

Discrete	Integrated





CPUs vs GPUs

CPUs	GPUs

• Designed for running a small number of potentially complex tasks
  • Tasks may be unconnected
  • Suitable to run system software like the OS and applications
• Small number of registers per core private to a task
  • Context switch between tasks is expensive in terms of time
  • Register set must be saved to memory and the next one restored from memory
	• Designed for running large number of simple tasks
	  • Suitable for data-parallelism

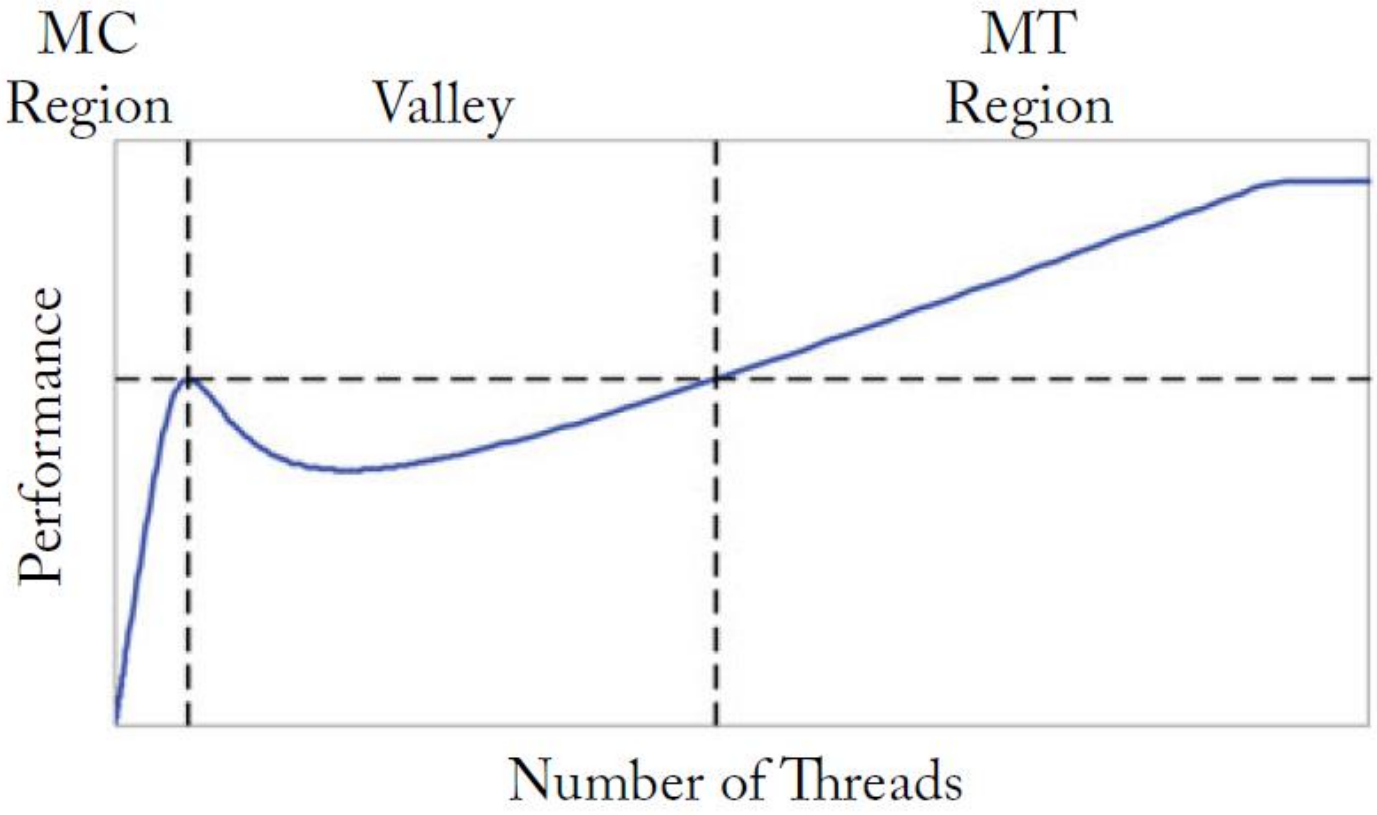
CPUs vs GPUs

CPUs	GPUs

• Small number of registers per core private to a task
  • Context switch between tasks is expensive in terms of time
  • Register set must be saved to RAM and the next one restored from RAM
	• Have a single set of registers but with multiple banks
	  • A context switch involves setting a bank selector to switch in and out the current set of registers
	  • Orders of magnitude faster than having to save to RAM

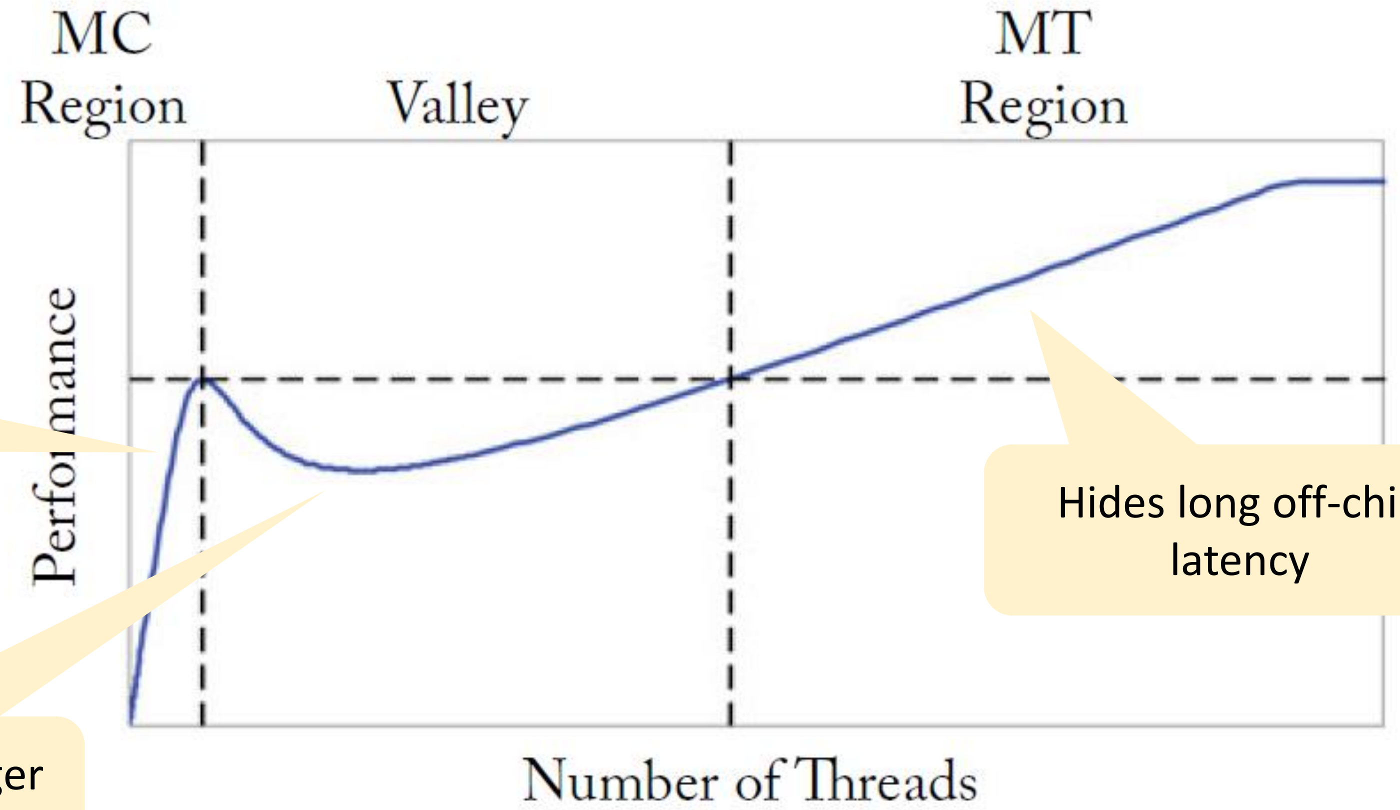
An Analytical Model-based Analysis

Simple cache model where threads do not share data and there is infinite off-chip memory bandwidth



An Analytical Model-based Analysis

Simple cache model where threads do not share data and there is infinite off-chip memory bandwidth



Large cache shared among few threads

Working set no longer fits in the cache

Hides long off-chip latency

CUDA Programming

What is CUDA?

- It is general purpose **parallel computing platform** and programming model that leverages the parallel compute engine in NVIDIA GPUs
  - Introduced in 2007 with NVIDIA Tesla architecture
  - CUDA C, C++, Fortran, PyCUDA are language systems built on top of CUDA
- **Three key abstractions** in CUDA
  - Hierarchy of thread groups
  - Shared memories
  - Barrier synchronization

CUDA Philosophy

SIMT philosophy

- Single Instruction Multiple Thread

Computationally intensive

- The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory

Massively parallel

- The computations can be broken down into hundreds or thousands of independent units of work

CUDA Programming Model

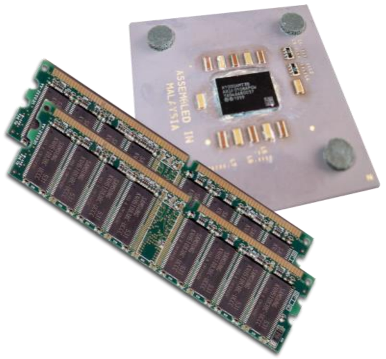
- Allows fine-grained data parallelism and thread parallelism nested within coarse-grained data parallelism and task parallelism

1. Partition the problem into coarse sub-problems that can be solved independently
2. Assign each sub-problem to a “block” of threads to be solved in parallel
3. Each sub-problem is also decomposed into finer work items that are solved in parallel by all threads within the “block”

Heterogeneous Computing

Host

- CPU and its memory (host memory)



Device

- GPU and its memory (device memory)

Heterogeneous Computing

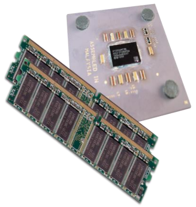
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS   3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}


void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}



int main(void) {
        int *in, *out;           // host copies of a, b, c
        int *d_in, *d_out;       // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);


        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);


        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

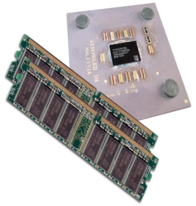
parallel fn

serial code

parallel code

serial code





Heterogeneous Computing

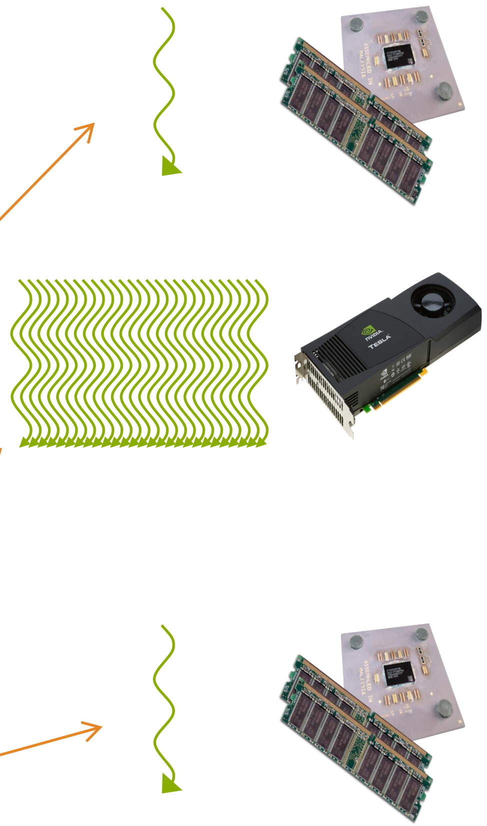
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS    3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}


void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}



int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);


        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);


        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

serial code

parallel code

serial code



Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
  printf("Hello world!\n");
}

int main() {
  hwkernel<<<1, 1>>>();
}
```

```
$ nvcc hello-world.cu

$./a.out

$
```

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
  printf("Hello world!\n");
}

int main() {
  hwkernel<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

```
$ nvcc hell-world.cu

$./a.out
Hello world!

$
```

Program returns immediately after launching the kernel. To prevent program to finish before kernel is completed, we call `cudaDeviceSynchronize()`.

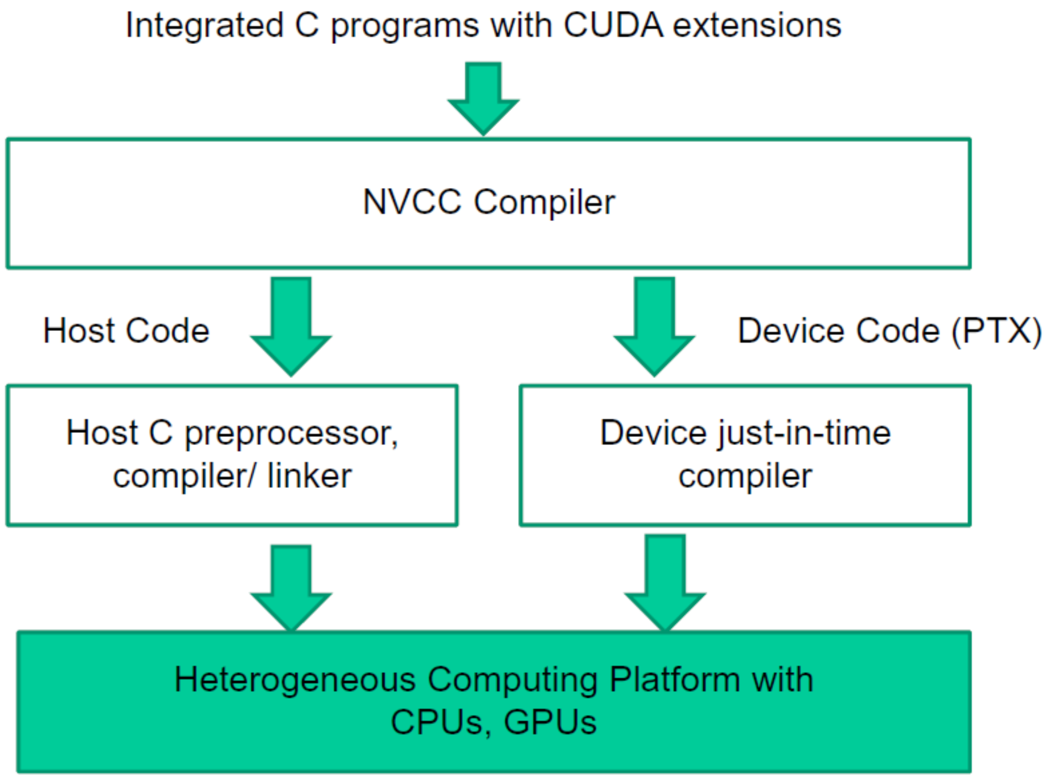
Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
  printf("Hello world!\n");
}

int main() {
  hwkernel<<<1, 32>>>();
  cudaThreadSynchronize();
}
```

```
$ nvcc hell-world.cu

$./a.out
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
…
…
$
```

How nvcc works?

- Nvcc is a driver program based on LLVM
  - Compiles and links all input files
  - Requires a general-purpose C/C++ host compiler
    - Uses gcc and g++ by default on Linux platforms
  - `nvcc --version`



https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

NVCC Details

Important Options	Description
`-std {c++03|c++11|c++14}`	Select a particular C++ dialect
`-m {32|64}`	Specify the architecture
`-arch ARCH`	Specify the class of the virtual GPU architecture
`-code CODE`	Specify the name of the GPU to assemble and optimize for

Input File Type	Description
`.cu`	CUDA source file
`.c, .cpp, .cxx, .cc`	C/C++ source files
`.ptx`	PTX intermediate assembly
`.cubin`	CUDA device binary code for a single GPU architecture
`.fatbin`	CUDA fat binary file that may contain multiple PTX and CUBIN files
`.a, .so, .lib`	…

NVIDIA. CUDA Compiler Driver NVCC. v10.1.

NVCC Details

Important Options	Description
-std {c++03|c++11|c++14}	Select a particular C++ dialect
-m {32|64}	Specify the architecture
-arch ARCH	Specify the class of the
-code CODE	

Input File Type	Description
.cu	CUDA source file
.c, .cpp, .cxx, .cc	C/C++ source files
.ptx	PTX intermediate assembly
	ary code for a ecture
	ile that may PTX and CUBIN
.a, .so, .lib	…

```
nvcc –arch=sm_50
⇩
nvcc –arch=compute_50 –code=sm_50,compute_50
```

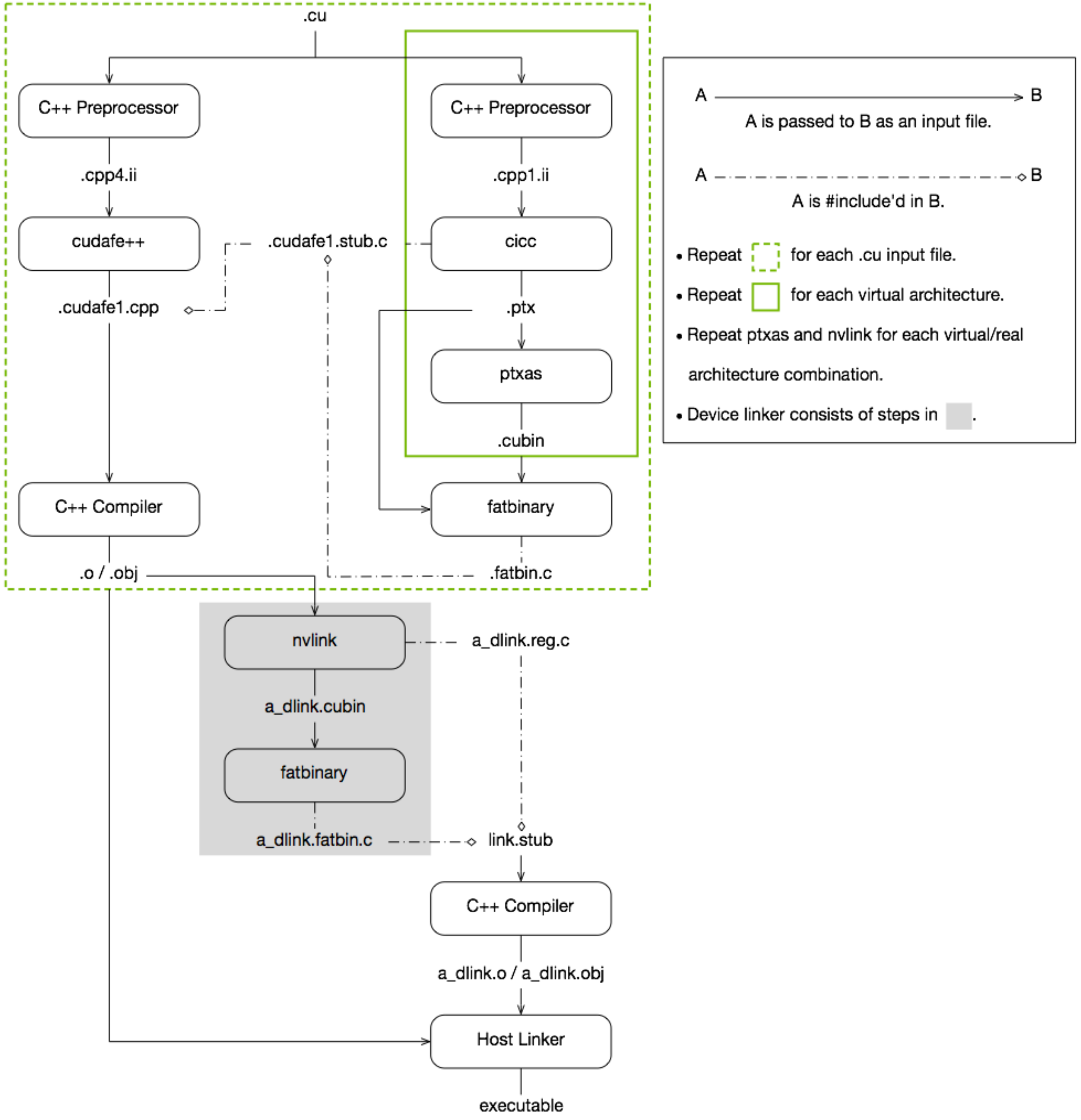
NVIDIA. CUDA Compiler Driver NVCC. v10.1.

CUDA Compilation Trajectory

- Conceptually, the flow is as follows
  - Input program is preprocessed for device compilation
  - It is compiled to a CUDA binary and/or PTX (Parallel Thread Execution) intermediate code which are encoded in a fatbinary
  - Input program is processed for compilation of the host code
    - CUDA-specific C++ constructs are transformed to standard C++ code
  - Synthesized host code and the embedded fatbinary are linked together to generate the executable

CUDA Compilation Trajectory

- A compiled CUDA device binary includes
  - Program text (instructions)
  - Information about the resources required
    - N threads per block
    - X bytes of local data per thread
    - M bytes of shared space per block



https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

Function Declarations in CUDA

	Executed on	Callable from
`__device__ float deviceFunc()`	Device	Device
`__global__ void kernelFunc()`	Device	Host
`__host__ float hostFunc()`	Host	Host

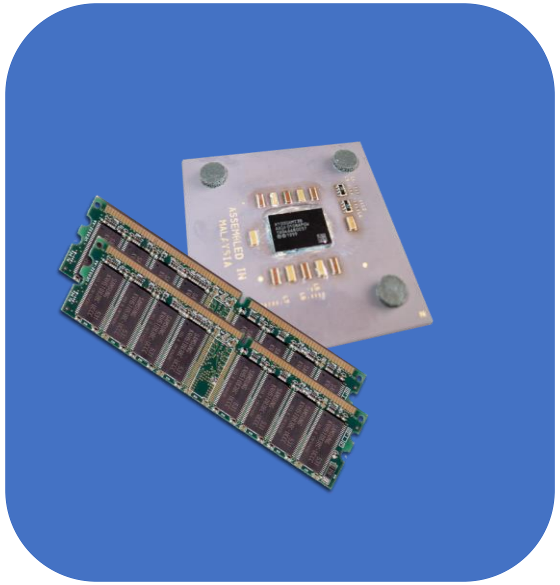
- __global__ define a kernel function, must return void
- __device__ functions can have return values
- __host__ is default, and can be omitted
- Prepending __host__ __device__ causes the system to compile separate host and device versions of the function

Dynamic Parallelism

- It is possible to launch kernels from other kernels
- Calling `__global__` functions from the device is referred to as dynamic parallelism
  - Requires CUDA devices of compute capability 3.5 and CUDA 5.0 or higher

Execution Model

Host
(serial execution)



Serial code on host

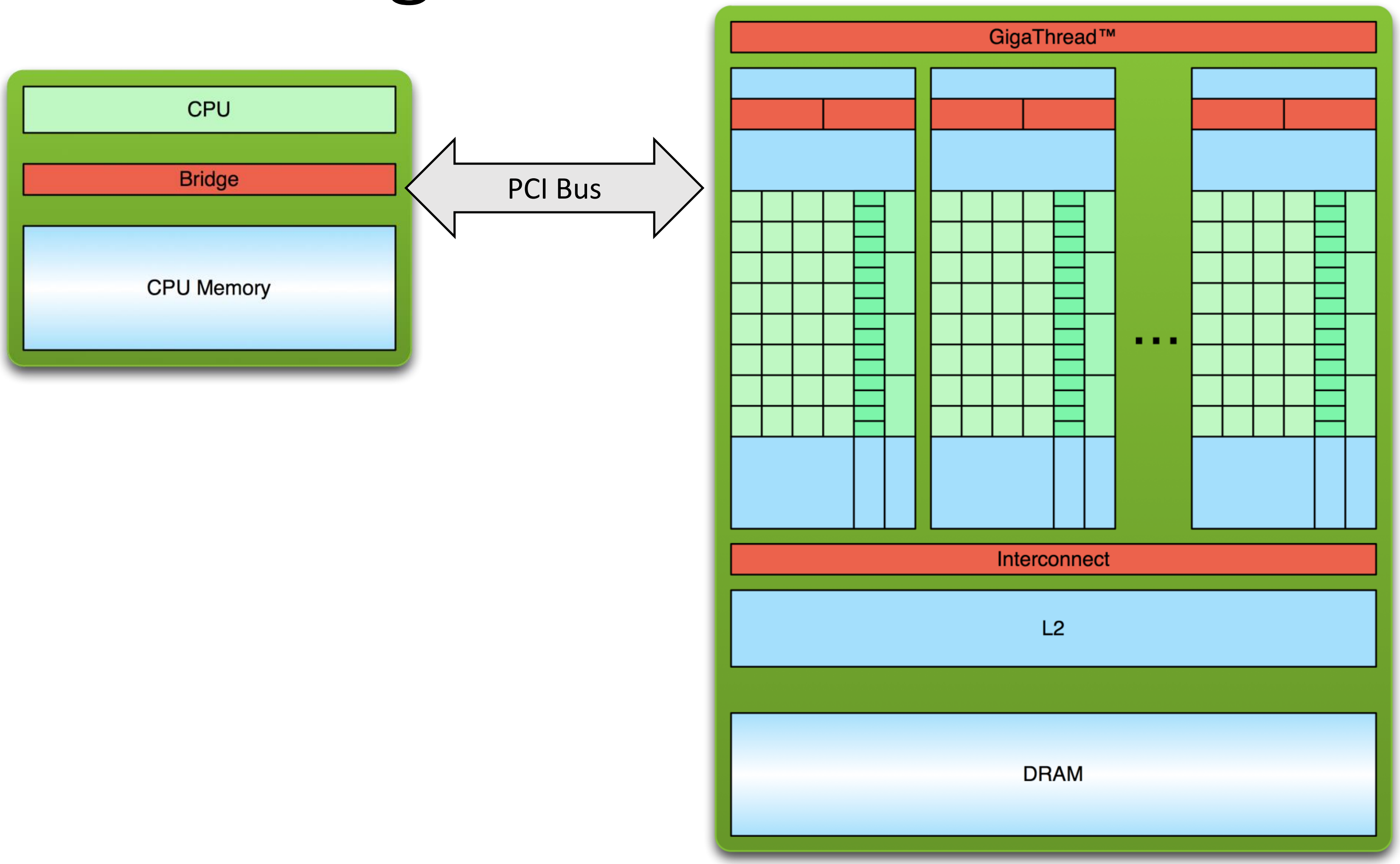
Parallel kernel on device

Serial code on host

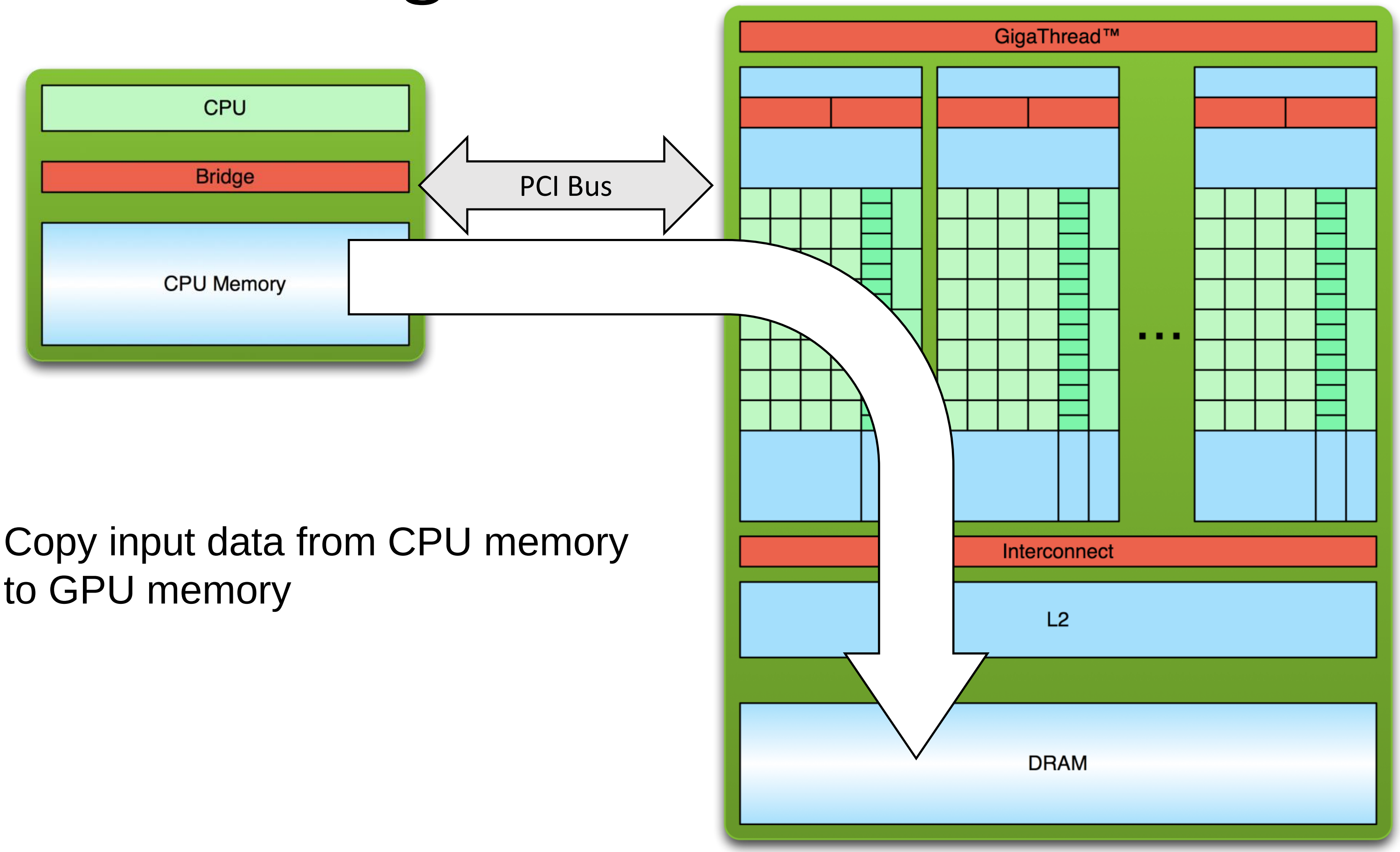
Parallel kernel on device

Device
(Parallel execution)

Simple Processing Flow

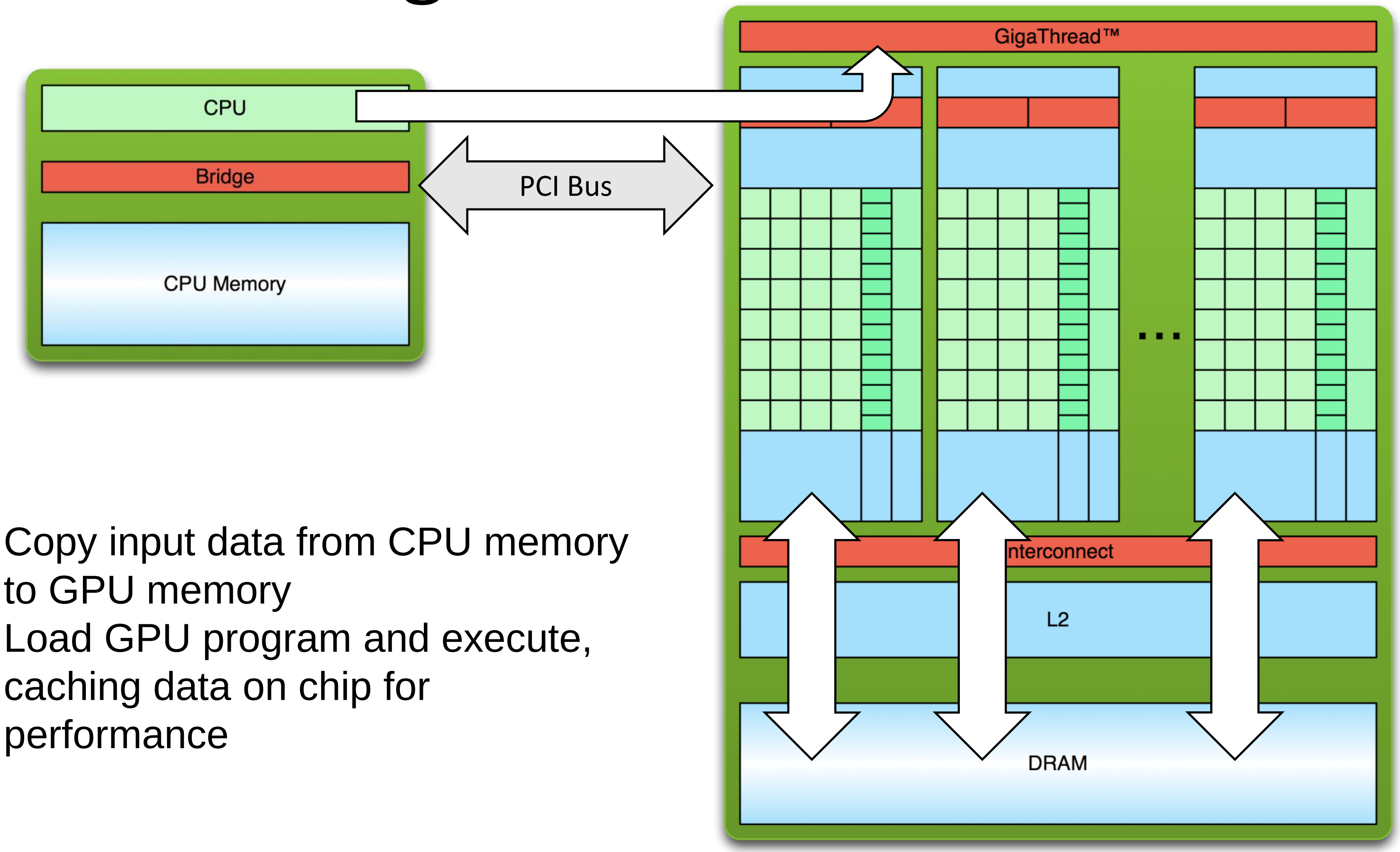


Simple Processing Flow



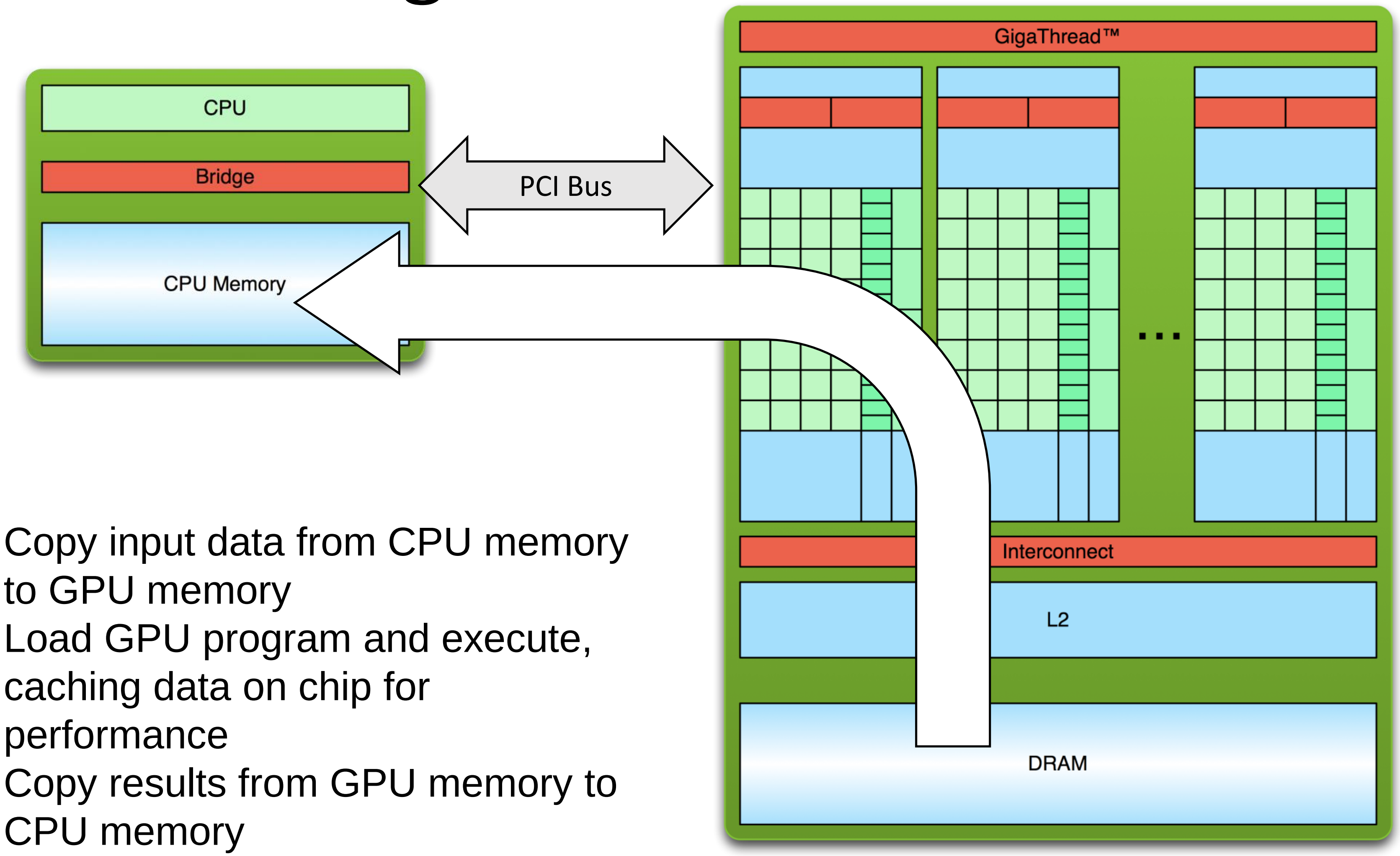
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Vector Addition Example

```
__global__ void VecAdd(float* A, float* B,
         float* C, int N) {
  int i = blockDim.x * blockIdx.x + threadI
dx.x;
  if (i < N)
    C[i] = A[i] + B[i];
}

int main() {
  …
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);
  float* h_C = (float*)malloc(size);
```

```
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);

// Copy vectors from host memory to
// device memory
cudaMemcpy(d_A, h_A, size,
           cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size,
           cudaMemcpyHostToDevice);
```

Vector Addition Example

```
  // Invoke kernel
  int threadsPerBlock = 256;
  int blocksPerGrid = N/threadsPerBlock;
  VecAdd<<<blocksPerGrid, threadsPerBloc
k>>>(d_A, d_B, d_C, N);
```

```
// Copy result from device memory to
// host memory
  cudaMemcpy(h_C, d_C, size,
             cudaMemcpyDeviceToHost);
  …
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
…
}
```

Typical CUDA Program Flow

1. Load data into CPU memory
   - `fread/rand`
2. Copy data from CPU to GPU memory
   - `cudaMemcpy(..., cudaMemcpyHostToDevice)`
3. Call GPU kernel
   - `yourkernel<<<x, y>>>(...)`
4. Copy results from GPU to CPU memory.
   - `cudaMemcpy(..., cudaMemcpyDeviceToHost)`
5. Use results on CPU

CUDA Extensions for C/C++

- Kernel launch
  - Calling functions on GPU
- Memory management
  - GPU memory allocation, copying data to/from GPU
- Declaration qualifiers
  - `__device__, __shared, __local, __global__, __host__`
- Special instructions
  - Barriers, fences, etc.
- Keywords
  - `threadIdx, blockIdx, blockDim`

C++11 Support from CUDA 7.5+

Supported Features

- auto
- lambdas
- constexpr
- rvalue references
- range-based for loops

Unsupported Features

- Standard library
  - You cannot use std::cout in device code

Kernels

- Special functions that a CPU can call to execute on the GPU
  - Executed N times in parallel by N different CUDA threads
  - Cannot return a value
- Each thread will execute `VecAdd()`
- Each thread has a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable

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

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

Kernels

- GPU spawns m blocks with n threads (i.e., m*n threads total) that run a copy of the same function

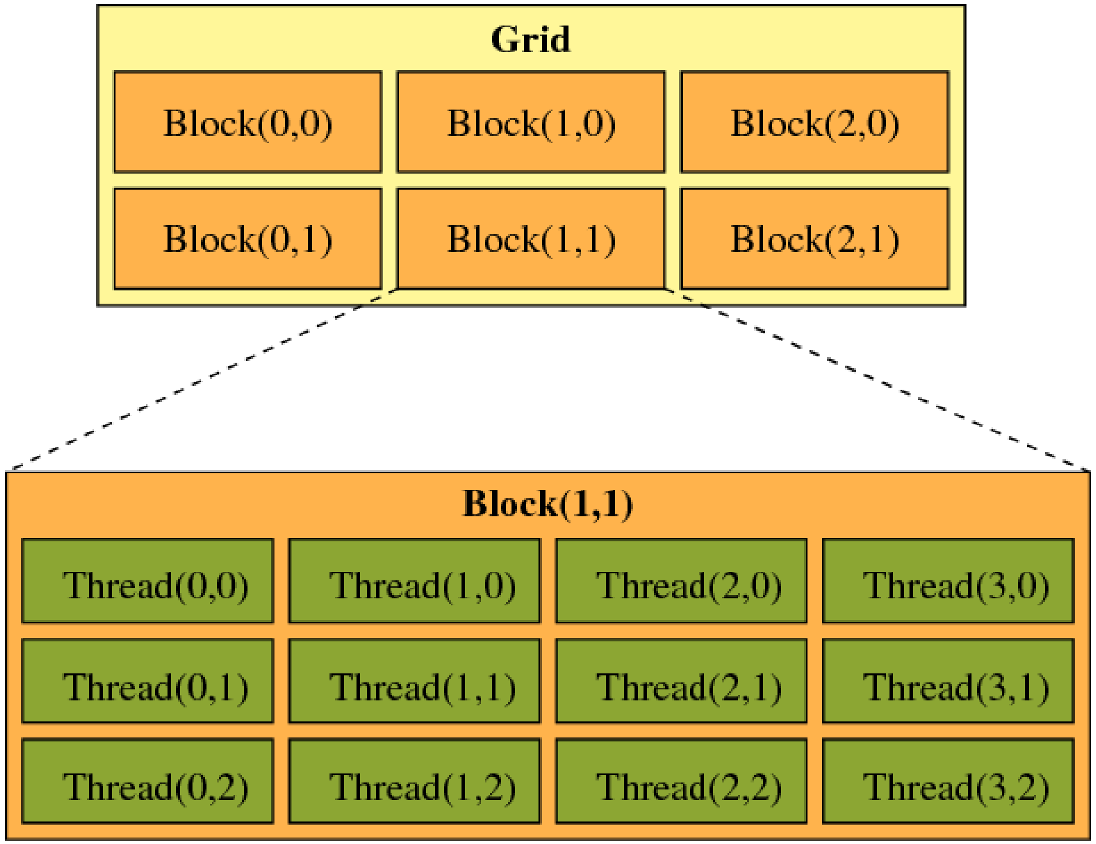
```
KernelName<<<m, n>>>(arg1, arg2, ...)
```

- CPU can continue processing while GPU runs kernel
- Kernel call returns when all threads have terminated

```
kernel1<<<X,Y>>>(...); // kernel starts execution, CPU continues to next statement
kernel2<<<X,Y>>>(...); // kernel2 placed in queue, will start after kernel1 finishes, CPU continues
cudaMemcpy(...); // CPU blocks until memory is copied, memory copy starts after all preceding CUDA calls finish
```

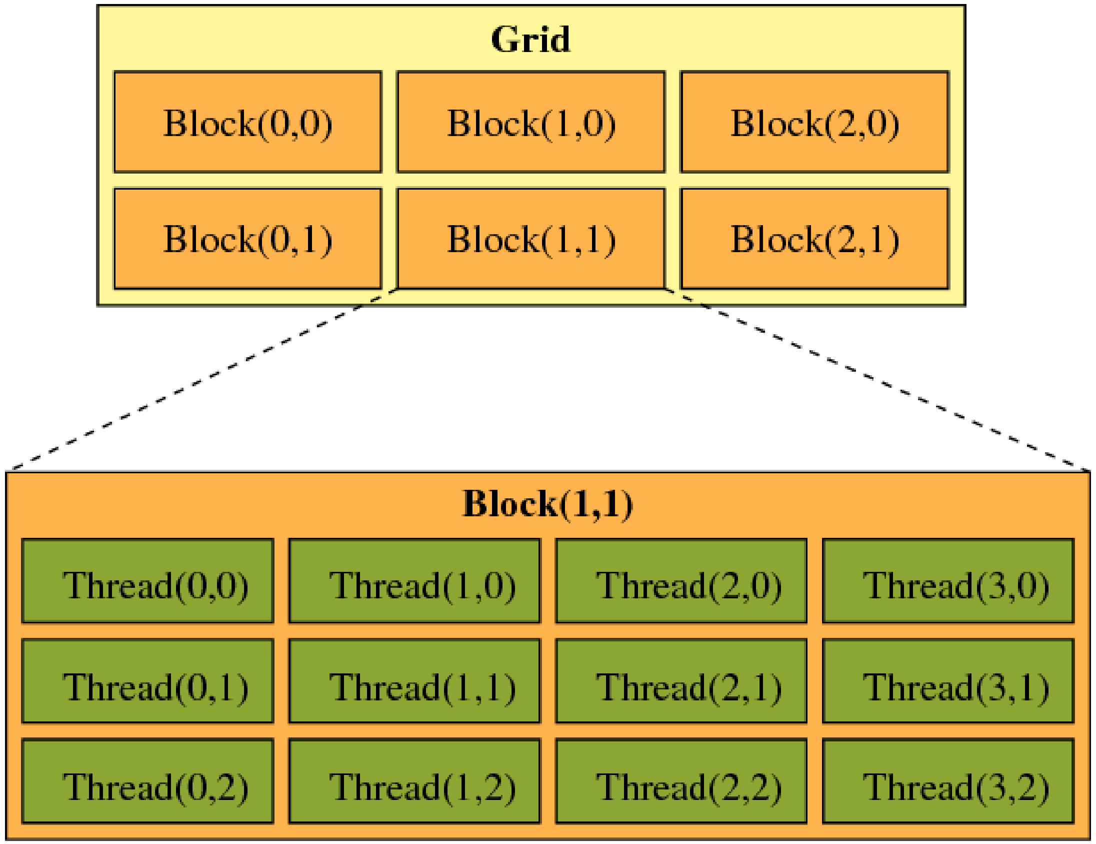
Thread Hierarchy

- A kernel executes in parallel across a set of parallel threads
- All threads that are generated by a kernel launch are collectively called a grid
- Threads are organized in thread blocks, and blocks are organized in to grids



Thread Hierarchy

- A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory
- A grid is an array of thread blocks that execute the same kernel
  - Read inputs to and write results to global memory
  - Synchronize between dependent kernel calls



Dimension and Index Variables

Type is `dim3`

Dimension

- `gridDim` specifies the number of blocks in the grid
- `blockDim` specifies the number of threads in each block

Index

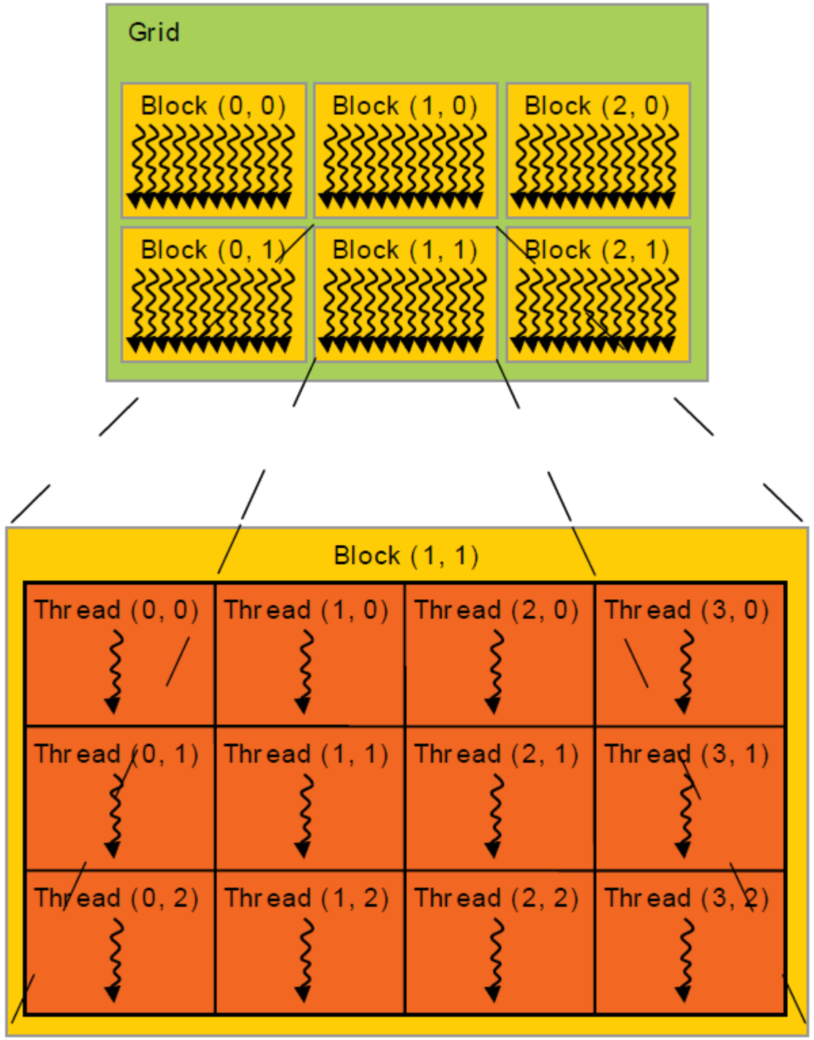
- `blockIdx` gives the index of the block in the grid
- `threadIdx` gives the index of the thread within the block

Thread Hierarchy

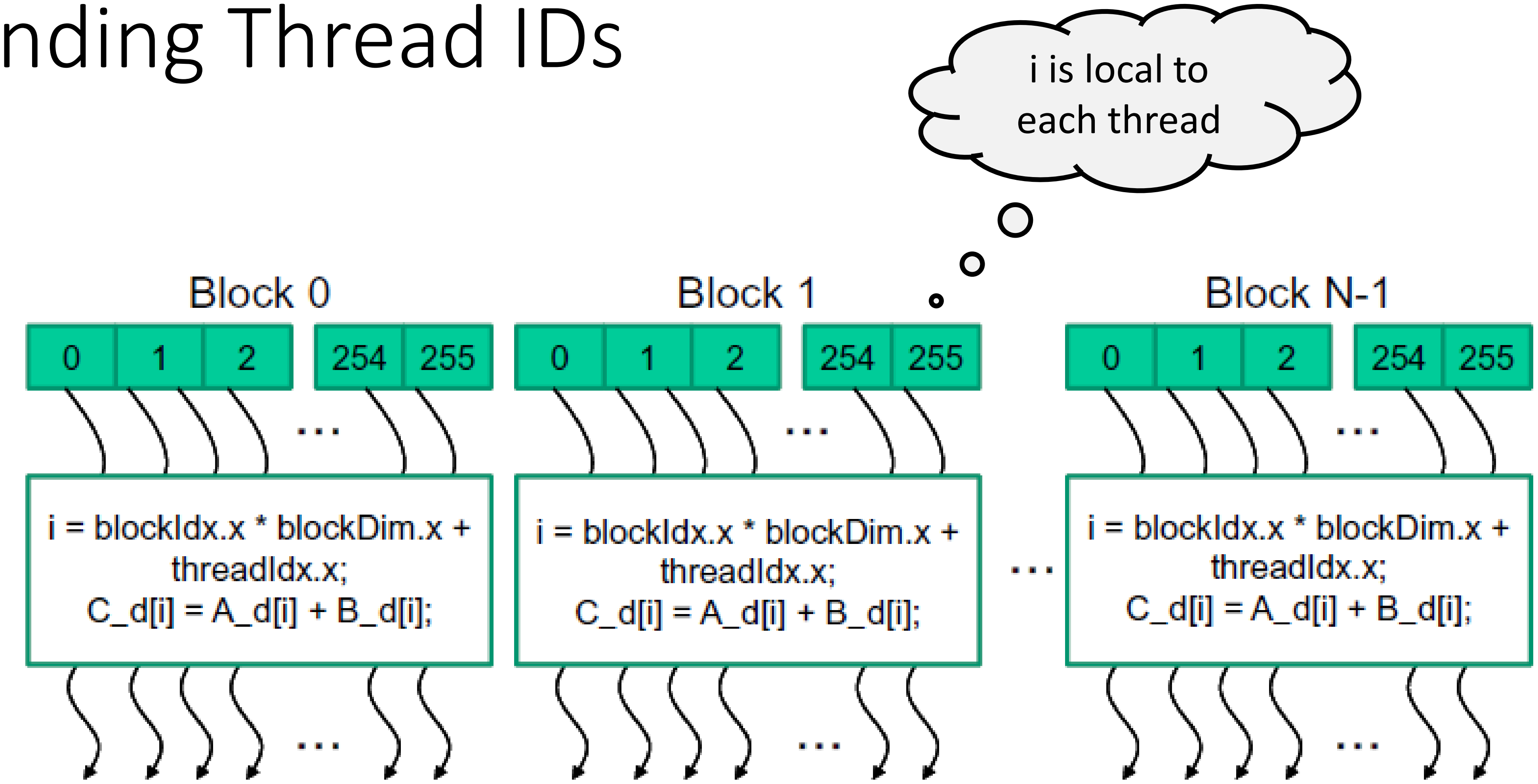
- `threadIdx` is a 3-component vector
  - Thread index can be 1D, 2D, or 3D
  - Thread blocks as a result can be 1D, 2D, or 3D
- How to find out the relation between thread ids and `threadIdx`?
  - 1D: `tid = threadIdx.x`
  - 2D block of size `(Dx, Dy)`: thread ID of a thread of index `(x, y)` is `(x + yDx)`
  - 3D block of size `(Dx, Dy, Dz)`: thread ID of a thread of index `(x, y, z)` is `(x + yDx + zDxDy)`

Thread Hierarchy

- Threads in a block reside on the same core, max 1024 threads in a block
- Thread blocks are organized into 1D, 2D, or 3D grids
  - Also called cooperative thread array
  - Grid dimension is given by `gridDim` variable
- Identify block within a grid with the `blockIdx` variable
  - Block dimension is given by `blockDim` variable



Finding Thread IDs



Determining Block Dimensions

- Assume a block with a maximum of 1024 allowed threads

Variable `blockDim`	Valid/Invalid
(512,1,1)	✓
(8, 16, 4)	✓
(32, 16, 2)	✓
(32, 32, 32)	✗

Find Device Information

```
int count;
cudaError_t err =
            cudaGetDeviceCount(&count);
if (err != cudaSuccess) {
  cerr << cudaGetErrorString(err) << endl;
}

cudaDeviceProp Props;
for (int i = 0; i < count; i++) {
  err = cudaGetDeviceProperties(&Props, i);
}
```

Device number: 3

Device name: GeForce GTX 1080 Ti

Integrated or discrete GPU? discrete

Clock rate: 1544 MHz

Compute capability: 6.1

Number of SMs: 28

Total number of CUDA cores: 3584

Max threads per SM: 2048

Max threads per block: 1024

Warp size: 32

Max grid size (i.e., max number of blocks): [2147483647,65535,65535]

Max block dimension: [1024,1024,64]

Total global memory: 11172 MB

Shared memory per SM: 96 KB

32-bit registers per SM: 65536

Shared mem per block: 48 KB

Registers per block: 65536

Total const mem: 64 KB

L2 cache size: 2816 KB

Device Management

- Application can query and select GPUs
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDevice(int *device)`
  - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`

- Multiple host threads can share a device
- A single host thread can manage multiple devices
  - `cudaSetDevice(i)` to select current device
  - `cudaMemcpy(…)` for peer-to-peer copies

Launching Kernels

```
// 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);
  ...
}
```

Execution Configuration

- Assume data is of length N, and say the kernel execution configuration is <<<N/TPB, TPB>>>
  - Each block has TPB threads
  - There are N/TPB blocks
- Suppose N = 64 and TPB = 32
  - Implies there are 2 blocks of 32 threads
- Dimension variables are vectors of integral type

Launching Kernels

```
// 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;
  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);
  ...
}
```

Execution Configuration Uses Integer Arithmetic

- Assume data is of length N, and say the kernel execution configuration is <<<N/TPB, TPB>>>
  - Each block has TPB threads
  - There are N/TPB blocks
- Suppose N = 64 and TPB = 32
  - Implies there are 2 blocks of 32 threads
- Dimension variables are vectors of integral type
- Now assume N = 65

So now what?

Execution Configuration Uses Integer Arithmetic

- Ensure that the grid covers the array length
- One strategy is to change the number of blocks from N/TPB to (N+TPB-1)/TPB to ensure rounding up

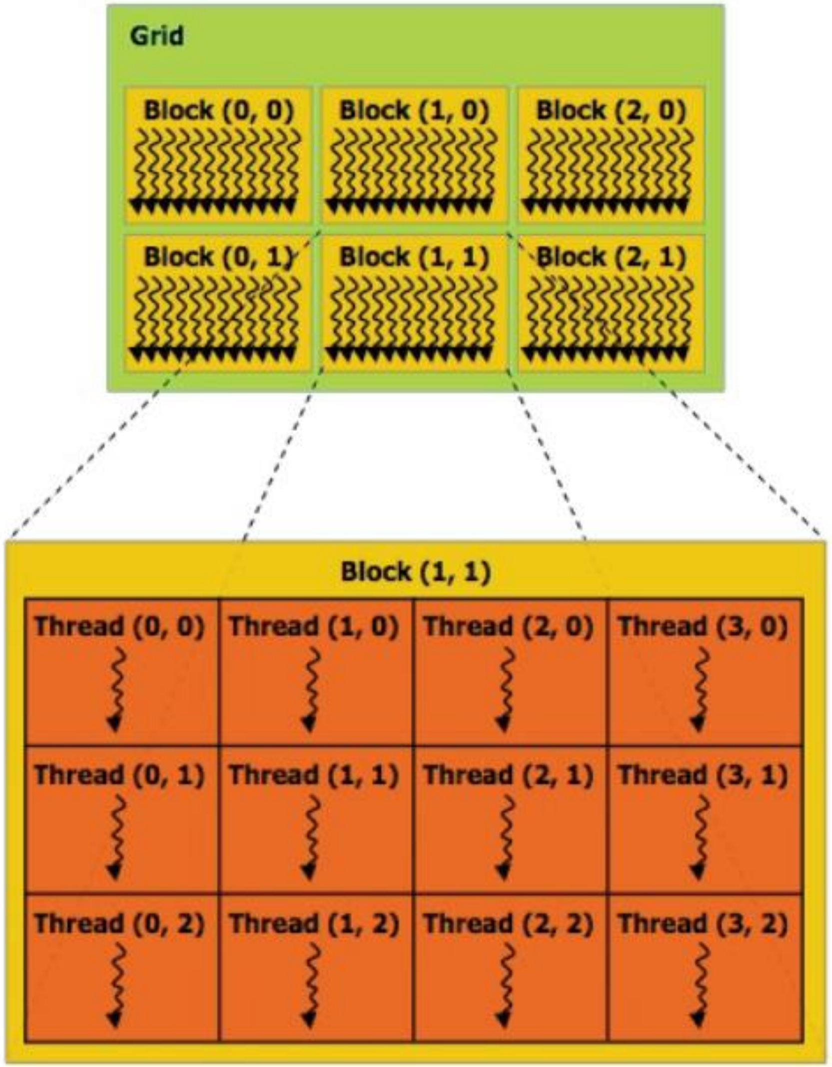
Execution Configuration Uses Integer Arithmetic

- Ensure that the grid covers the array length
- One strategy is to change the number of blocks from N/TPB to (N+TPB-1)/TPB to ensure rounding up

- This means that a thread index can exceed the maximum array index
- Many examples use a control statement in the kernel to check for such corner cases

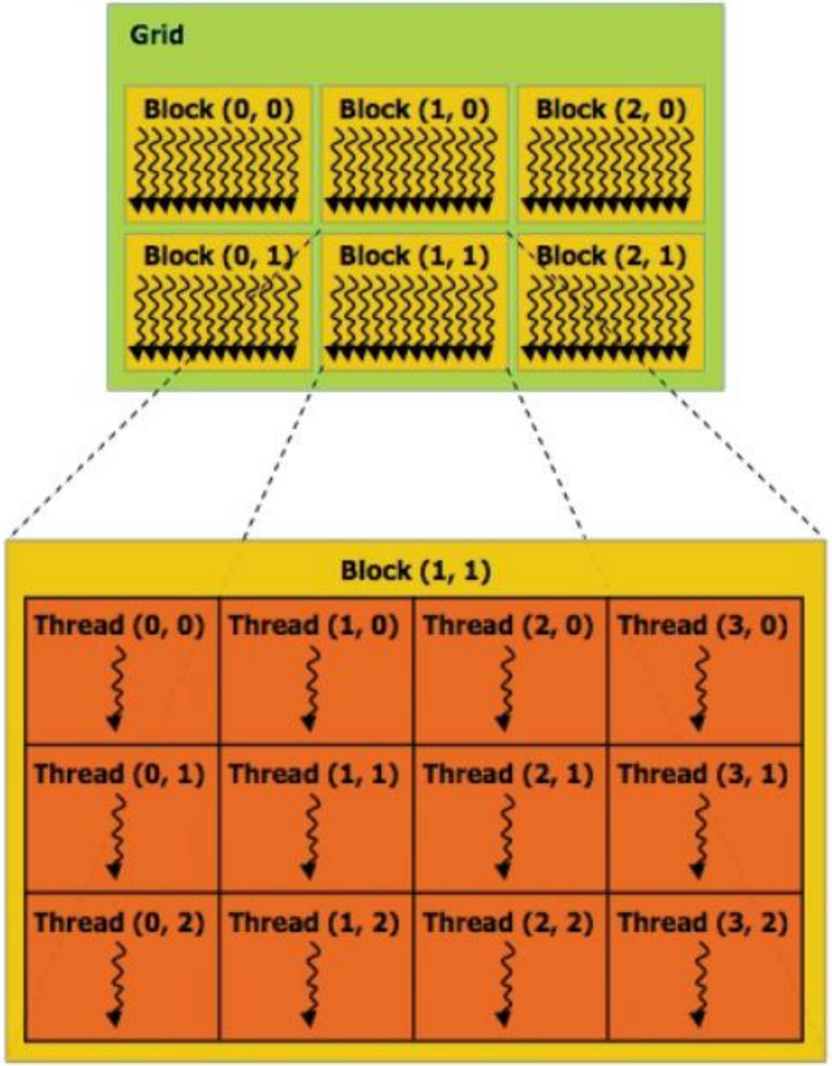
What should be numBlocks?

```
const int Nx = 11; // not a multiple of
threadsPerBlock.x
const int Ny = 5; // not a multiple of
threadsPerBlock.y

//////////////////////////////////////////////
dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(x, y, z);

// assume A, B, C are allocated Nx x Ny float arrays
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```



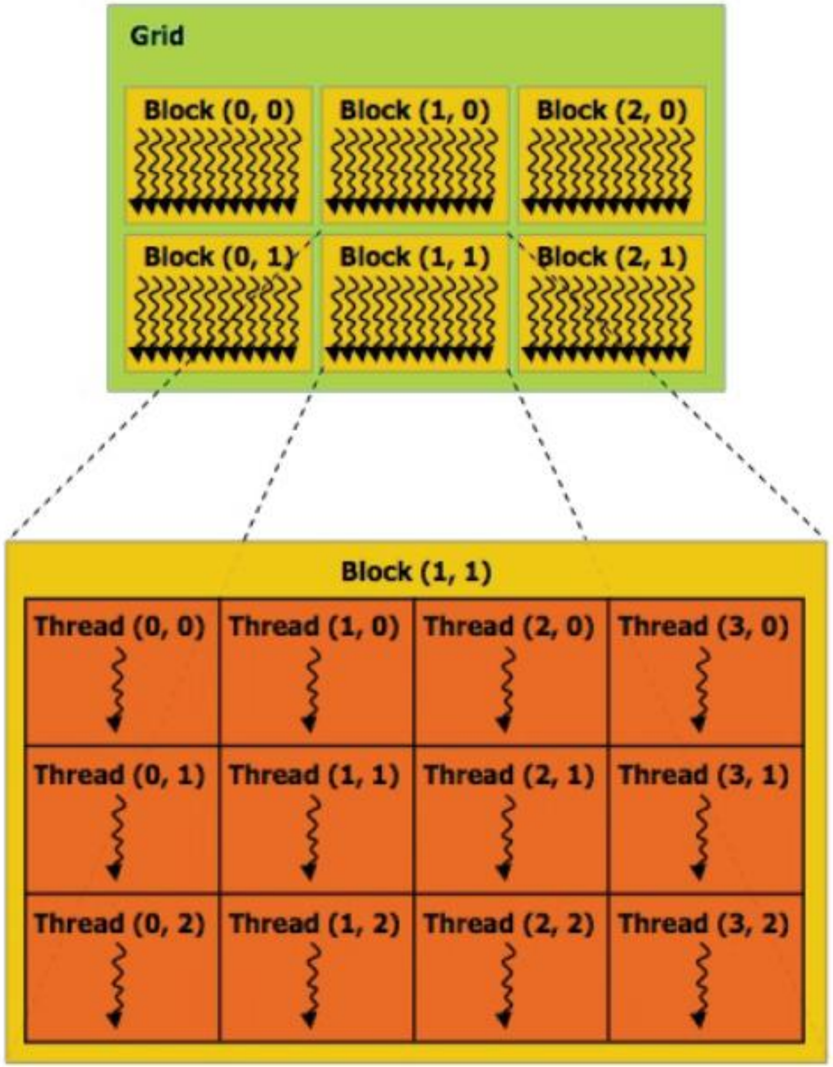
What should be numBlocks?

```
const int Nx = 11; // not a multiple of
threadsPerBlock.x
const int Ny = 5; // not a multiple of
threadsPerBlock.y

//////////////////////////////////////////////
dim3 threadsPerBlock(4, 3, 1);
dim3
numBlocks((Nx+threadsPerBlock.x-1)/threadsPerBlock.x,
(Ny+threadsPerBlock.y-1)/threadsPerBlock.y,
1);

// assume A, B, C are allocated Nx x Ny float arrays
// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```



Example

```
__global__ void matrixAdd(float* A,
                          float* B, float* C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  // Guard against out of bounds array access
  if (i < N && j < N)
    C[i+N*j] = A[i+N*j] + B[i+N*j];
}
```



Find Thread IDs

```
__global__ void thread_details(uint32_t * const block, uint32_t * const thr
ead, uint32_t * const warp, uint32_t * const calc_thread) {
  const uint32_t thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
  block[thread_idx] = blockIdx.x;
  thread[thread_idx] = threadIdx.x;
  warp[thread_idx] = threadIdx.x / warpSize;
  calc_thread[thread_idx] = thread_idx;
}

thread_details<<<num_blocks, num_threads>>>(gpu_block, gpu_thread,
                                            gpu_warp, gpu_calc_thread);
```

Matrix Multiplication Example

```
int main() {
  int SIZE = N * N;
  cudaError_t status;

  float *hostA, *hostB, *hostC;
  hostA = (float*)malloc(SIZE * sizeof(f
loat));
  hostB = (float*)malloc(SIZE * sizeof(f
loat));
  hostC = (float*)malloc(SIZE * sizeof(f
loat));
```

```
  float *deviceA, *deviceB, *deviceC;
  status = cudaMalloc((void**)&deviceA,
SIZE * sizeof(float));
  if (status != cudaSuccess) {
    cerr << cudaGetErrorString(status) <
< endl;
  }


  status = cudaMalloc((void**)&deviceB,
SIZE * sizeof(float));
  status = cudaMalloc((void**)&deviceC,
SIZE * sizeof(float));
```

Matrix Multiplication Example

```
  status = cudaMemcpy(deviceA, hostA, SI
ZE * sizeof(float), cudaMemcpyHostToDevi
ce);
  status = cudaMemcpy(deviceB, hostB, SI
ZE * sizeof(float), cudaMemcpyHostToDevi
ce);

  dim3 blocksPerGrid(1, 1);
  dim3 threadsPerBlock(N, N);
  matmulKernel<<<blocksPerGrid, threadsP
erBlock>>>(deviceA, deviceB, deviceC);
```

```
  cudaMemcpy(hostC, deviceC, SIZE * size
of(float), cudaMemcpyDeviceToHost);

  …

  cudaFree(deviceA);
  cudaFree(deviceB);
  cudaFree(deviceC);

  free(hostA);
  free(hostB);
  …
}
```

Matrix Multiplication Example

```
__global__ void matmulKernel(float* A, float* B, float* C) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;

  float tmp = 0;
  if (i < N && j < N) {
    // Each thread computes one element of the matrix
    for (int k = 0; k < N; k++) {
      tmp += A[i * N + k] * B[k * N + j];
    }
  }
  C[i * N + j] = tmp;
}
```

Choosing Optimal Execution Configuration

- 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
  - It is okay to have a much greater number of threads
- No fixed rule, needs exploration and experimentation
- Choose number of threads in a block to be some multiple of 32

Timing a CUDA Kernel

```
float memsettime;
cudaEvent_t start, stop;
// initialize CUDA timer
cudaEventCreate(&start);  cudaEventCreate(&stop);
cudaEventRecord(start,0);

// CUDA Kernel
…

cudaEventRecord(stop,0); // stop CUDA timer
cudaEventSynchronize(stop);
cudaEventElapsedTime(&memsettime,start,stop); // in milliseconds
std::cout << "Kernel execution time: " << memsettime << "\n";
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself or error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

Mapping Blocks and Threads

- A GPU executes one or more kernel grids
- When a CUDA kernel is launched, the thread blocks are enumerated and distributed to SMs
  - Potentially >1 block per SM
- An SM executes one or more thread blocks
  - Each GPU has a limit on the number of blocks that can be assigned to each SM
  - For example, a CUDA device may allow up to eight blocks to be assigned to each SM
  - Multiple thread blocks can execute concurrently on one SM

Mapping Blocks and Threads

- The threads of a block execute concurrently on one SM
  - CUDA cores in the SM execute threads
- A block begins execution only when it has secured all execution resources necessary for all the threads
- As thread blocks terminate, new blocks are launched on the vacated multiprocessors
- Blocks are mostly not supposed to synchronize with each other
  - Allows for simple hardware support for data parallelism

Mapping Blocks and Threads

- The threads of a block execute concurrently on one SM
  - CUDA cores in the SM execute threads
- A block begins execution only when it has secured all execution res
- As ed mu

CUDA runtime can execute blocks in any order

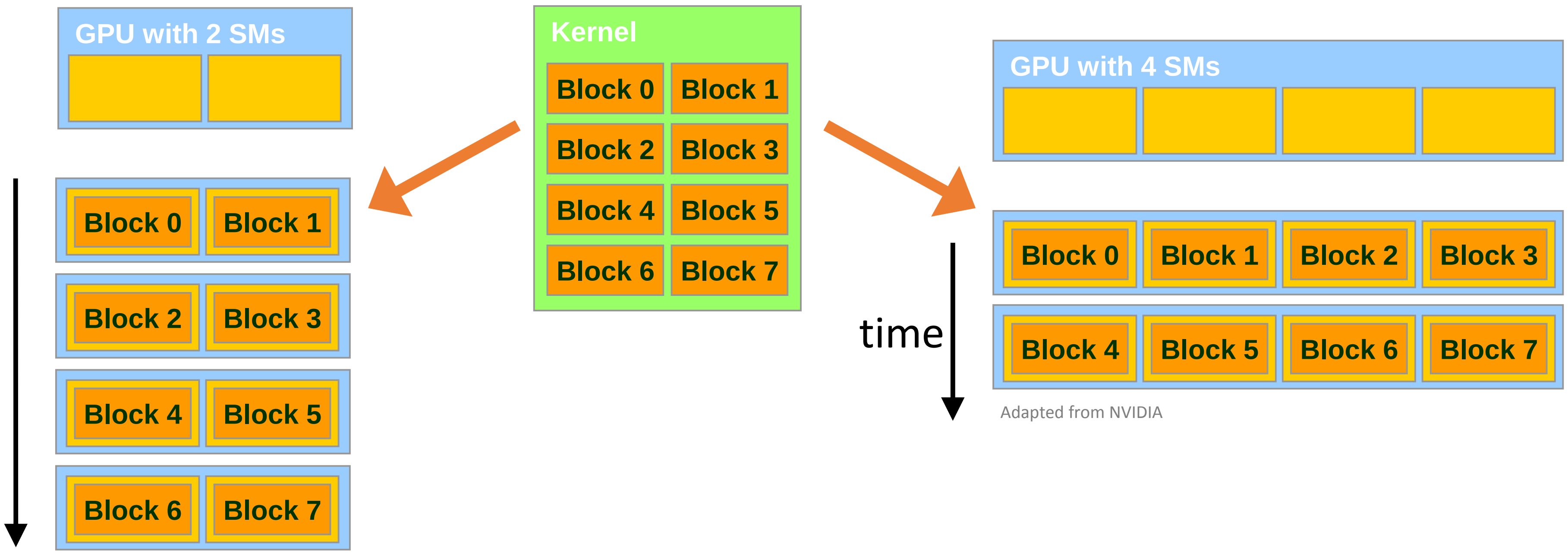
- Blocks are mostly not supposed to synchronize with each other
  - Allows for simple hardware support for data parallelism

Scheduling Blocks

- Number of threads that can be simultaneously tracked and scheduled is bounded
  - Requires resources for an SM to maintain block and thread indices and their execution status
- Up to 2048 threads can be assigned to each SM on recent CUDA devices
  - For example, 8 blocks of 256 threads, or 4 blocks of 512 threads
- Assume a CUDA device with 28 SMs
  - Each SM can accommodate up to 2048 threads
  - The device can have up to 57344 threads simultaneously residing in the device for execution

Block Scalability

- Hardware can assign blocks to SMs in any order
  - A kernel with enough blocks scales across GPUs
  - Not all blocks may be resident at the same time

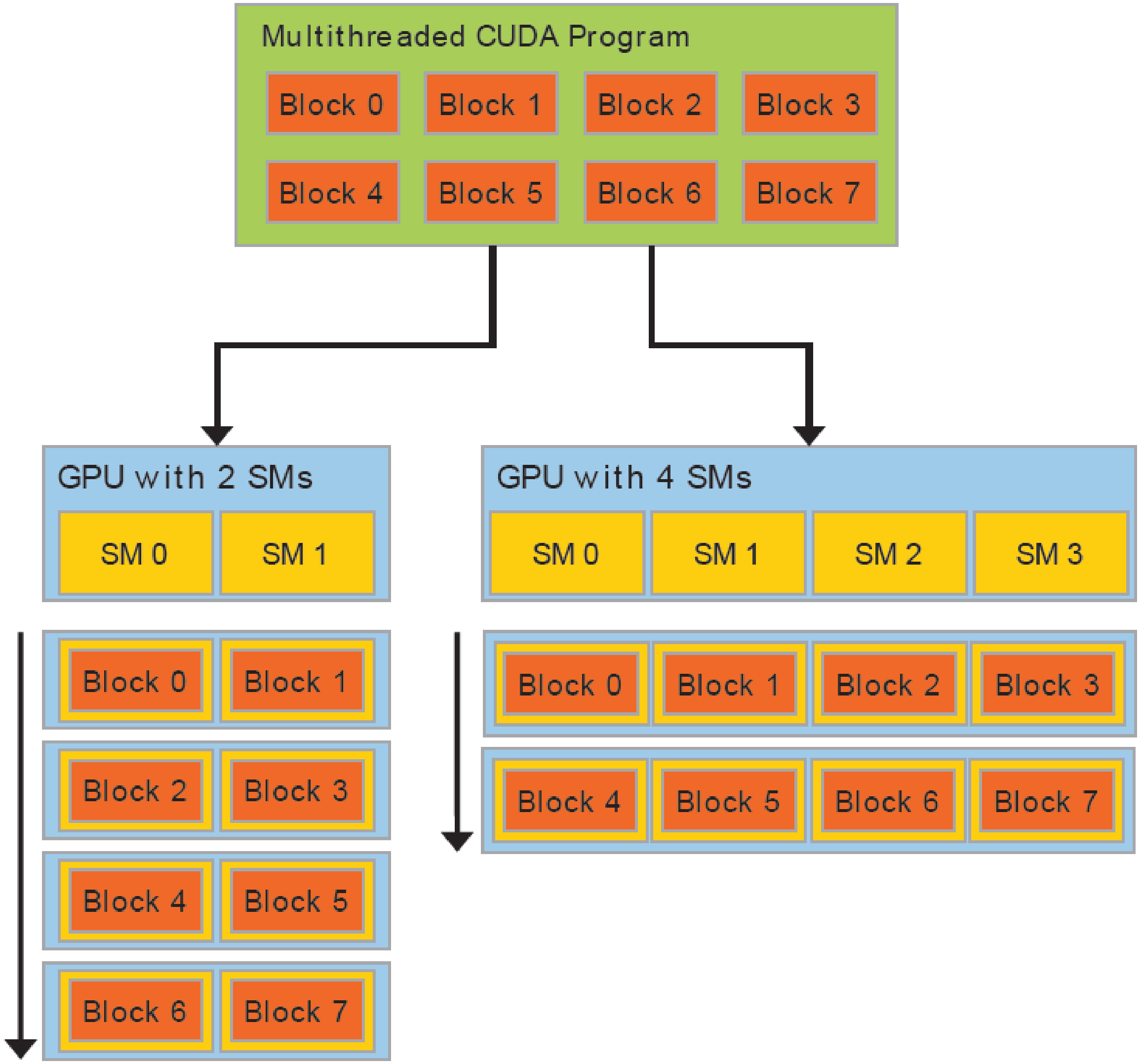


Adapted from NVIDIA

Scalability of GPU Architecture

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.



Thread Warps

- Conceptually, threads in a block can execute in any order
- Sharing a control unit among compute units reduce hardware complexity, cost, and power consumption
- A set of consecutive threads (currently 32) that execute in SIMD fashion (wavefront in AMD with 64 threads)
- Warps are scheduling units in an SM
    - Part of the implementation in NVIDIA, not the programming model

Thread Warps

- All threads in a warp run in lockstep
  - Warps share an instruction stream
  - Same instruction is fetched for all threads in a warp during the instruction fetch cycle
    - Prior to Volta, warps used a single shared program counter
  - In the execution phase, each thread will either execute the instruction or will execute nothing
  - Individual threads in a warp have their own instruction address counter and register state

Thread Warps

- Warp threads are fully synchronized
  - There is an implicit barrier after each step/instruction

- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
  - Each Block is divided into 256/32 = 8 warps
  - There are 8 * 3 = 24 warps

Thread Divergence

- If some threads take the if branch and other threads take the else branch, they cannot operate in lockstep
  - Some threads must wait for the others to execute
  - Renders code at that point to be serial rather than parallel
- Divergence occurs only within a warp
- The programming model does not prevent thread divergence
  - Performance problem at the warp level

Parallelism in GPUs

- Two levels of parallelism
  - Concurrent thread blocks
    - Coarse-grained data parallelism or task parallelism
  - Concurrent warps
    - Use several threads per block
    - Fine-grained data parallelism or thread parallelism

Scheduling Thread Warps

- Each SM launches warps of threads, and excutes threads on a timesharing basis
  - Timesharing is implemented in hardware, not software
- SM schedules and executes warps that are ready to run
  - Warps run for fixed-length time slices like processes
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Selection of ready warps for execution does not introduce any idle time into the execution timeline
    - Zero-overhead scheduling in hardware

Scheduling Thread Warps

- If more than one warp is ready for execution, a priority mechanism is used to select one for execution
- Thread blocks execute on an SM, thread instructions execute on a core
- CUDA virtualizes the physical hardware
  - Thread is a virtualized scalar processor (registers, PC, state)
  - Block is a virtualized multiprocessor (threads, shared memory)

Scheduling Thread Warps

- Suppose an instruction executed by a warp has to wait for the result of a previously initiated long-latency operation
  - The warp is not selected for execution
  - Another warp that is not waiting for results is selected for execution

- Hide latency of long operations with work from other threads
  - Called latency tolerance or latency hiding

Scheduling Thread Warps

- Goal is to have enough threads and warps around to utilize hardware in spite of long-latency operations
  - GPU hardware will likely find a warp to execute at any point in time
  - With warp scheduling, the long waiting time of warp instructions is “hidden” by executing instructions from other warps

- As warps and thread blocks complete, resources are freed

Question

- Assume that a CUDA device allows up to 8 blocks and 1024 threads per SM, whichever becomes a limitation first
  - It allows up to 512 threads in each block
- Say for the matrix-matrix multiplication kernel, should we use 8x8, 16x16, or 32x32 thread blocks?

Explanation

- 8x8 threads/block
  - If we use 8x8 blocks, each block would have only 64 threads
  - We will need 1,024/64=16 blocks to fully occupy an SM
  - Since there is a limitation of up to 8 blocks in each SM, we have 64x8 = 512 threads/SM
  - There will be fewer warps to schedule around long-latency operations
  - Implies that the SM execution resources will likely be underutilized
- 16x16 threads/block
  - 16x16 blocks give 256 threads per block
  - Each SM can take 1024/256=4 blocks, which is within the 8-block limitation
  - Reasonable configuration since we have full thread capacity in each SM and a maximal number of warps for scheduling around the long-latency operations
- 32x32 threads/block
  - 32x32 blocks give 1024 threads in each block, exceeding the limit of 512 threads per block

SIMT Architecture

- GPUs employ SIMD hardware to exploit the data-level parallelism
- In vectorization, users program SIMD hardware directly
- CUDA features a MIMD-like programming model
  - Launch large number of threads
  - Each thread can have its own execution path and access arbitrary memory locations
  - At runtime, the GPU hardware executes warps in lockstep
  - Exploits regularity and spatial locality on GPU SIMD hardware
- This execution model is called single-instruction multiple-thread (SIMT)

SIMT Architecture

- Very similar in flavor to SIMD
  - In SIMD, you program with the vector width in mind
  - Possibly use auto-vectorization or intrinsics
- SIMT can be thought of as SIMD with multithreading
  - Software analog compared to the hardware perspective of SIMD
  - For e.g., we rarely need to know the number of cores with CUDA

SIMD vs SPMD

SIMD	SPMD
• Processing units are executing the same instruction at any instant	• Parallel processing units execute the same program on multiple parts of the data • All the processing units may not execute the same instruction at the same time

Memory Hierarchy

Memory Access Efficiency

- Compute to global memory access ratio
  - Number of floating-point operations performed for each access to global memory

```
for (int i = 0; i < N; i++)
  tmp += A[i*N+K]*B[k*N+j];
```

- Assume a GPU device with 800 GB/s global memory bandwidth and peak single-precision performance of 1500 GFLOPS
  - What is the performance we expect?

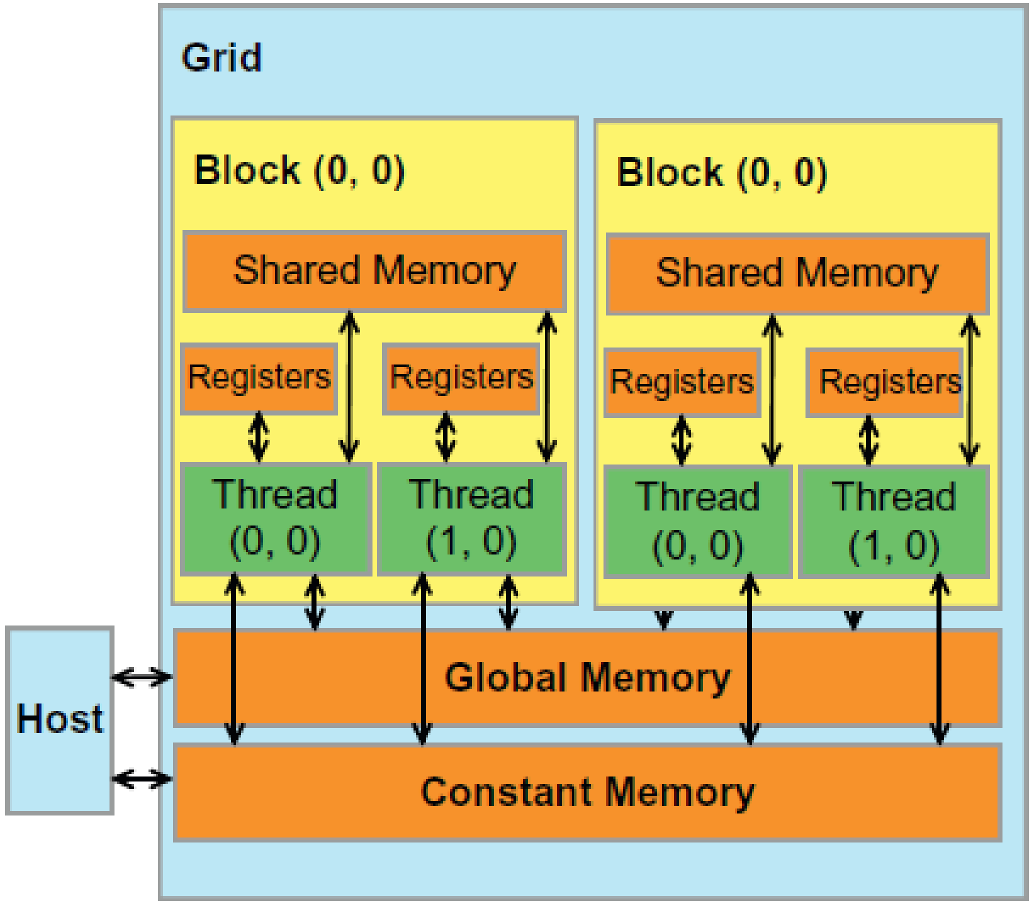
Memory Hierarchy in CUDA

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories



Variable Type Qualifiers in CUDA

	Memory	Scope	Lifetime
`int localVar`	Register	Thread	Kernel
`__device__ __local__ int localVar`	Local	Thread	Kernel
`__device__ __shared__ int sharedVar;`	Shared	Block	Kernel
`__device__ int globalVar`	Global	Grid	Application
`__device__ __constant__ int constVar`	Constant	Grid	Application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
    - Except arrays that reside in local memory
- Pointers can only point to memory allocated or declared in global memory

Memory Organization

- Host and device maintain their own separate memory spaces
  - A variable in CPU memory may not be accessed directly in a GPU kernel
- It is programmer's responsibility to keep them in sync
  - A programmer needs to maintain copies of variables

Registers

- 64K 32-bit registers per SM
  - So 256KB register file per SM, a CPU in contrast has a few (1-2 KB) per core
- Up to 255 registers per thread
- If a code uses the maximum number of registers per thread (255) and an SM has 64K of registers then the SM can support a maximum of 256 threads
- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread

Registers

- 64K 32-bit registers per SM
  - So 256KB register file per SM, a CPU in contrast has a few (1-2 KB) per core
- Up to 255 registers per thread
- If a code us ... d (255) and an SM has ... kimum of 256 threads

> What if each thread uses 33 registers?

- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread

Registers

- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread
- What if each thread uses 33 registers?
  - Fewer threads, reduce number of blocks

- There is a big difference between “fat” threads which use lots of registers, and “thin” threads that require very few!

Shared Memory

- Shared memory aims to bridge the gap in memory speed and access
  - Also called scratchpad memory
  - Usually 16-64KB of storage that can be accessed efficiently by all threads in a block
- Primary mechanism in CUDA for efficiently supporting thread cooperation
- Each SM contains a single shared memory
  - Resides adjacent to an SM, on-chip
  - The space is shared among all blocks running on that SM

Shared Memory

- Variable in shared memory is allocated using the `__shared__` specifier
  - Faster than global memory
  - Can be accessed only by threads within a block
- **Amount of shared memory per block limits occupancy**

- Say an SM with 4 thread blocks has 16 KB of shared memory

```
__shared__ float min[256];
__shared__ float max[256];
__shared__ float avg[256];
__shared__ float stdev[256];
```

Registers vs Shared Memory

Registers	Shared Memory
• Faster than shared memory • Private to a thread	• On-chip memory space, requires load/store operations • Visible to all threads in a block

Global Variables

- Variable `lock` can be accessed by both kernels
  - Resides in global memory space
  - Can be both read and modified by all threads

```
__device__ int lock=0;

__global__ void kernel1(...) {
 // Kernel code
}
__global__ void kernel2(...) {
  // Kernel code
}
```

Global Memory

- On-device memory accessed via 32, 64, or 128 B transactions
- An warp executes an instruction that accesses global memory
  - The addresses are coalesced into transactions
  - Number of transactions depend on the access size and distribution of memory addresses
  - More transactions mean less throughput
    - For example, if 32 B transaction is needed for a thread’s 4 B access, throughput is essentially 1/8th

Constant Memory

- Use for data that will not change during kernel execution
  - Constant memory is 64KB
- Constant memory is cached
  - Each SM has a read-only constant cache that is shared by all cores in the SM
  - Used to speed up reads from the constant memory space which resides in device memory
  - Read from constant memory incurs a memory latency on a miss
    - Otherwise, it is a read from constant cache, which is almost as fast as registers

Constant Variables

- Constant variables can't be modified by kernels
  - Reside in constant memory
  - Accessible from all threads within a grid
- They are defined with global scope within the kernel using the prefix `__constant__`
- Host code can access via `cudaMemcpyToSymbol()` and `cudaMemcpyFromSymbol()`

Local Memory

- Local memory is off-chip memory
  - More like thread-local global memory, so it requires memory transactions and consumes bandwidth
- Automatic variables are placed in local memory
  - Arrays for which it is not known whether indices are constant quantities
  - Large structures or arrays that consume too much register space
  - In case of register spilling
- Inspect PTX assembly code (compile with `–ptx`)
  - Check for `ld.local` and `st.local` mnemonic

Device Memory Management

- Global device memory can be allocated with `cudaMalloc()`
- Freed by `cudaFree()`
- Data transfer between host and device is with `cudaMemcpy()`
- Initialize memory with `cudaMemset()`

- There are asynchronous versions

GPU Caches

- GPUs have L1 and L2 data caches on devices with CC 2.x and higher
  - Texture and constant cache are available on all devices
- L1 cache is per SM
  - Shared memory is partitioned out of unified data cache and its size can be configured, remaining portion is the L1 cache
  - Can be configured as 48 KB of shared memory and 16 KB of L1 cache, or 16 KB of shared memory and 48 KB of L1 cache, or 32 KB each
  - L1 caches are 16-48 KB
- L2 cache is shared by all SMs

GPU Caches

- L1 cache lines are 128 B wide in Fermi onward, while L2 lines are 32 B

CPU Caches vs GPU caches

CPU	GPU

• Data is automatically moved by hardware between caches
  • Association between threads and cache does not have to be exposed to programming model
• Caches are generally coherent
	• Data movement must be orchestrated by programmer
	  • Association between threads and storage is exposed to programming model
	• L1 cache is not coherent, L2 cache is coherent

Synchronization in CUDA

Race Conditions and Data Races

- A race condition occurs when program behavior depends upon relative timing of two (or more) event sequences
- Execute: `*c += sum;`
  - Read value at address `c`
  - Add `sum` to value
  - Write result to address `c`

Be Careful to Avoid Race Conditions!

time

Thread 0, Block 0	Thread 3, Block 7
• Read value at address c	
	• Read value at address c
	• Add sum to value
	• Write result to address c
• Add sum to value	
• Write result to address c	

Synchronization Constructs in CUDA

1. `__syncThreads()` synchronizes threads within a block
2. `cudaDeviceSynchronize()` synchronizes all threads in a grid
   - There are other variants
3. Atomic operations prevent conflicts associated with multiple threads concurrently accessing a variable
   - Atomic operations on both global memory and shared memory variables
   - For e.g., `float atomicAdd(float* addr, float amount)`

`__syncthreads()`

- A `__syncthreads()` statement must be executed by all threads in a block
- `__syncthreads()` is in an `if` statement
  - Either all threads in the block execute the path that includes the `__syncthreads()` or none of them does
- `__syncthreads()` statement is in each path of an `if-then-else` statement
  - Either all threads in a block execute the `__syncthreads()` on the then path or all of them execute the else path
  - The two `__syncthreads()` are different barrier synchronization points

Synchronization Between Grids

- For threads from different grids, system ensures writes from kernel happen before reads from subsequent grid launches

Atomic Operations

- Perform read-modify-write (RMW) atomic operations on data residing in global or local memory
  - `atomicAdd()`, `atomicSub()`, `atomicMin()`, `atomicMax()`, `atomicInc()`, `atomicDec()`, `atomicExch()`, `atomicCAS()`
- Predictable result when simultaneous access to memory required

Overlap Host and Device Computation

```
cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice)
kernel1<<<1,N>>>(d_a);
cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/
https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

Overlap Host and Device Computation

```
cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice)
kernel1<<<1,N>>>(d_a);
cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

```
cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice)
kernel1<<<1,N>>>(d_a);
h_func(h_b);
cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Utilize GPU Hardware

- Overlap kernel execution with memory copy between host and device
- Overlap execution of multiple kernels if there are resources

- Depends on whether the GPU architecture supports overlapped execution

CUDA Streams

- Sequence of operations that execute on the device in the order in which they were issued by the host
  - Operations across streams can interleave and run concurrently
- All GPU device operations run in the default “null” stream
  - Default stream is synchronizing
  - No operation in the default stream will begin until all previously issued operations in any stream have completed
  - An operation in the default stream must complete before any other operation in any stream will begin
  - CUDA 7 provides options to change behavior of default streams

Non-Default Streams

- Operations in a non-default stream are non-blocking with host
- Use `cudaDeviceSynchronize()`
  - Blocks host until all previously issued operations on the device have completed
- Cheaper alternatives
  - `cudaStreamSynchronize()`, `cudaEventSynchronize()`, …

```
cudaStream_t stream1;

cudaError_t res;

res = cudaStreamCreate(&stream1);

res = cudaMemcpyAsync(d_a, a, N,
cudaMemcpyHostToDevice, stream1);

increment<<<1,N,0,stream1>>>(d_a);

res = cudaStreamDestroy(&stream1);
```

Why Use CUDA Streams?

- Memory copy and kernel execution can be overlapped in they occur in different, non-default streams
  - Check for GPU device capabilities
- Individual kernels can overlap if there are enough resources on the GPU

Performance Bottlenecks with CUDA

Key Ideas for Performance

- Try and reduce resource consumption
- Strive for good locality
  - Use tiling to exploit shared memory
  - Copy blocks of data from global memory to shared memory and operate on them (for e.g., matrix multiplication kernel)
  - Improve throughput by reducing global memory traffic
- Exploit SIMT
  - Reduce thread divergence in a warp
- Memory access optimization
  - Global memory: memory coalescing
  - Shared memory: avoid bank conflicts

What can we say about this code?

```
__global__ void dkernel(float *vector, int vectorsize) {
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  switch (id) {
    case 0: vector[id] = 0; break;
    case 1: vector[id] = vector[id] * 10; break;
    case 2: vector[id] = vector[id - 2]; break;
    case 3: vector[id] = vector[id + 3]; break;
    …
    case 31: vector[id] = vector[id] * 9; break;
  }
}
```

Deal with Thread Divergence

- Thread divergence renders execution sequential
  - SIMD hardware takes multiple passes through the divergent paths

```
if (threadIdx.x / WARP_SIZE > 2) {}
```

```
if (threadIdx.x > 2) {}
```

Deal with Thread Divergence

- Condition evaluating to different truth values is not bad

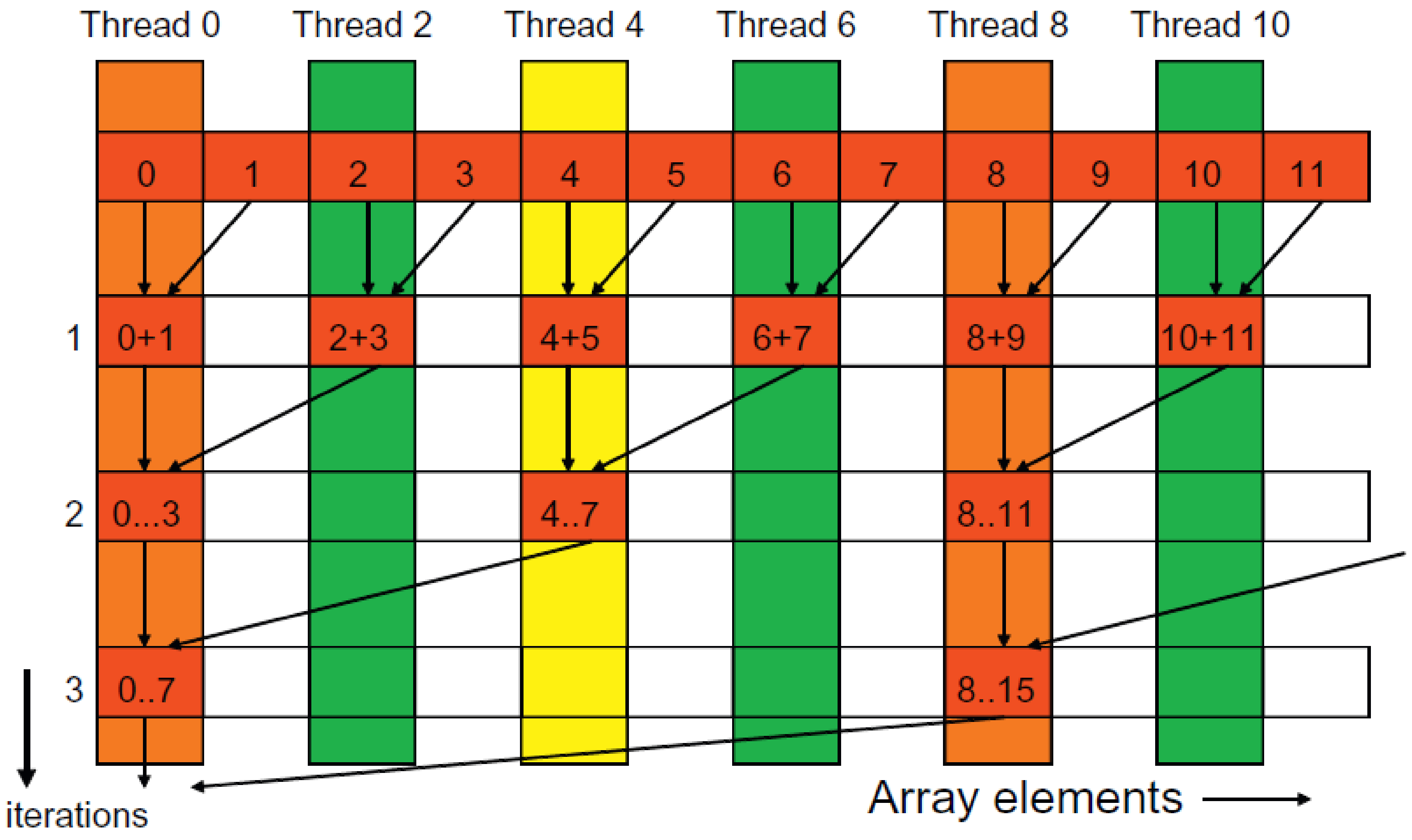
```
if (threadIdx.x / WARP_SIZE > 2) {}
```

  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
- Conditions evaluating to different truth-values for threads in a warp is bad

```
if (threadIdx.x > 2) {}
```

  - Creates two different control paths for threads in a block; branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp

Implement a Reduction Kernel in CUDA



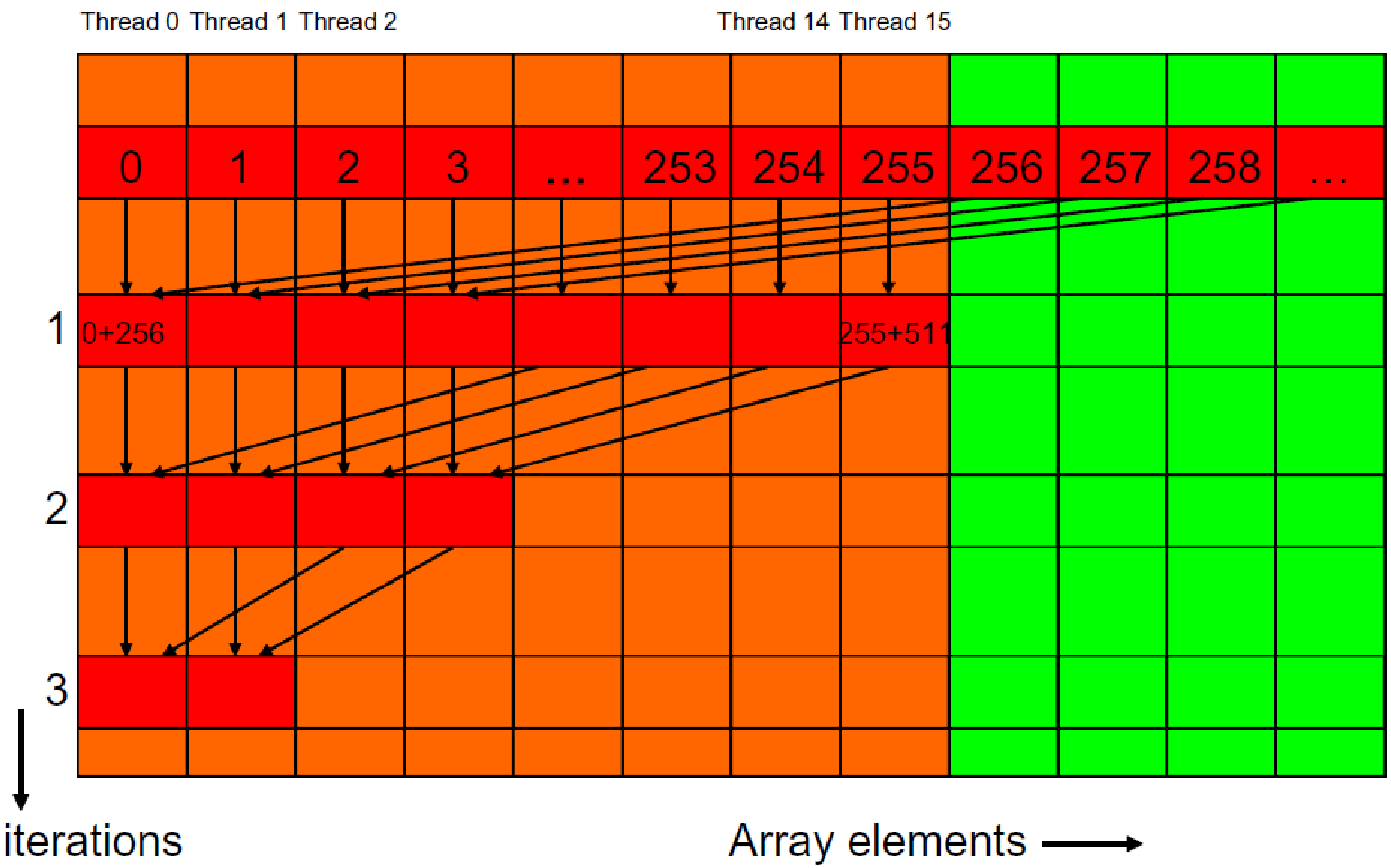
Reduction Kernel

```
__shared__ float partialSum[];
…
unsigned int t = threadIdx.x;
for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)  {
  __syncthreads();
  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
}
```

Reduction Kernel

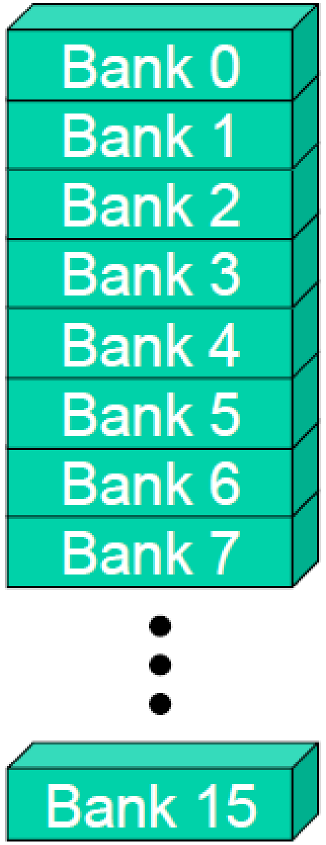
```
__shared__ float partialSum[];
…
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x; stride > 1; stride /= 2) {
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

Execution of the Revised Kernel



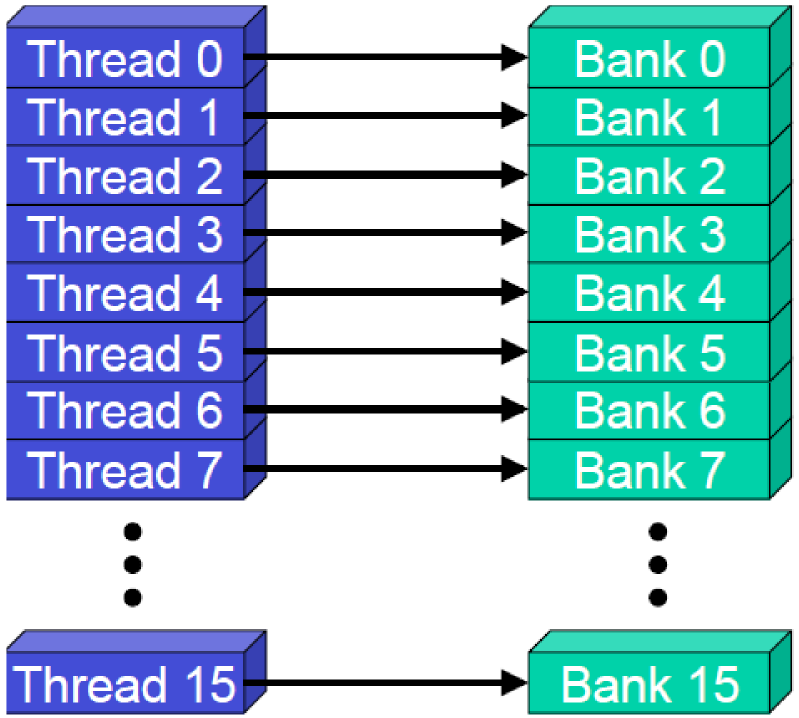
Parallel Memory Architecture

- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized

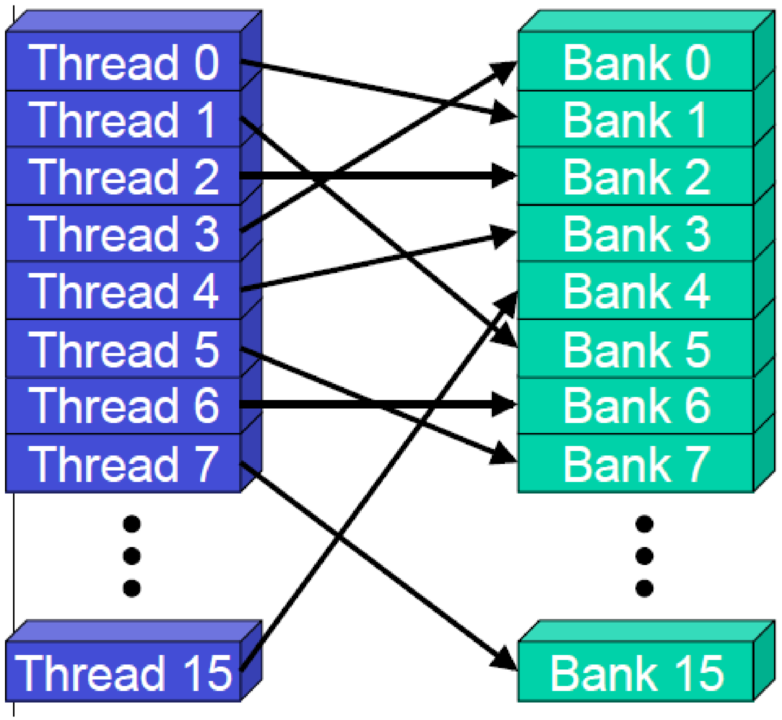


Example of Bank Addressing

- No bank conflicts
  - Linear addressing, stride=1

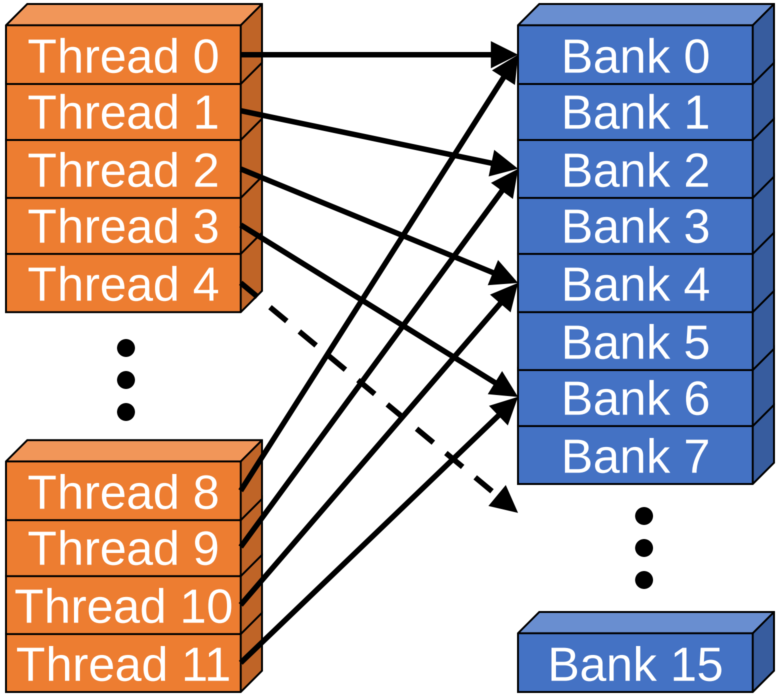


- No bank conflicts
  - Random permutation

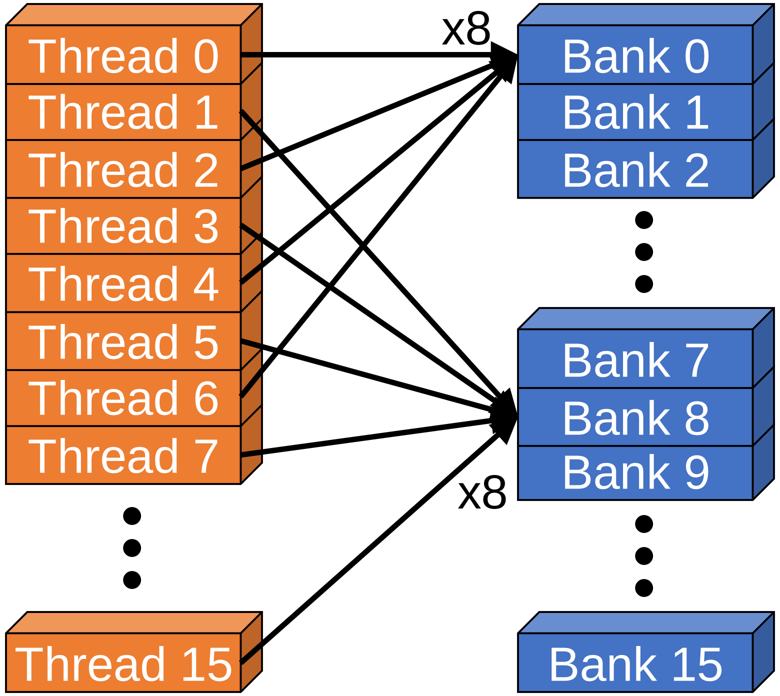


Example of Bank Addressing

- 2-way Bank Conflicts
  - Linear addressing, stride = 2



- 8-way Bank Conflicts
  - Linear addressing, stride = 8



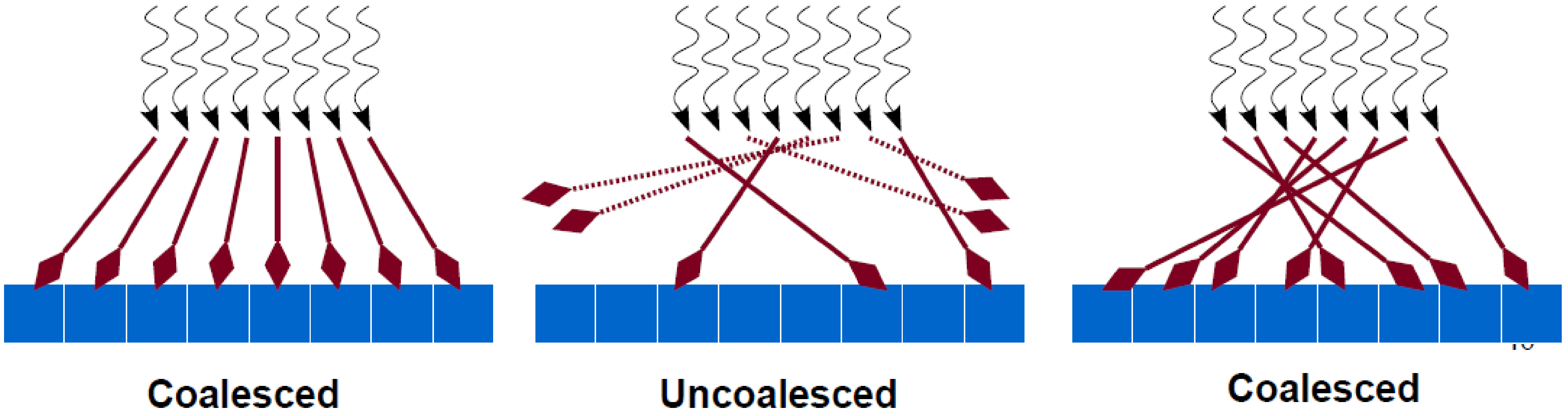
Shared Memory Bank Conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- Fast case
    - If all threads of a warp access different banks, there is no bank conflict
    - If all threads of a warp access the identical address, there is no bank conflict
- Slow case
    - Bank Conflict: multiple threads in the same half-warp access the same bank
    - Must serialize the accesses
    - Cost = max # of simultaneous accesses to a single bank

Memory Coalescing

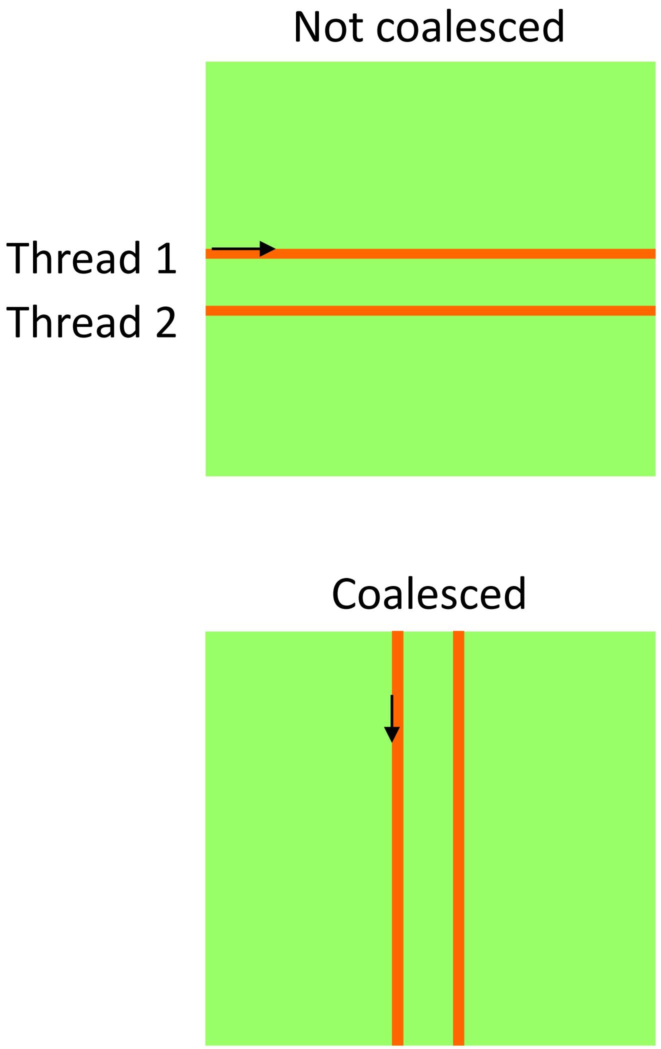
- Coalesced memory access
  - A warp of threads access adjacent data in a cache line
  - In the best case, this results in one memory transaction (best bandwidth)
- Uncoalesced memory access
  - A warp of threads access scattered data all in different cache lines
  - This may result in 32 different memory transactions (poor bandwidth)

Memory Coalescing



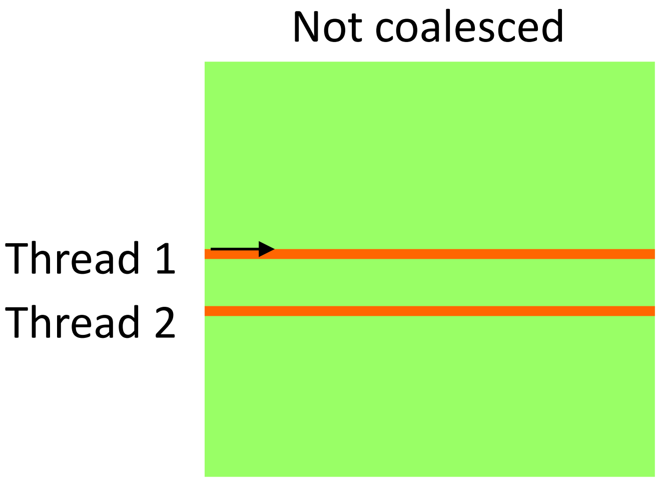
Global Memory Accesses

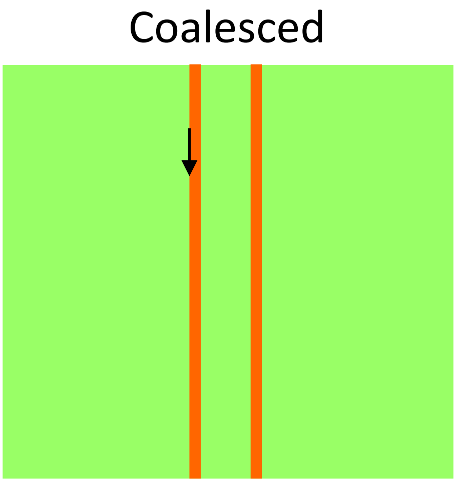
- Global memory (DRAM) is off-chip
  - Only one address per memory transaction
  - Each load transaction brings some number of aligned, contiguous bytes (say 32 B lines) from memory
  - Hardware automatically combines requests to same line from different threads in warp (coalescing)
  - Multiple lines are processed sequentially



Matrix Multiplication Example

```
__global__ void matmulKernel(float* A, float* B, float* C) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  float tmp = 0;
  if (row < N && col < N) {
    // Each thread computes one element of the matrix
    for (int k = 0; k < N; k++) {
      tmp += A[row * N + k] * B[k * N + col];
    }
  }
  C[row * N + col] = tmp;
}
```

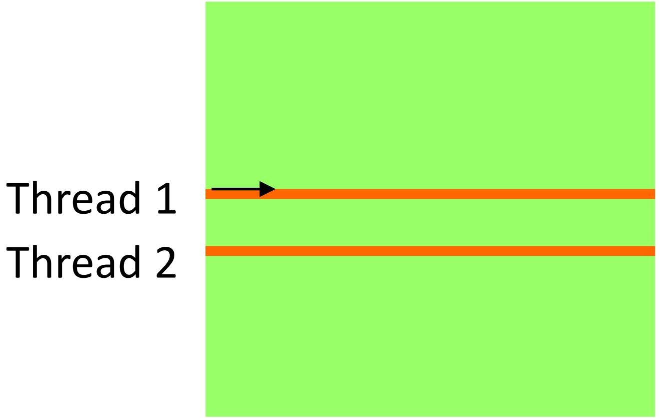




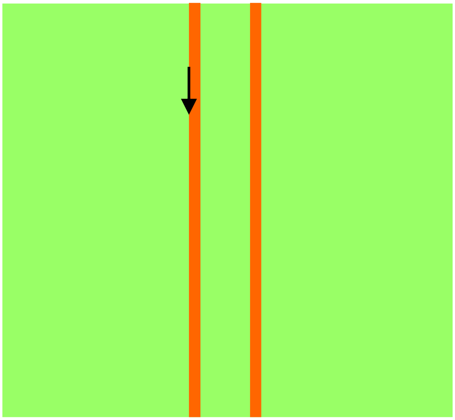
Optimizing Global Memory Accesses

- Try to ensure that memory requests from a warp can be coalesced
  - Stride-one access across threads in a warp is good
  - Use structure of arrays rather than array of structures

Not coalesced



Coalesced



References

- NVIDIA – CUDA C Programming Guide v10.1.
- NVIDIA – CUDA C Best Practices Guide v10.1.
- D. Kirk and W. Hwu – Programming Massively Parallel Processors.
- N. Matloff – Programming on Parallel Machines.
- Shane Cook - CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs.
- T. Aamodt et al. – General-Purpose Graphics Processor Architecture.
- J. Sanders and E. Kandrot – CUDA By Example: An Introduction to General-Purpose GPU Programming.