

# CUDA Programming

## Lecture 6

# Outline

- GPUs: brief history and present
- Nvidia GPU architecture model
- CUDA programming model
- CUDA software development
- Optimizing CUDA programs

# GPU – Graphics Processing Unit

- Originally used primarily for 3D game rendering
- Nowadays: accelerate computational workloads in areas such as financial modeling, scientific research, and oil and gas exploration
- Combination of CPU with GPU
  - System performance, price, and power
- TOP500 No. 1 & 2 Summit and Sierra use GPUs
  - Nvidia Volta GV100
- GPGPU – General purpose GPU

# A Bit of History for GPUs

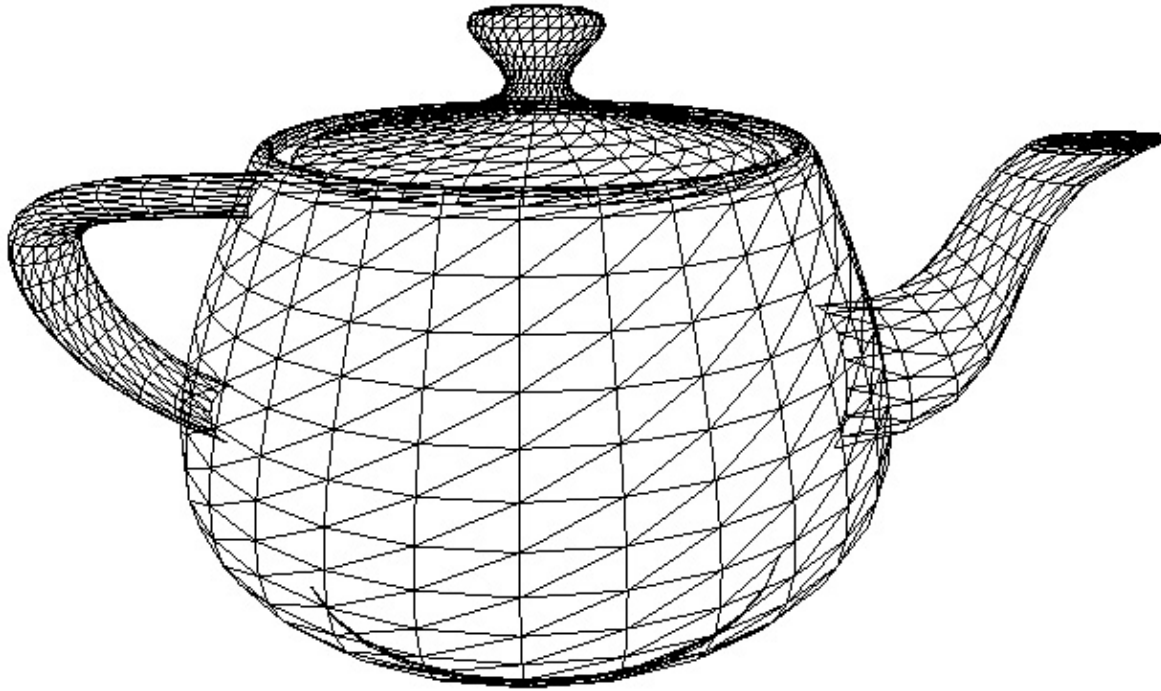
## ■ Shaders GPUs

- ❑ “Old school” approach to GPGPU programming
- ❑ Awkward programming interface - unnecessarily dependent on graphics pipeline

## ■ Drawbacks

- ❑ Hard to transfer data between GPU and CPU
- ❑ No scatter
- ❑ No communication between fragments (threads)
- ❑ Coarse thread synchronization

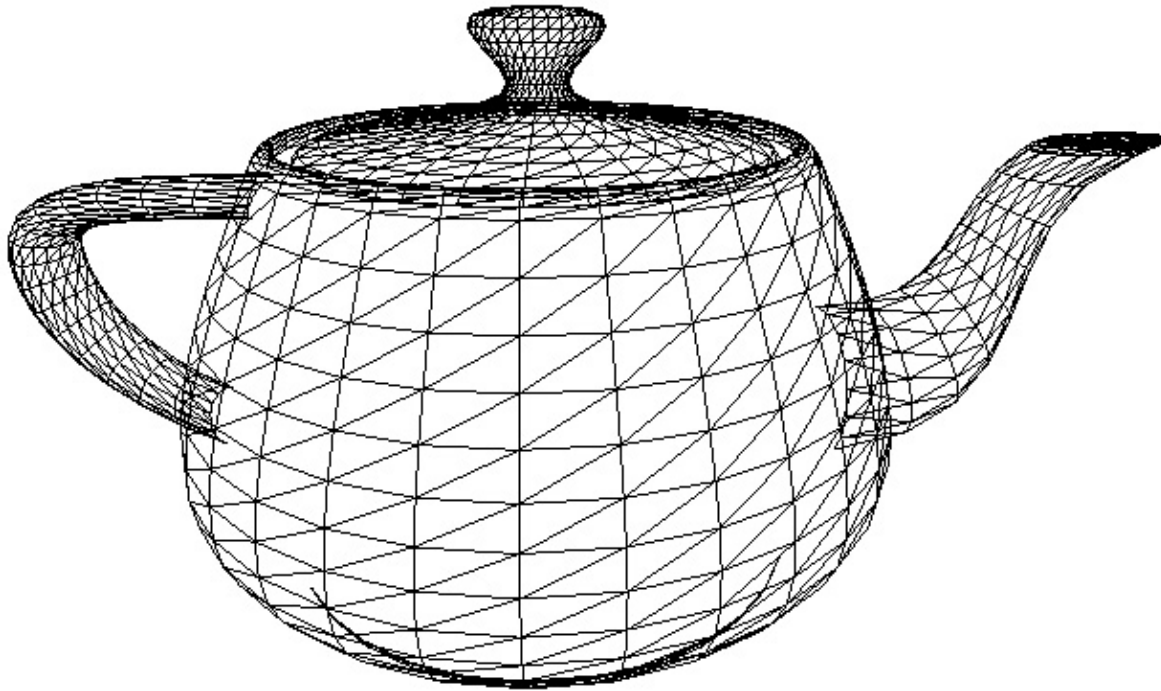
# Rendering the Teapot



■ Input: triangle mesh of the scene

■ Output: image of the scene

# Data Parallelism

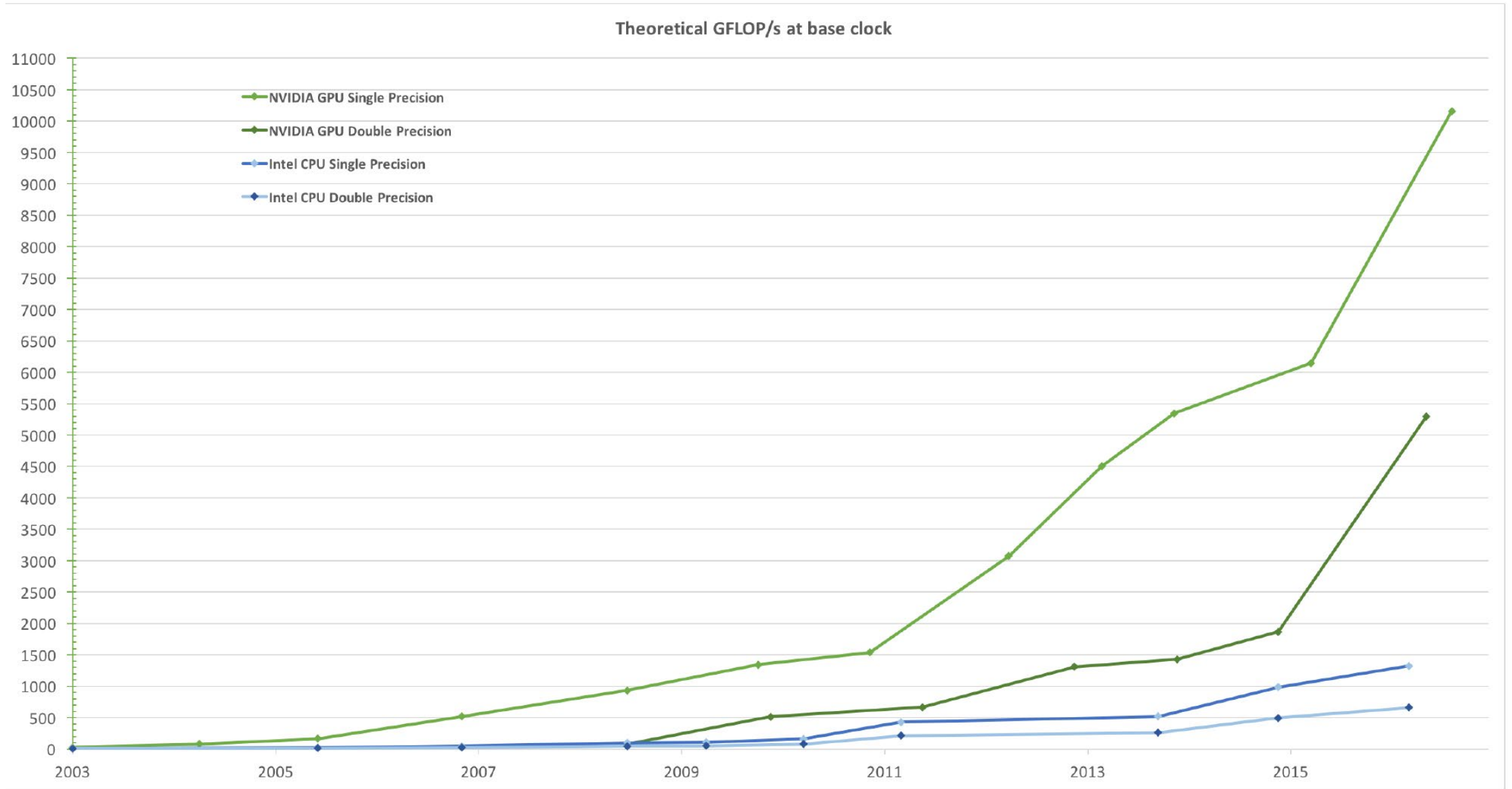


- Fast processors for performing the same computation (shader programs) on large collections of data (streams of vertices, fragments, and pixels)

# Current GPGPU Solutions

- General-purpose parallel computing architecture & programming on GPUs
  - NVIDIA CUDA
  - OpenCL (Apple)
  - DirectCompute (Windows)
  - etc.
- Programming independent of rendering pipeline

# Nvidia: Floating-Point Operations per Second for the CPU and GPU





# GPUs used in Industry

- Early adopters: Baidu, Google, Facebook, Microsoft
  - Respond to your spoken word
  - Translate speech or text to another language
  - Recognize and automatically tag images
  - Recommend newsfeeds, entertainment, and products
- Self-driving cars improved through deep learning
- Robots learn by trial and error
- Healthcare - understand how genetic variations can lead to disease

# GPUs in Artificial Intelligence

- Early progress in deep learning due to the advent of fast graphics processing units (GPUs) – train networks 10-20x faster
  - 2011: deep learning on GPUs to achieve performance of Google Brain running on 2000 CPUs
  - 2015: GPUs speedup parallel workloads for deep neural networks (DNN) for speech and image recognition

# GPU Computing Applications

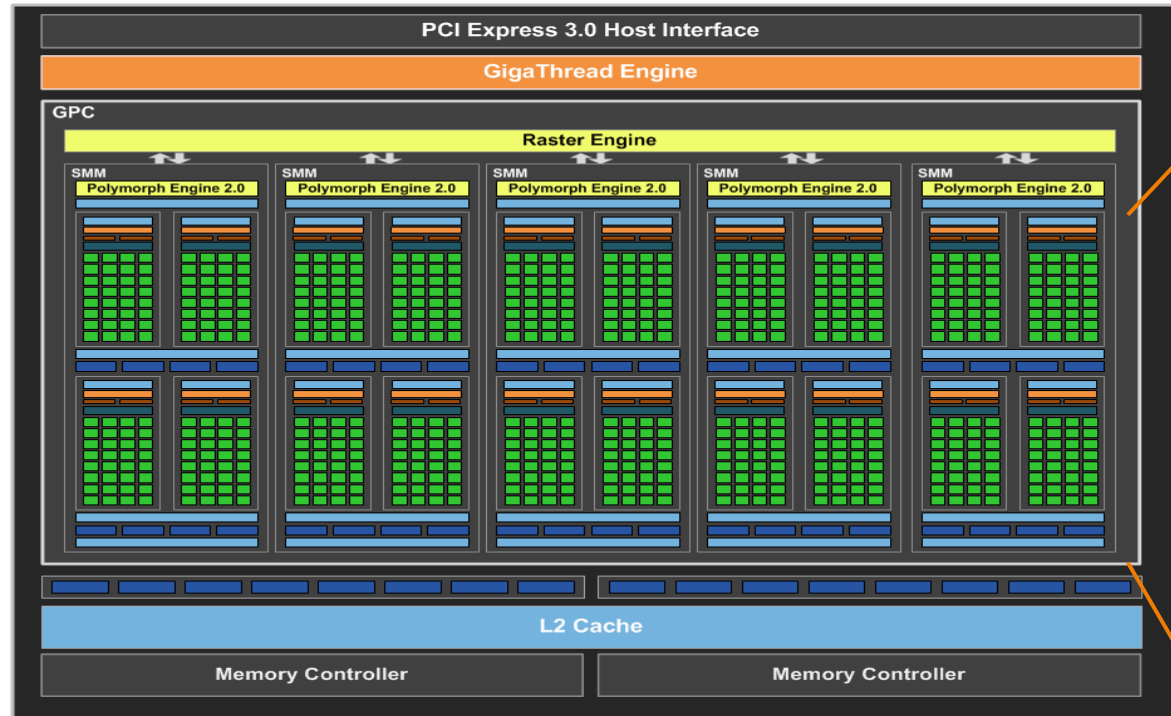
GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)	
CUDA-enabled NVIDIA GPUs						
Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series		
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series		
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series		
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series		
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series		
	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER		

# GPU Architecture

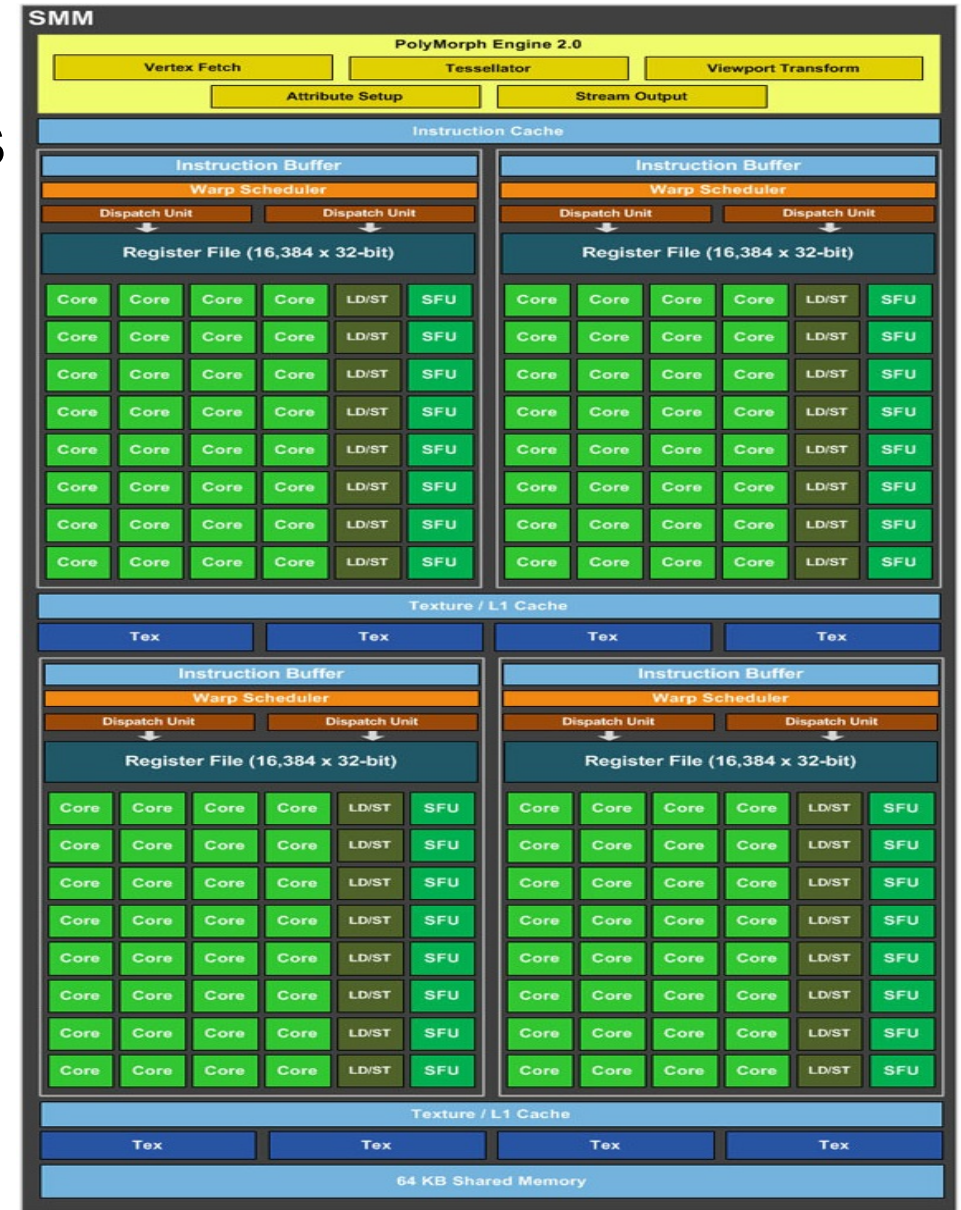
- Multiple **Streaming Multiprocessors** (SMs)
  - Memory and cache
  - Connecting interface (usually PCI Express)
- SM consists of multiple compute cores
  - Memories (registers, L1 cache, texture memory, and shared memory)
  - Logic for thread and instruction management

# Architecture

## GPU



## SMMs



# Fermi Streaming Multiprocessor (SM)



# Compute Capability

- Architecture and capabilities differ
  - 1.x to 7.x
- Architecture name
  - Tesla (1.x), Fermi (2.x), Kepler (3.x), Maxwell (5.x), Pascal (6.x), **Volta** (7.0, 7.2), **Turing** (7.5), Ampere (8.0, 8.6)



# CUDA Programming Model



# CUDA Programming Model

- “Compute Unified Device Architecture”
- Massively hardware multithreaded
  - GPU = dedicated many-core co-processor
- General purpose programming model
  - Simple extension to standard C
  - Mature software stack (high-level and low-level access)
  - User launches batches of threads on the GPU
  - Fully general load/store memory model (CRCW)
- Not another graphics API
  - Though graphics API interoperability possible

# CUDA

- Works on NVIDIA GPUs
- Designed to scale well over time
- Comes with
  - Hardware architecture
  - Programming model
  - Programming interface
    - CUDA C (extension to the C language)
    - CUDA Driver API
  - Software development environment
    - Tools (compiler, debugger, profiler), C runtime, libraries

# CUDA Programming Model

- Transparently scales to hundreds of cores and thousands of parallel threads
- Programmers focus on parallel algorithms
- Enable heterogeneous systems (i.e. CPU + GPU)

# CUDA Kernels and Threads

## ■ Definitions

- ❑ **Device** = GPU
- ❑ **Host** = CPU
- ❑ **Kernel** = function that runs on the device

## ■ Parallel portions execute on device as kernels

- ❑ One kernel is executed at a time
- ❑ Multiple kernels are allowed in newer CUDA hardware

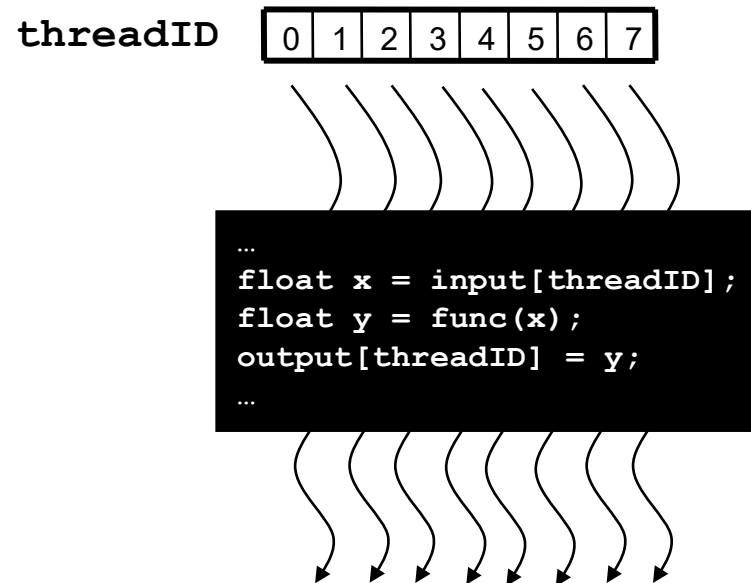
## ■ CUDA threads are extremely lightweight

- ❑ Very little creation overhead
- ❑ Instant switching

## ■ CUDA uses thousands of threads to achieve efficiency

# Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
    - SPMD model - Single Program, Multiple Data
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

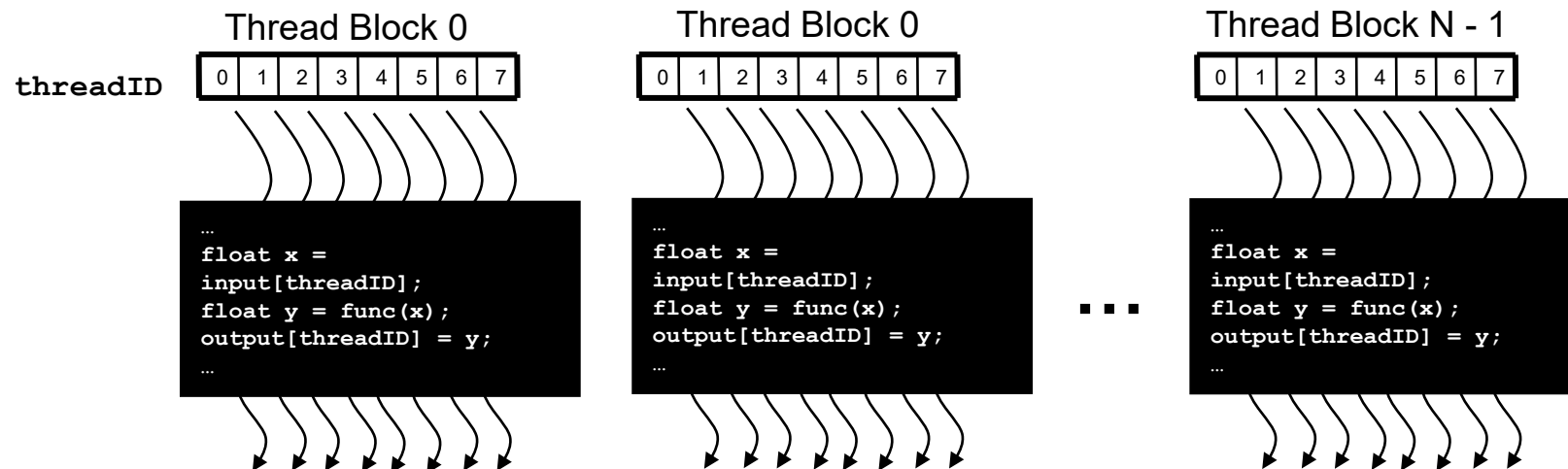


# Thread Cooperation

- Threads in the array need not be completely independent
- Powerful feature of CUDA
- Valuable
  - ❑ Share results to save computation
  - ❑ Share memory accesses
    - Drastic bandwidth reduction

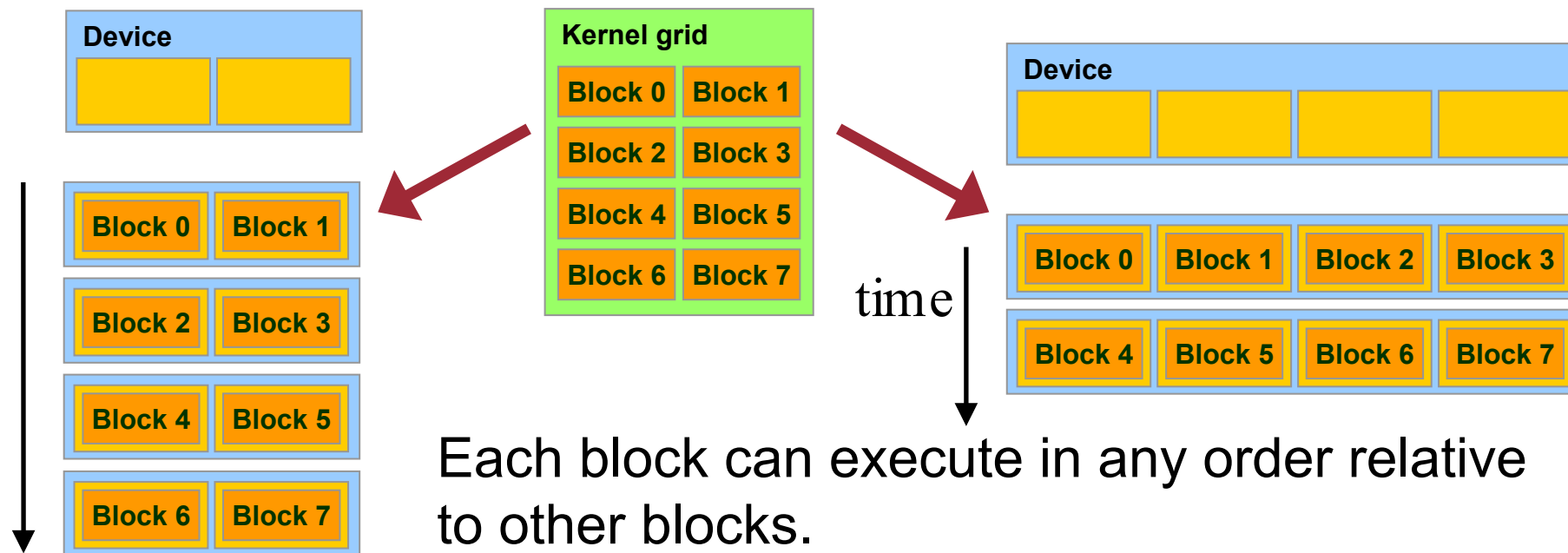
# Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple **blocks**
  - In a block: **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate
- Enables programs to transparently scale to any number of processors



# Transparent Scalability

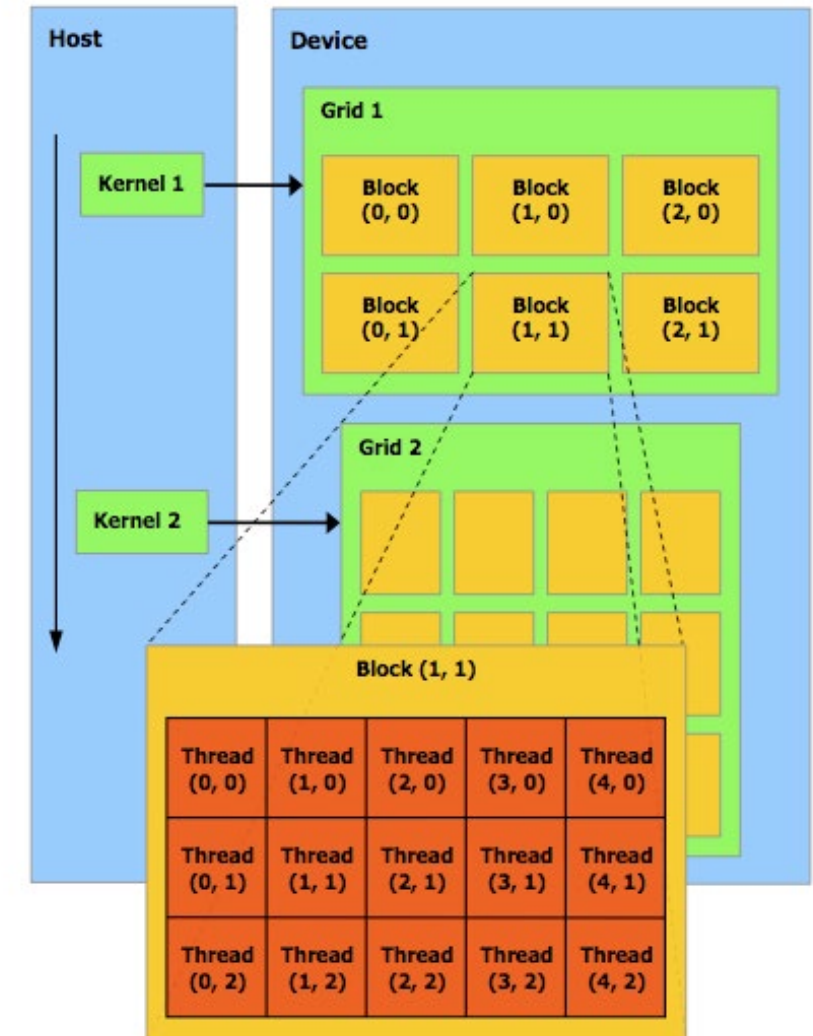
- Hardware is free to schedule thread blocks to any processor at any time
  - A kernel scales across any number of parallel multiprocessors





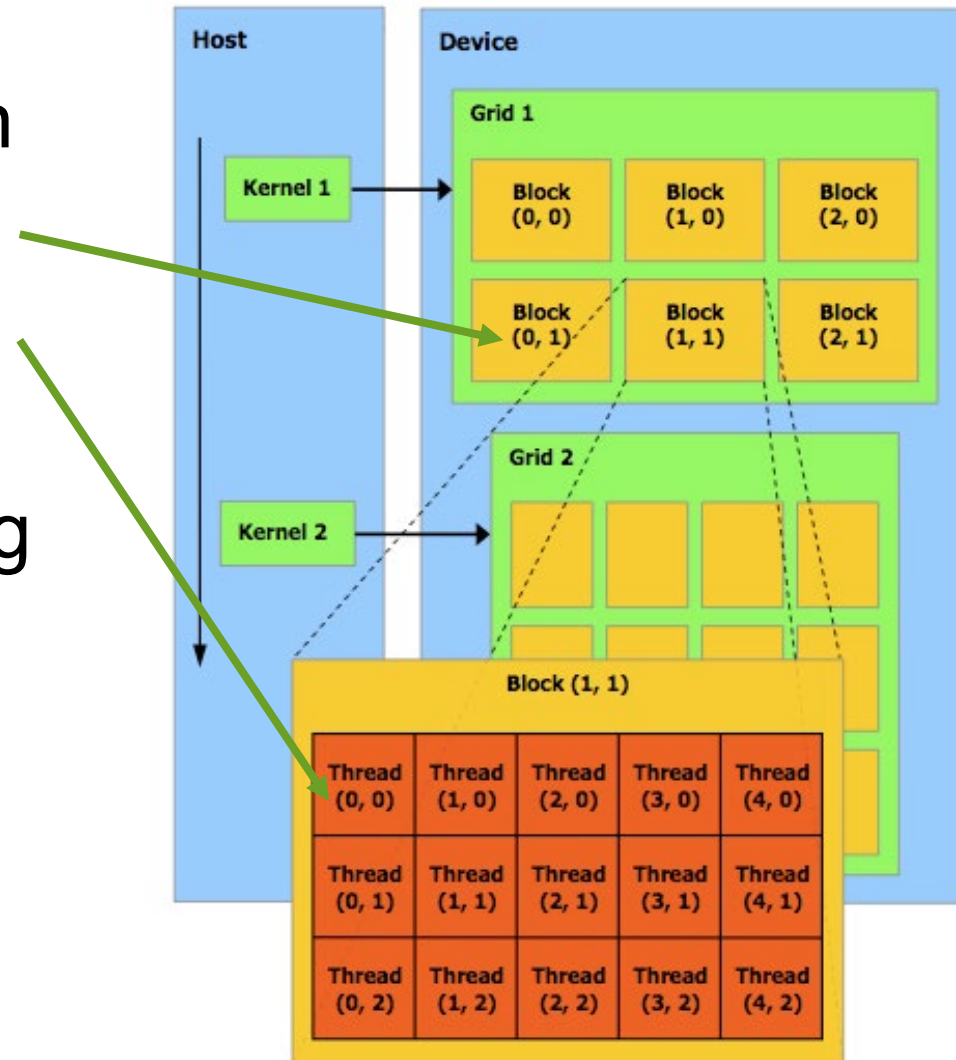
# CUDA Thread Hierarchy

- A **kernel** is executed by a grid of thread blocks
- **Blocks**
  - ❑ Share data through shared memory
  - ❑ Synchronize their execution
- Threads from different blocks cannot cooperate



# Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - ❑ Block ID: 1D, 2D, or 3D
  - ❑ Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - ❑ Image processing
  - ❑ Solving PDEs on volumes
  - ❑ ...



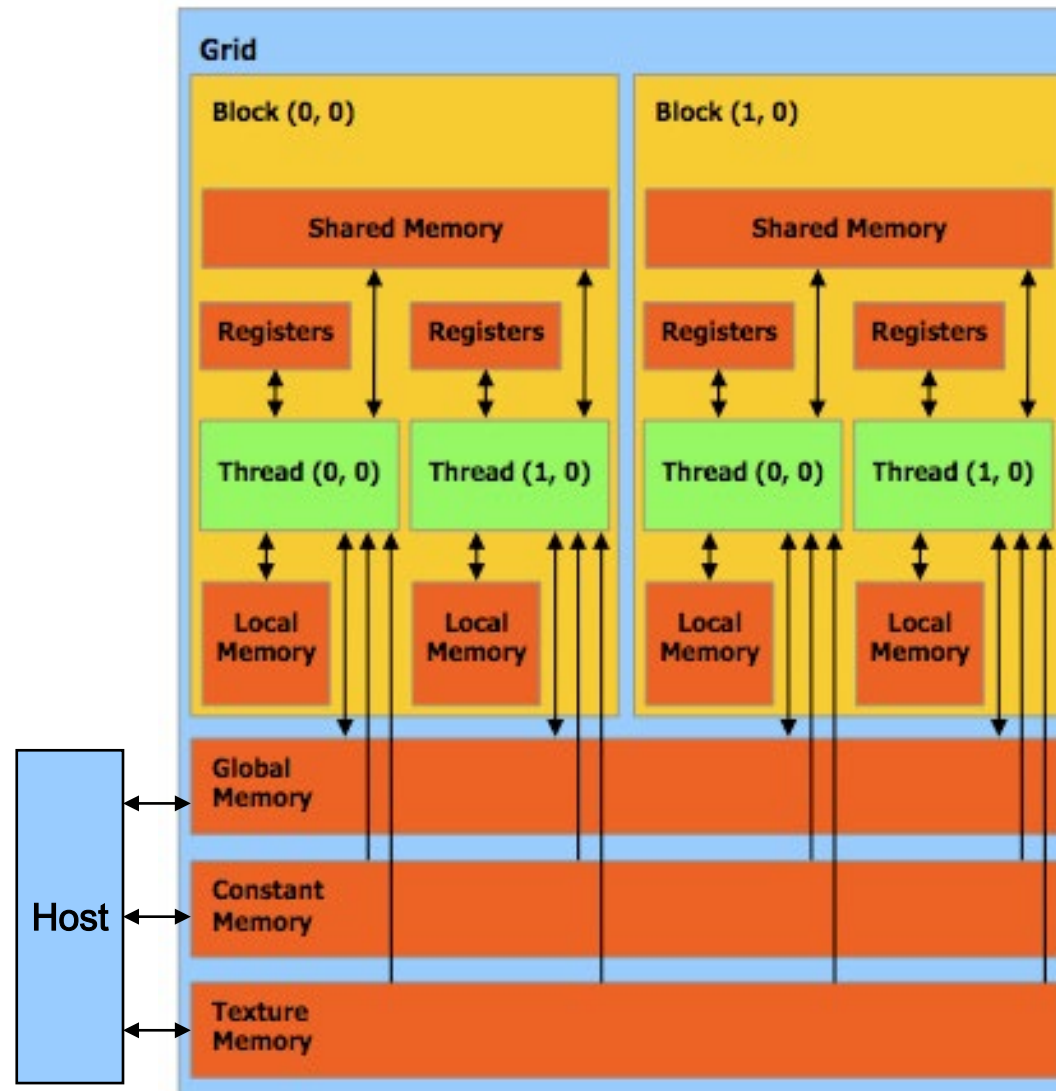
# Execution Model Mapping to Architecture

- Kernels are launched in grids
  - One or multiple kernels execute at a time
- A block executes on one **streaming multiprocessor (SM)**
  - Does not migrate
  - GPUs usually have multiple SMs
- Several blocks can reside concurrently on one SM
  - Control limitations (depending on capability)
  - Number is further limited by SM resources
    - Register file is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks

# Thread Execution Mapping to Architecture

- **SIMT** (single-instruction, multiple-thread) execution model
- Multiprocessor creates, manages, schedules and execute threads in SIMT **warps** (group of 32 parallel threads)
- Threads in a warp start together at the same program address
  - Threads have individual instruction program counter and register state
- A block is always split into warps the same way
  - Each warp contains threads of consecutive, increasing thread IDs
- Warp executes one common instruction at a time

# CUDA Memory Model



# CUDA Memory Spaces

- Global Memory and Shared Memory
  - Most important, commonly used
- Local, Constant, and Texture Memory for convenience / performance
  - **Local**: automatic array variables allocated there by compiler
  - **Constant**: useful for uniformly-accessed read-only data
    - Cached
  - **Texture**: useful for spatially coherent random-access read-only data
    - Cached
    - Provides filtering, address clamping and wrapping

# Shared Memory

- Higher bandwidth and lower latency than local or global memory
- Divided into equally-sized memory modules, called **banks**
  - Addresses from different banks can be accessed simultaneously
- Bank conflict: two addresses of a memory request fall in the same memory bank
  - Memory access has to be serialized

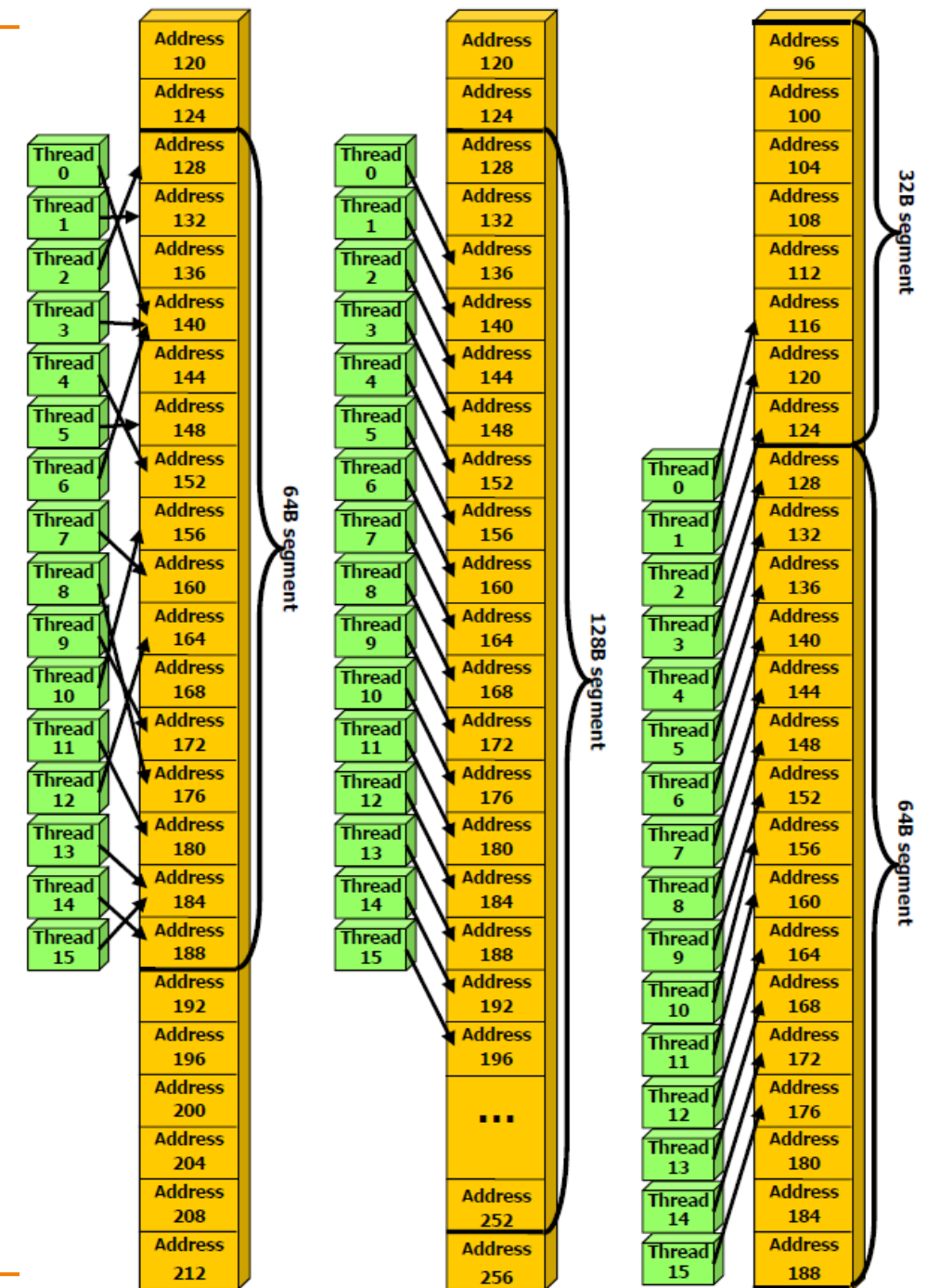
# Coalesced Access to Global Memory

- Simultaneous accesses to global memory by threads in a **half-warp** (16 threads) can be **coalesced** into as few as a single memory transaction of 32, 64 or 128 bytes
- Requirements for coalescing are based on compute capability



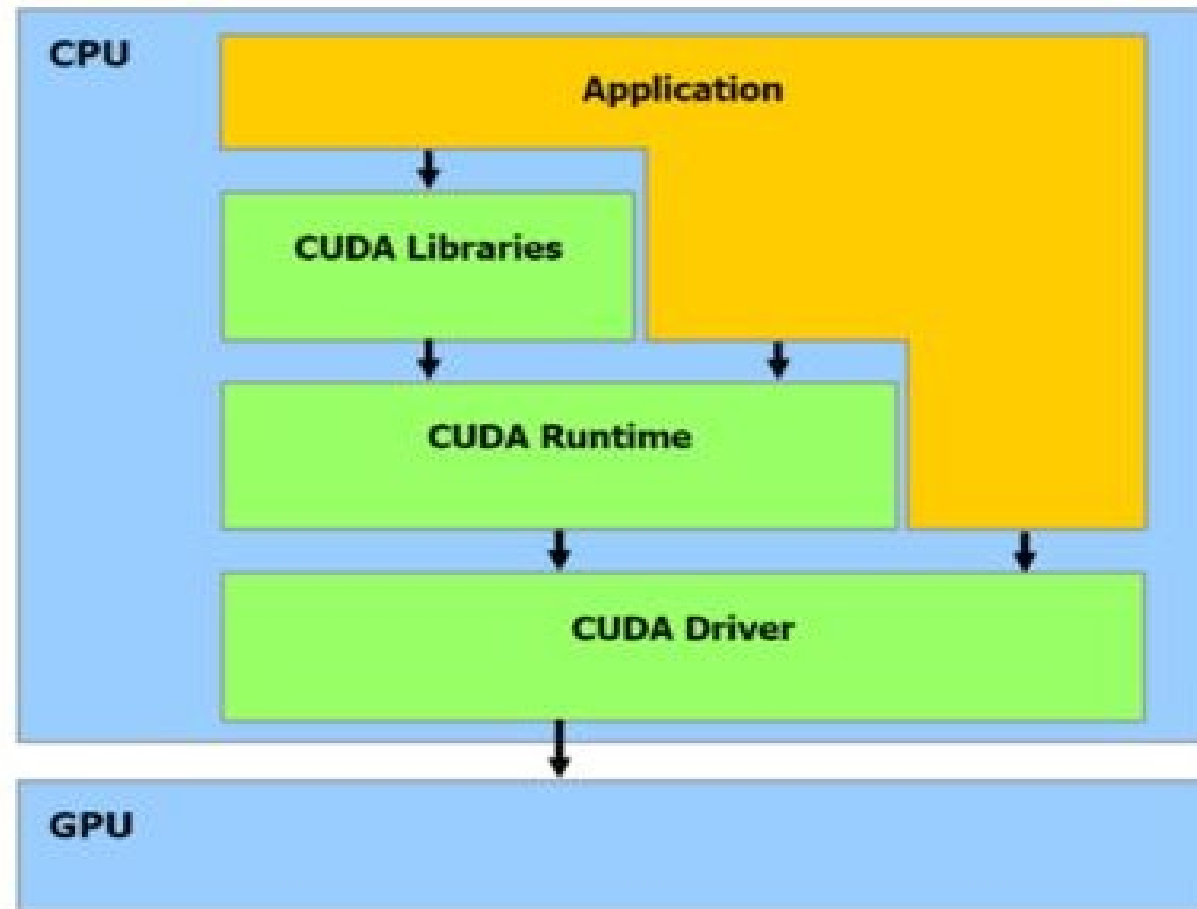
# Examples of Global Memory Access

- For Compute Capability 1.2 and higher
- Left: random float memory access within a 64B segment, resulting in one memory transaction
- Center: misaligned float memory access, resulting in one transaction
- Right: misaligned float memory access, resulting in two transactions



# CUDA Software Development

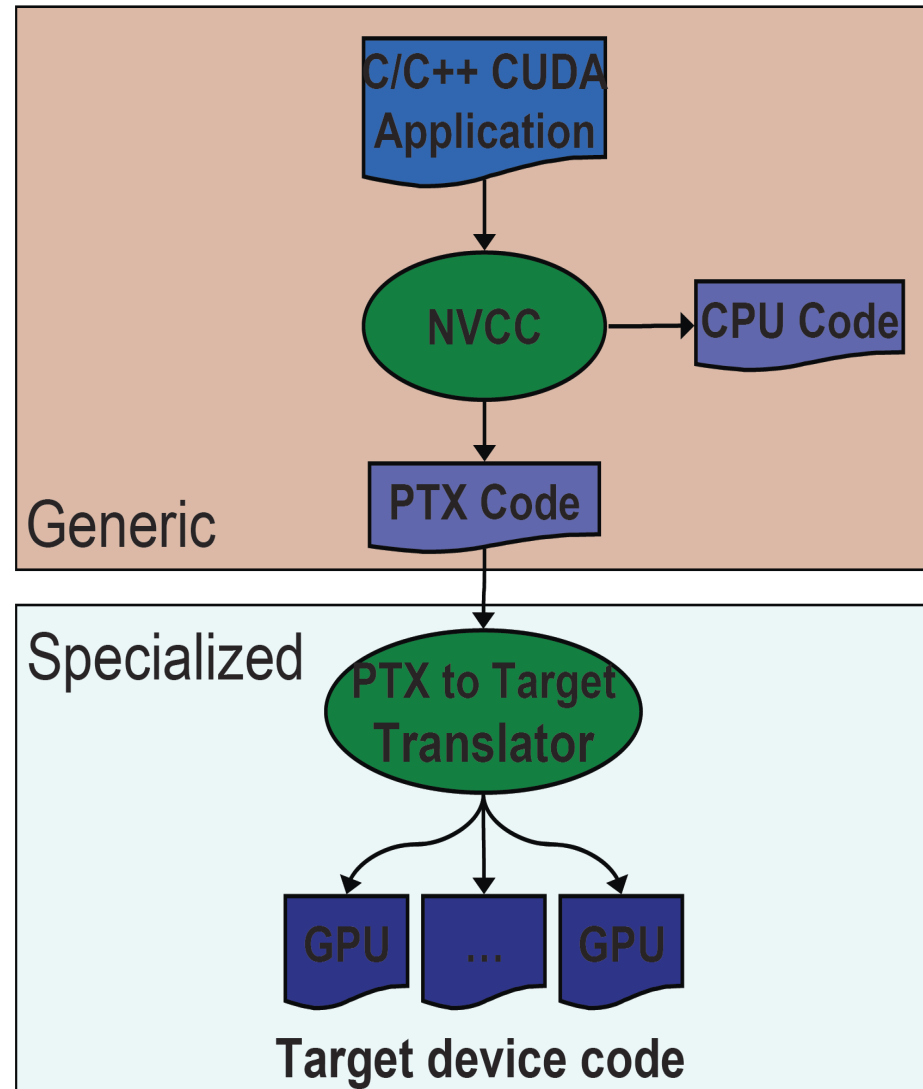
# CUDA Layers



# CUDA Programming Interfaces

- CUDA C Runtime
  - ❑ Minimal set of extensions to the C language
  - ❑ Kernels defined as C functions embedded in application source code
  - ❑ Requires a runtime API (built on top of CUDA driver API)
- CUDA driver API
  - ❑ Low-level C API to load compiled kernels, inspect their parameters, and to launch them
  - ❑ Kernels are written in C and compiled to CUDA binary or assembly code
  - ❑ Requires more code, harder to program and debug
  - ❑ Much like using the OpenGL API on GLSL shaders

# Compiling a CUDA Program



# Compilation and Linking

- NVCC is a compiler driver for CUDA source
  - Works by invoking other tools and compilers like cudacc, g++, cl, ...
- NVCC outputs
  - C code (host CPU code)
    - Must then be using another tool
  - PTX
    - Object code or PTX source interpreted at runtime
- Linking with two static/dynamic libraries
  - CUDA runtime library (cudart)
  - CUDA core library (cuda)

# Programming in CUDA

# Device Code

- C functions with some restrictions
  - ❑ Can only access GPU memory
  - ❑ No variable number of arguments (“varargs”)
  - ❑ No static variable
  - ❑ No recursion (\*)



# Function Types

## ■ Function qualifiers

- ❑ `__host__`

- ❑ `__device__`

- ❑ `__global__`

- Function is a kernel

- Must have void return type

- A call to function must specify execution configuration

- Function parameters are passed via shared memory (\*)

- ❑ `__host__` and `__device__` can be used together

# Launching Kernels

- Modified C function call syntax
  - `kernel<<<dim3 grid, dim3 block, int smem, int stream>>>(...)`
- Execution Configuration (“<<< >>>”)
  - grid dimensions: x and y
  - thread-block dimensions: x, y, and z
  - shared memory: number of bytes per block for extern smem variables declared without size
    - optional, 0 by default
  - stream ID
    - optional, 0 by default

# CUDA Built-In Device Variables

- All `__global__` and `__device__` functions have access to these automatically defined variables
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (`gridDim.z` unused)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

# Variable Qualifiers (Device Code)

## ■ `__device__`

- ❑ Stored in global memory (large, high latency, no cache)
- ❑ Read/write by all threads within grid
- ❑ Written by CPU via `cudaMemcpyToSymbol()`
- ❑ Lifetime: application

## ■ `__constant__`

- ❑ Same as `__device__`, but cached and read-only by all threads within grid
- ❑ Written by CPU via `cudaMemcpyToSymbol()`
- ❑ Lifetime: application

# Variable Qualifiers (Device Code)

- `__shared__`
  - ❑ Stored in on-chip shared memory (very low latency)
  - ❑ Read/write by all threads in the same thread block
  - ❑ Lifetime: block
- Unqualified variables (in device code)
  - ❑ Scalars and built-in vector types are stored in registers
  - ❑ Arrays of more than 4 elements or run-time indices stored in local memory
  - ❑ Read/write by thread only
  - ❑ Lifetime: thread

# GPU Memory Allocation / Release / Copy

## ■ GPU memory allocation / release

- ❑ `cudaMalloc(void **pointer, size_t nbytes);`
- ❑ `cudaMemset(void *pointer, int value, size_t count);`
- ❑ `cudaFree(void* pointer);`

## ■ Data copy

- ❑ `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - `enum cudaMemcpyKind`
    - ❑ `cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice`
- ❑ **Unified memory model** does not need data transfer
  - `__managed__`

# Thread Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
  - ❑ Generates barrier synchronization instruction
  - ❑ Used to avoid RAW / WAR / WAW hazards when accessing shared memory

# Example Code Walkthrough



# Code Example to Add Two Arrays

## ■ CUDA C Program

A CUDA kernel

```
__global__
void addMatrixG( float *a, float *b, float *c, int N )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j * N;
    if ( i < N && j < N )
        c[index] = a[index] + b[index];
}
```

Device code

```
void main()
{
    .....
    dim3 dimBlk( 16, 16 );
    dim3 dimGrd( N/dimBlk.x, N/dimBlk.y );
    addMatrixG<<<dimGrd, dimBlk>>>( a, b, c, N );
}
```

Host code

# CUDA Example 1 – Add Two Arrays (1)

// Device code

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

// Host code

```
int main()  
{
```

```
    // Allocate vectors in device memory  
    size_t size = N * sizeof(float);  
    float *d_A;  
    cudaMalloc( (void**)&d_A, size );  
    float *d_B;  
    cudaMalloc( (void**)&d_B, size );  
    float *d_C;  
    cudaMalloc( (void**)&d_C, size );
```

# CUDA Example 1 – Add Two Arrays (2)

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

```
// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) /
                    threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);
```

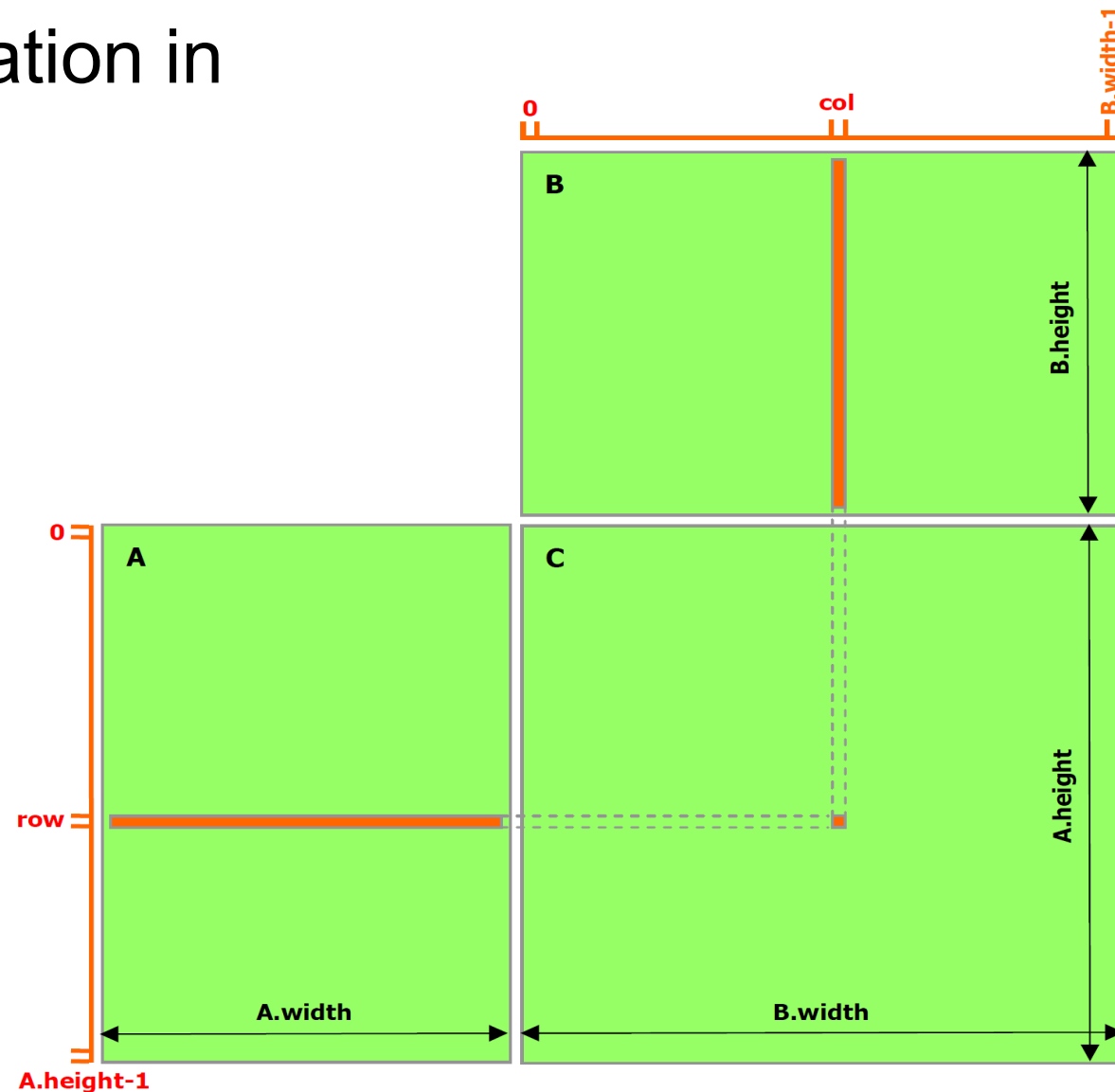
```
// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy( h_C, d_C, size, cudaMemcpyDeviceToHost );
```

```
// Free device memory
cudaFree( d_A ); cudaFree( d_B ); cudaFree( d_C );
```

```
}
```

# CUDA Example 2 – Matrix Multiplication (1)

- Matrix multiplication in global memory



## CUDA Example 2 – Matrix Multiplication (2)

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
```

## CUDA Example 2 – Matrix Multiplication (3)

```
Matrix d_B;  
d_B.width = B.width; d_B.height = B.height;  
size = B.width * B.height * sizeof(float);  
cudaMalloc((void**)&d_B.elements, size);  
cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);
```

```
// Allocate C in device memory  
Matrix d_C;  
d_C.width = C.width; d_C.height = C.height;  
size = C.width * C.height * sizeof(float);  
cudaMalloc((void**)&d_C.elements, size);
```

```
// Invoke kernel  
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);  
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);  
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);  
  
// Read C from device memory  
cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);  
  
// Free device memory  
cudaFree(d_A.elements); cudaFree(d_B.elements); cudaFree(d_C.elements);  
}
```

## CUDA Example 2 – Matrix Multiplication (4)

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;

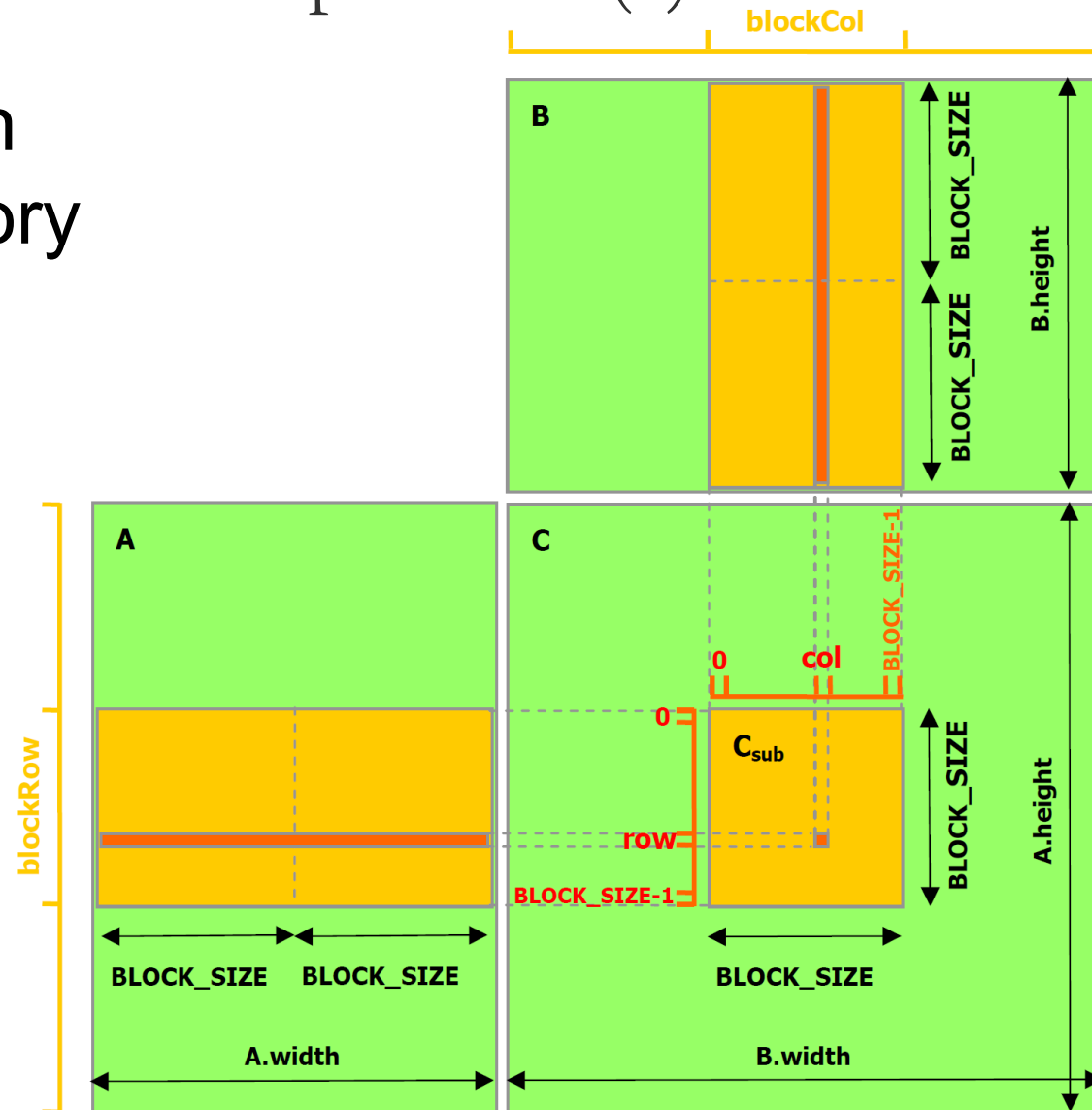
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                  * B.elements[e * B.width + col];

    C.elements[row * C.width + col] = Cvalue;
}
```

## CUDA Example 3 – Matrix Multiplication (1)

- Matrix multiplication using shared memory





# Optimizing CUDA Programs

# Overall Optimization Strategies

## (1) Maximizing **parallel execution**

- ❑ Restructure algorithm to expose as much **data parallelism** as possible
- ❑ Map to hardware by carefully choosing the execution configuration of each kernel invocation

## (2) Optimizing memory usage to achieve maximum **memory bandwidth**

- ❑ Different memory spaces and access patterns have vastly different performance

## (3) Optimizing instruction usage to achieve maximum **instruction throughput**

- ❑ Use high throughput arithmetic instructions
- ❑ Avoid different execution paths within same warp

# Execution Configuration

- Number of thread blocks should be larger than number of multiprocessor
  - So that all multiprocessors have at least one block to execute
- Multiple blocks per multiprocessor - no idling when a block synchronizes
- Threads per block should be multiple of warp size
- Threads per block should be multiple of 64 to avoid memory bank conflicts
- Block size limited by amount of register and shared memory used

# Memory Optimizations

- (1) Minimize data transfer between host and device
- (2) Ensure global memory accesses are coalesced whenever possible
- (3) Minimize global memory accesses by using shared memory
- (4) Minimize bank conflicts in shared memory accesses

# Data Transfer Between Host and Device

- Peak bandwidth between device memory and GPU is much higher than that between host memory and device memory
- Should minimize data transfer between host and device memory
  - Even if that means running kernels on GPU that do not have any speed-up over CPU
- Use page-locked or pinned memory transfer
- Overlapping asynchronous transfers with GPU computations

# Maximize Instruction Throughput

- Minimize the use of arithmetic instructions with low throughput
  - Trading precision for speed
- Minimize divergent warps caused by control flow instructions
- Reduce the number of instructions
  - Optimizing out synchronization points
  - Using restricted pointers `__restrict__`

# References

- Accelerating AI with GPUs: A New Computing Model
  - <https://blogs.nvidia.com/blog/2016/01/12/accelerating-ai-artificial-intelligence-gpus/>
- NVIDIA CUDA C Programming Guide, Version 11.0
  - [http://docs.nvidia.com/cuda/pdf/CUDA\\_C\\_Programming\\_Guide.pdf](http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf)
- CUDA C Best practices guide
  - [http://docs.nvidia.com/cuda/pdf/CUDA\\_C\\_Best\\_Practices\\_Guide.pdf](http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf)