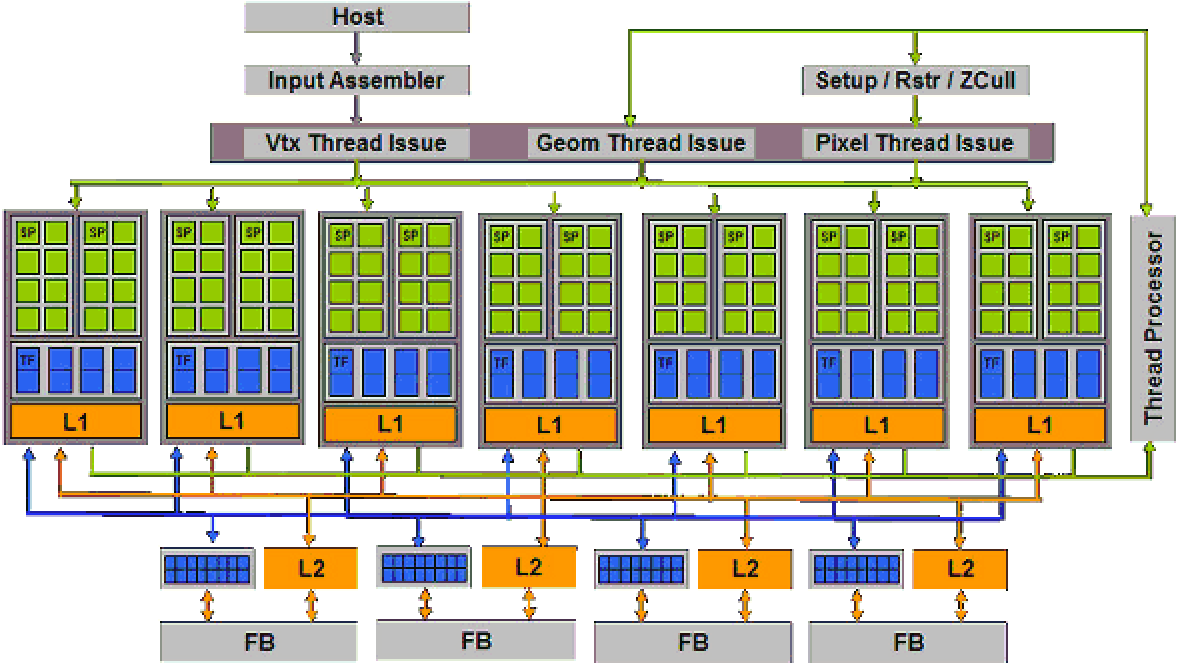
# CUDA Programming

Week 1. Basic Programming Concepts

Materials are copied from the reference list

# G80/G92 Device

- SP: Streaming Processor (Thread Processors)
- SM: Streaming Multiprocessor
  - 128 SP grouped into 16 SMs
- TPC: Texture Processing Clusters



# CUDA Programming Model

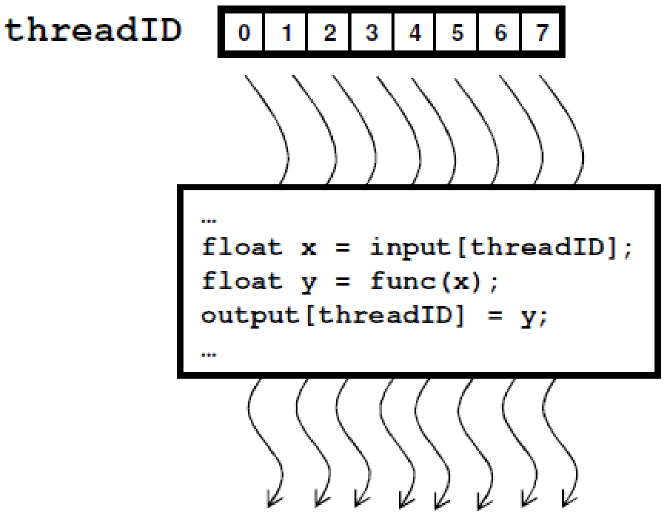
- The GPU is a compute device
  - serves as a coprocessor for the host CPU
  - has its own device memory on the card
  - executes many threads in parallel
- Parallel kernels run a single program in many threads
  - GPU expects 1000's of threads for full utilization

# CUDA Programming Kernels

- Device = GPU
- Host = CPU
- Kernel = function called from the host that runs on the device
  - One kernel is executed at a time
  - Many threads execute each kernel

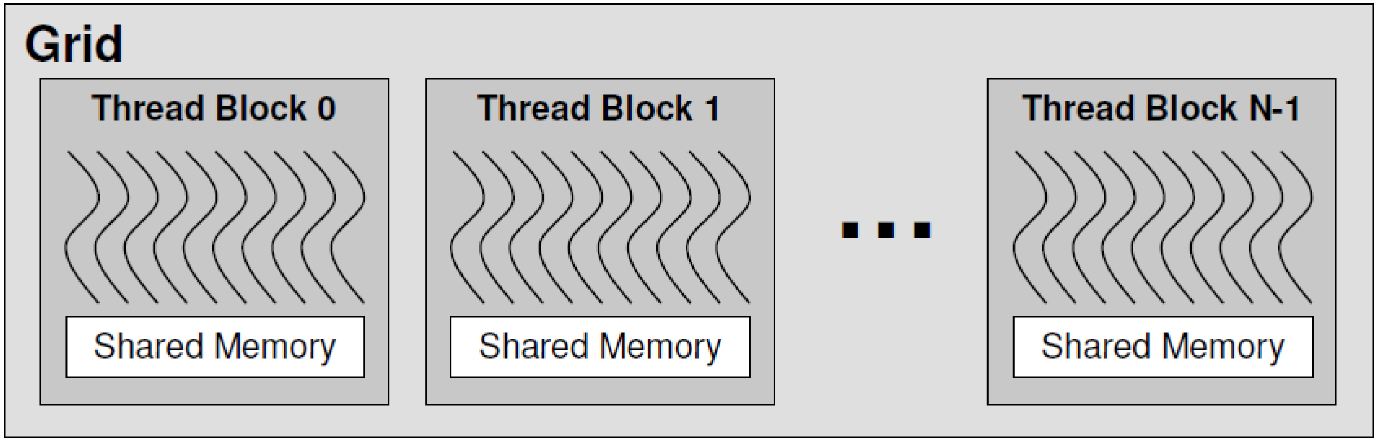
# CUDA Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID
    - Compute memory addresses
    - Make control decisions
- CUDA threads are extremely lightweight
  - Very little *creation overhead*
  - *Instant switching*



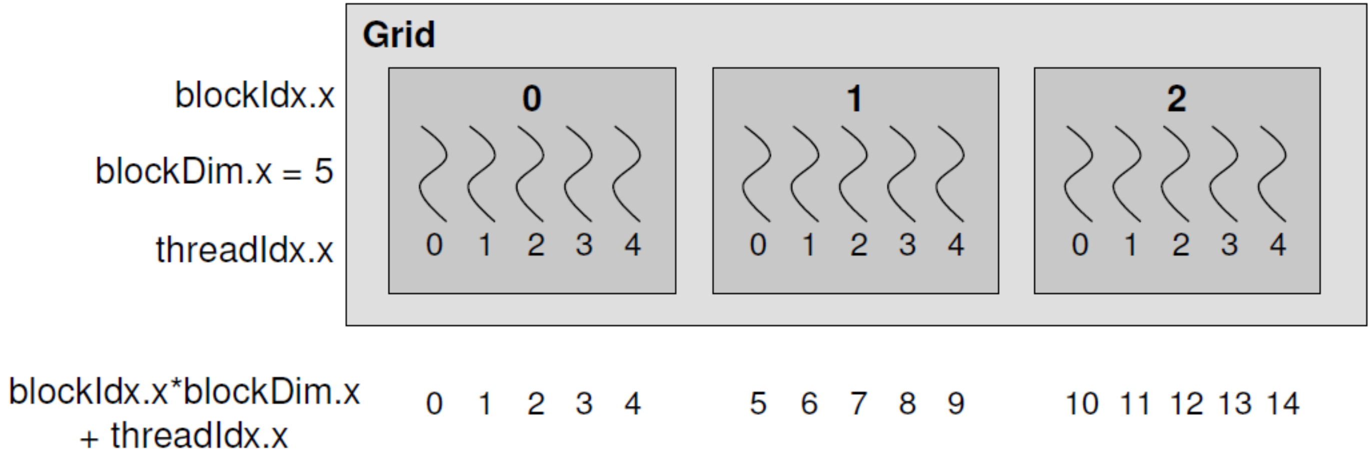
# Thread Batching

- Kernel launches a grid of thread blocks
  - Threads within a block can
    - Share data through shared memory
    - Synchronize their execution
  - Threads in different block cannot cooperate



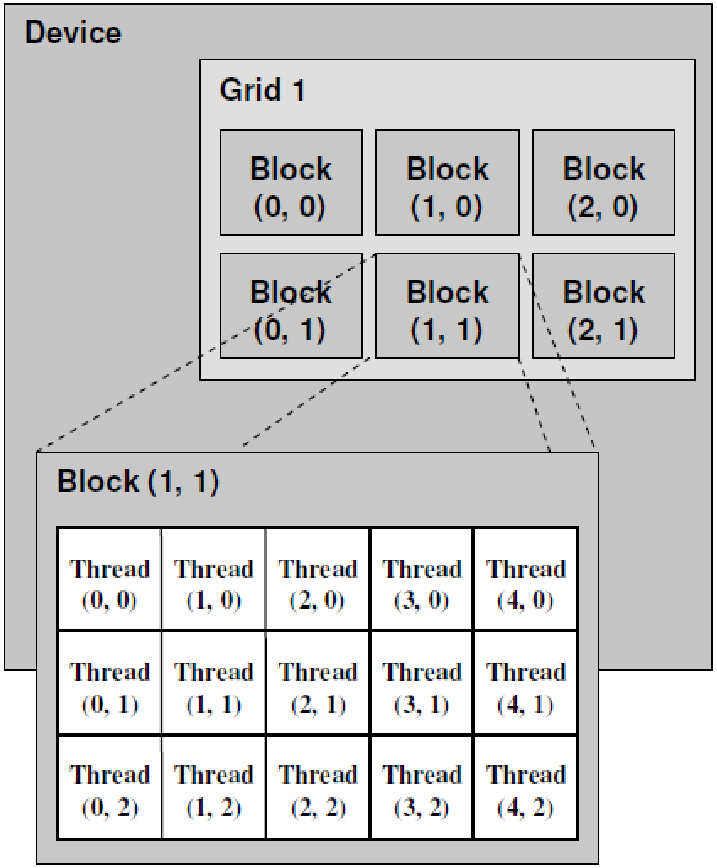
# Thread ID

- Each thread has access to:
  - threadIdx.x - thread ID within block
  - blockIdx.x - block ID within grid
  - blockDim.x - number of threads per block



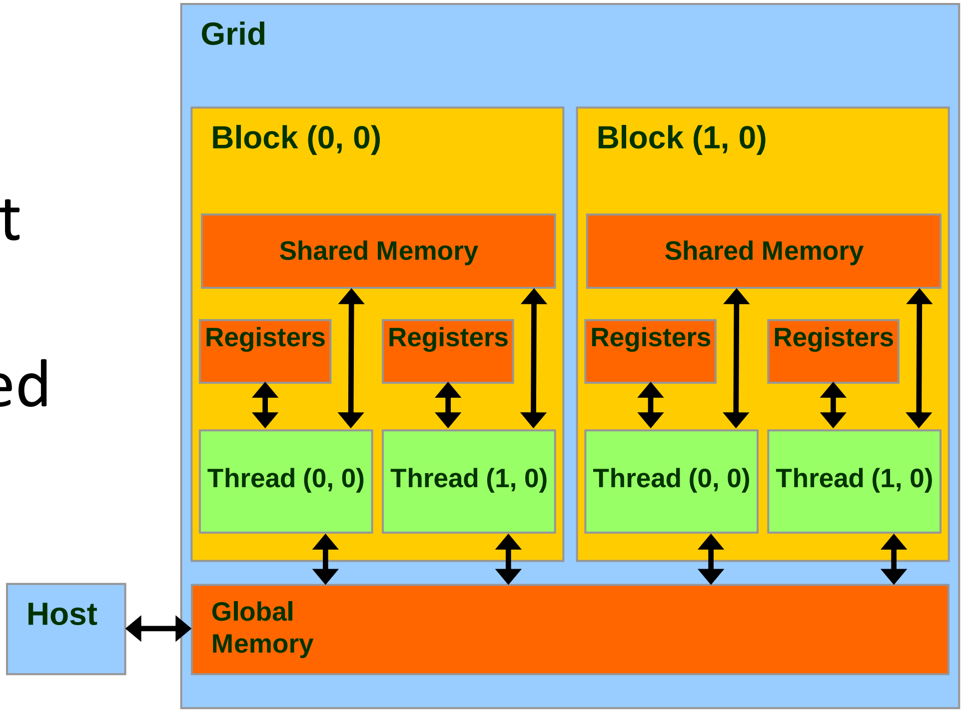
# Multidimensional IDs

- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing for processing multidimensional data
  - We will talk about it later

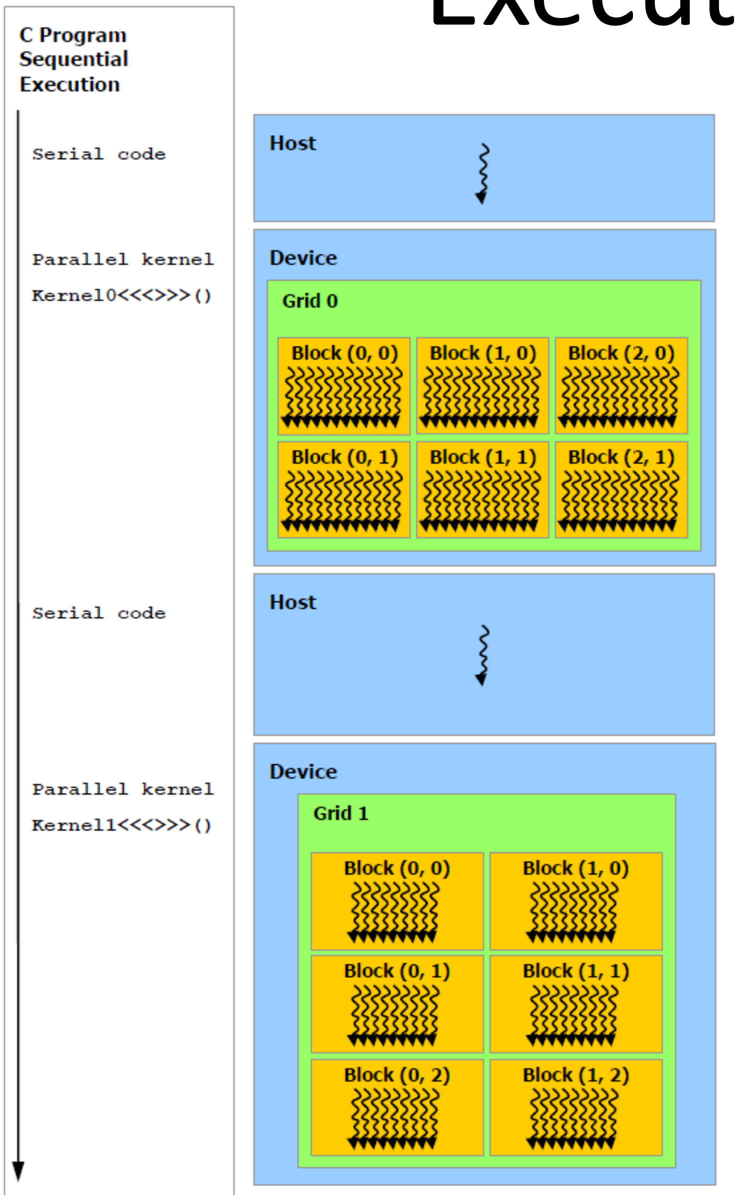


# Kernel Memory Access

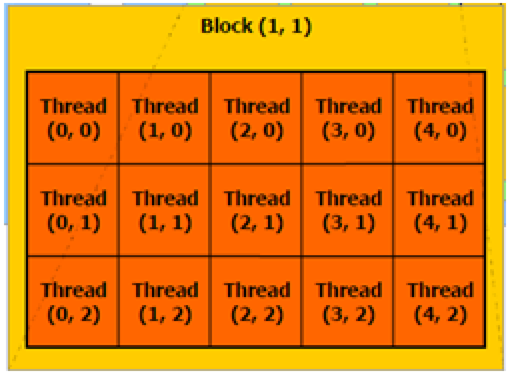
- Registers
- Global Memory
  - Kernel input and output data reside here
  - Off-chip, large, uncached
- Shared Memory
  - Shared among threads in a single block
  - On-chip, small, as fast as registers
- The host can read & write global memory but not shared memory



# Execution Model




- Kernels are launched in grids
  - One kernel executes at a time
- A block executes on one multiprocessor
  - Does not migrate



# Programming Basics

# Outline

- New stuffs
- Executing codes on GPU
- Memory management
  - Shared memory
- Schedule and synchronization

# NEW STUFFS

# C Extension

- **New syntax and built-in variables**
- **New restrictions**
  - No recursion in device code
  - No function pointers in device code
- **API/Libraries**
  - CUDA Runtime (Host and Device)
  - Device Memory Handling (cudaMalloc,...)
  - Built-in Math Functions (sin, sqrt, mod, ...)
  - Atomic operations (for concurrency)
  - Data types (2D textures, dim2, dim3, ...)

# New Syntax

- <<< … >>>
- __host__, __global__, __device__
- __constant__, __shared__, __device__
- __syncthreads()

# Built-in 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

dim3 (Based on uint3)
struct dim3{int x,y,z;}
Used to specify dimensions
Default value (1,1,1)

# Function Qualifiers

- __global__ : called from the host (CPU) code, and run on GPU
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions, and run on GPU
  - cannot be called from host (CPU) code
- __host__ : called from host , and run on CPU,
- __host__ and __device__:
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

# Variable Qualifiers (GPU code)

- __device__: stored in global memory (not cached, high latency)
  - accessible by all threads
  - lifetime: application
- __constant__: stored in global memory (cached)
  - read-only for threads, written by host
  - Lifetime: application
- __shared__: stored in shared memory (like registers)
  - accessible by all threads in the same threadblock
  - lifetime: block lifetime
- Unqualified variables: stored in local memory
  - scalars and built-in vector types are stored in registers
  - arrays are stored in device memory

# EXECUTING CODES ON GPU

# __global__

```
__global__ void minimal( int* d_a)
{
*d_a = 13;
}
```

```
__global__ void assign( int* d_a, int value)
{
int idx = blockDim.x * blockIdx.x + threadIdx.x;
d_a[idx] = value;
}
```

# Launching kernels

- Modified C function call syntax:

  kernel<<<dim3 grid, dim3 block>>>(…)

  - Execution Configuration (“<<< >>>”):
  - grid dimensions: x and y
  - thread-block dimensions: x, y, and z

# EX: VecAdd

- Add two vectors, A and B, of dimension N, and put result to 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
        VecAdd<<<1, N>>>(A, B, C);
}
```

# EX: MatAdd

- Add two matrices, A and B, of dimension N, and put result to 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
	dim3 dimBlock(N, N);
	MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

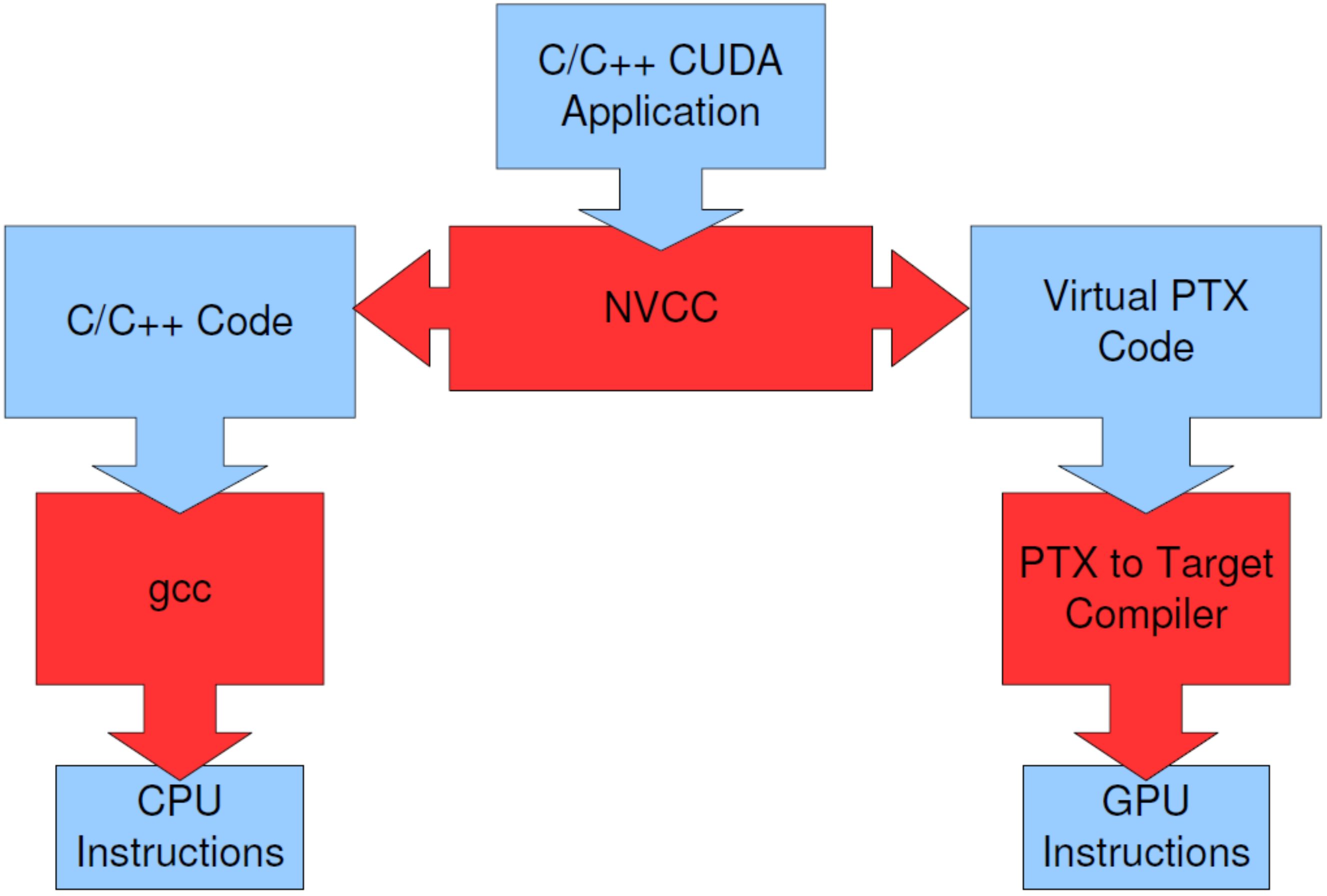
# Ex: MatAdd

```
// 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 dimBlock(16, 16);
        dim3 dimGrid((N + dimBlock.x – 1) / dimBlock.x,
                     (N + dimBlock.y – 1) / dimBlock.y);
        MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

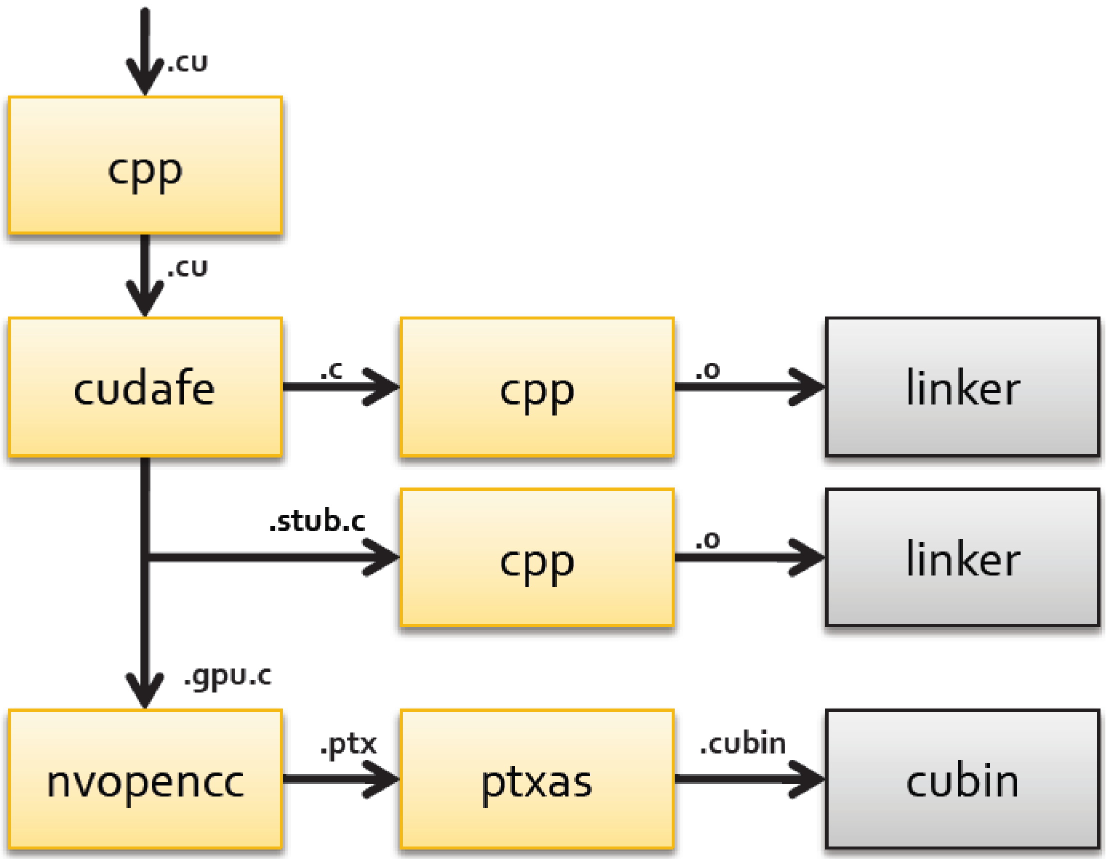
# Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Can only access GPU memory
  - Must have void return type
  - No variable number of arguments (“varargs”)
  - Not recursive
  - No static variables
- Function arguments automatically copied from CPU to GPU memory

# Compiling a CUDA Program



# Compiled files



# MEMORY MANAGEMENT

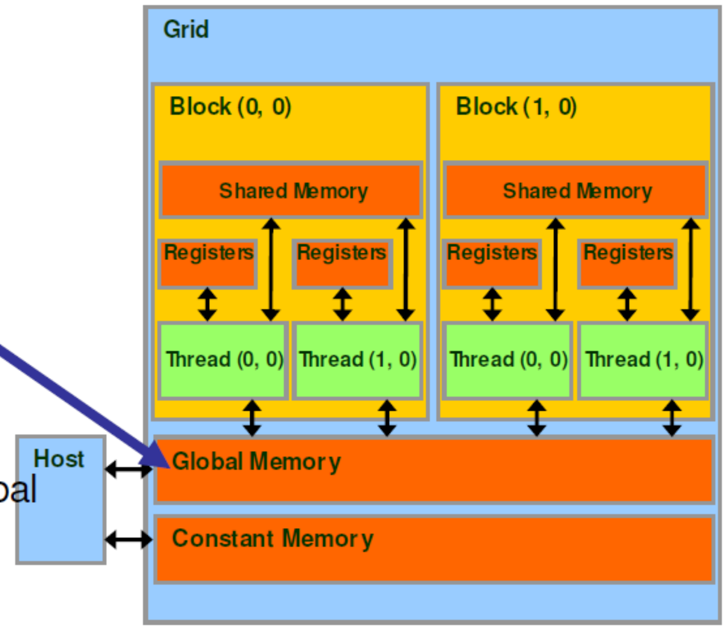
# Managing Memory

- Host (CPU) code manages device (GPU) memory:
  - Applies to *global device memory (DRAM)*
- Tasks
  - Allocate/Free
  - Copy data

# GPU Memory Allocation / Release

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

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - **Pointer** to freed object



# Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - enum cudaMemcpyKind
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice
  - Blocks CPU thread: returns after the copy is complete
  - Doesn't start copying until previous CUDA calls complete

# Ex: VecAdd

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

// Host code
int main() {
        int N = ...;
        size_t size = N * sizeof(float);
        // Allocate input h_A and h_B in host memory
        float* h_A = malloc(size);
        float* h_B = malloc(size);
        // Allocate vectors in device memory
        float *d_A, *d_B, *d_C;
        cudaMalloc((void**)&d_A, size);
        cudaMalloc((void**)&d_B, size);
        cudaMalloc((void**)&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);

    // 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);
}
```

# Shared Memory

- __shared__ : variable qualifier
- EX: parallel sum

```
__global__ void reduce0(int *g_idata, int *g_odata) {
        __shared__ int sdata[N];
        // each thread loads one element from global to shared mem
        unsigned int tid = threadIdx.x;
        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
        sdata[tid] = g_idata[i];
        // do reduction in shared mem
        …
        // write result for this block to global mem
        if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# Dynamic Shared Memory

- When the size of the shared memory is determined in the runtime.

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    // do reduction in shared mem
    …
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# How to decide the SM size?

- When CPU launches kernel function, the 3rd argument specify the size of the shared memory.

```
kernel<<<gridDim, blockDim,SMsize>>>(…)
```

# SYNCHRONIZATION

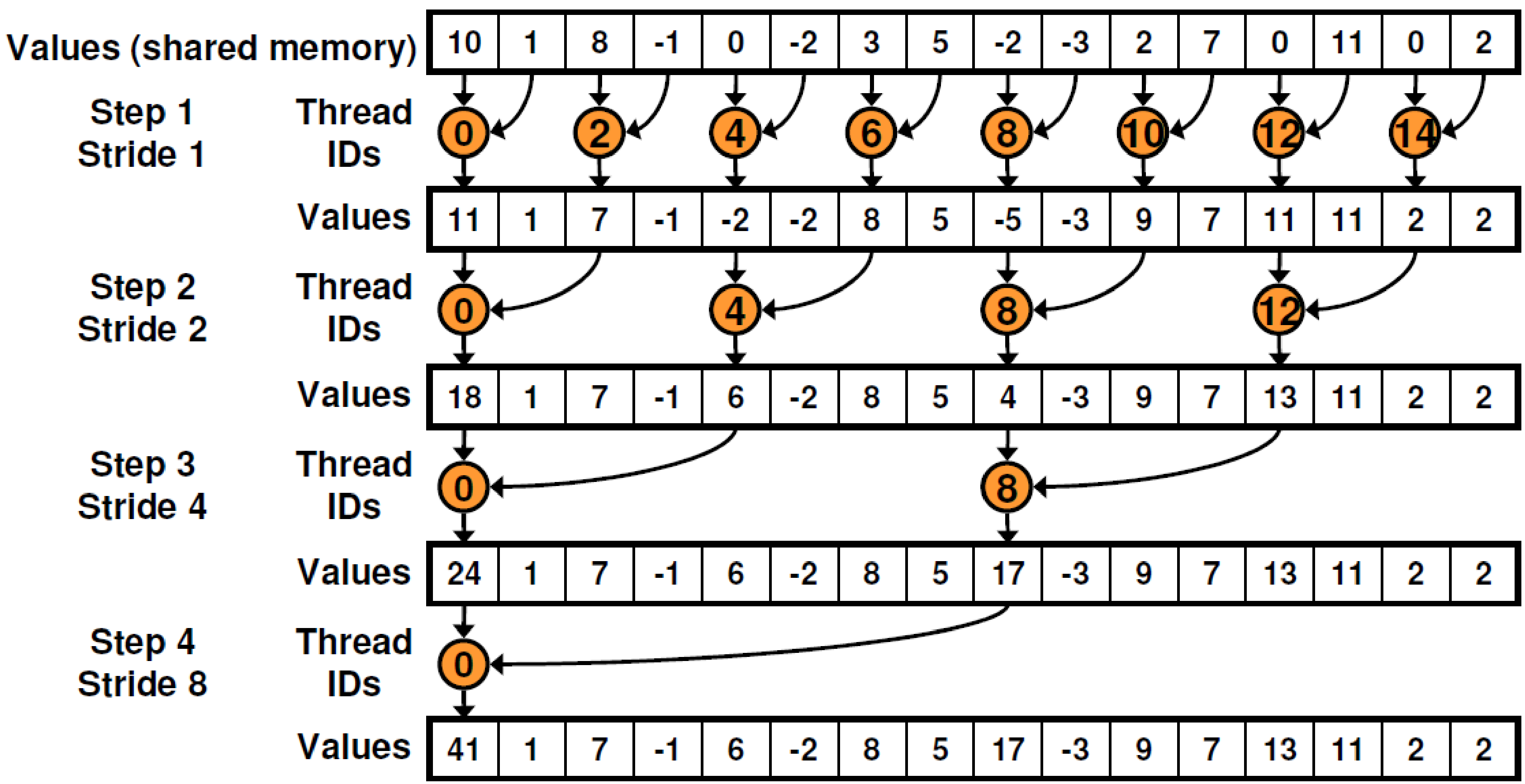
# Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- cudaMemcpy() is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete

# Device Runtime Synchronization

- void __syncthreads();
- Synchronizes all threads in a block
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared
- Allowed in conditional code only if the conditional is uniform across the entire thread block

# Ex: Parallel summation



# Ex: Parallel summation

```
__global__ void reduce0(int *g_idata, int *g_odata) {
	extern __shared__ int sdata[];
	// each thread loads one element from global to shared mem
	unsigned int tid = threadIdx.x;
	unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
	sdata[tid] = g_idata[i];
	__syncthreads();
	// do reduction in shared mem
	for(unsigned int s=1; s < blockDim.x; s *= 2) {
		if (tid % (2*s) == 0) {
				sdata[tid] += sdata[tid + s];
		}
		__syncthreads();
	}
	// write result for this block to global mem
	if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# Homework

- Read programming guide chap 1 and chap 2
- Implement matrix-matrix multiplication.
  - C=A*B, where A,B,C are NxN matrices.
  - C[i][j]=sum_{k=1,...,N} A[i][k]*B[k][j]
  - Let each thread compute one C[i][j]
  - Try (1) not to use shared memory and (2) use shared memory