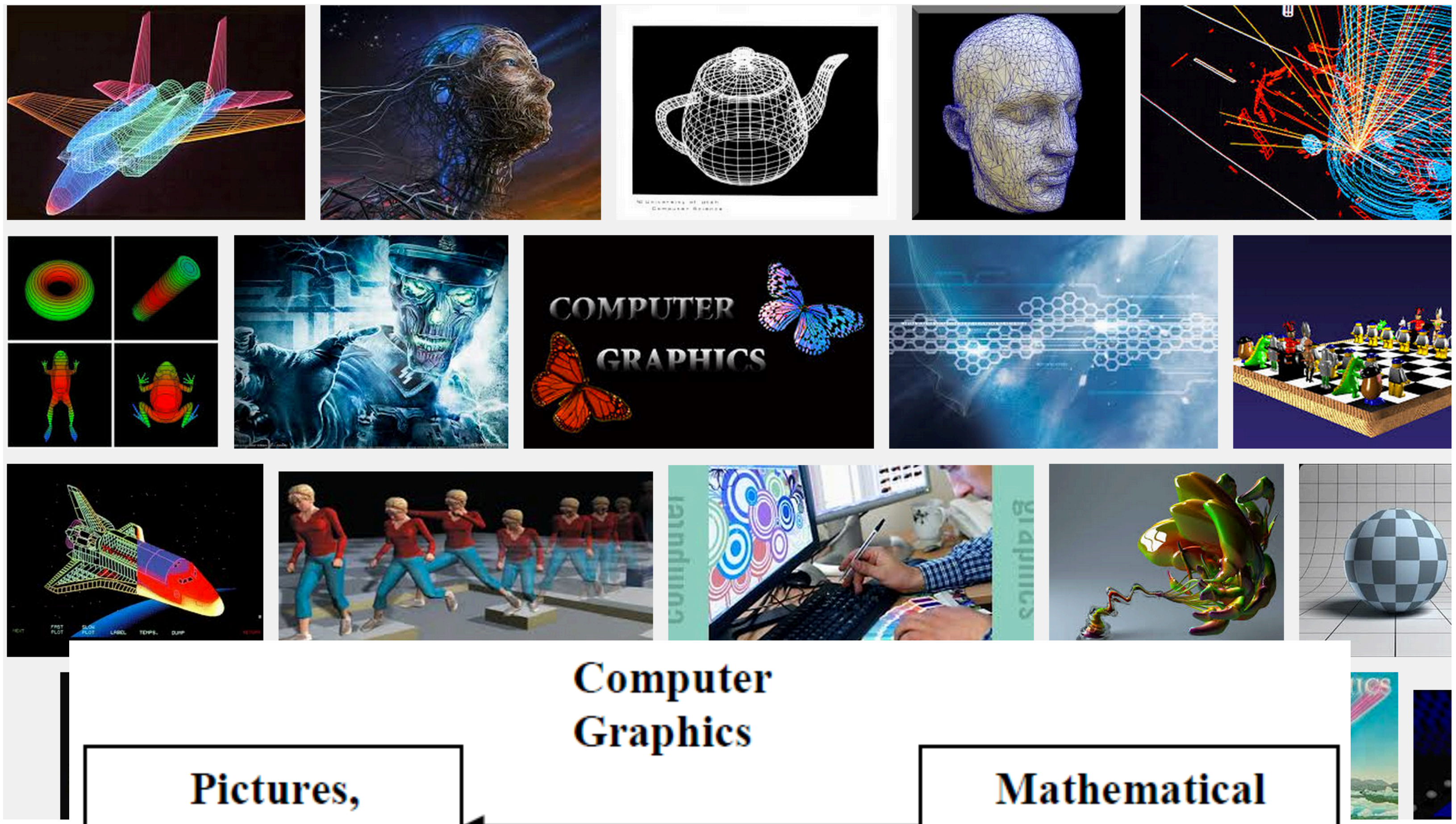
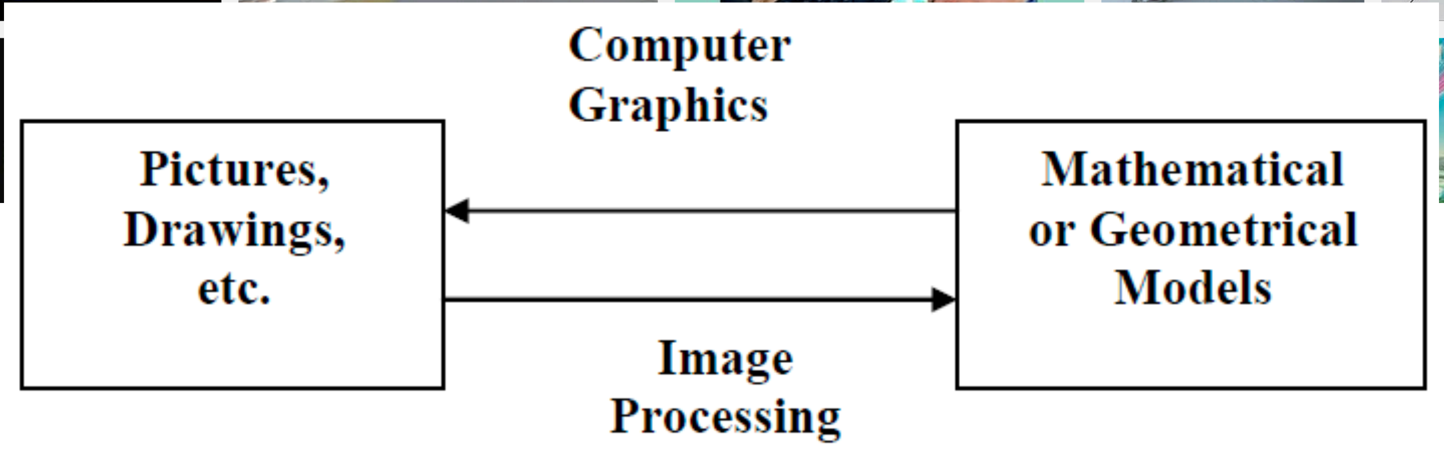
Lecture: Manycore GPU Architectures and CUDA Programming, Review

CSCE 569 Parallel Computing

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
https://passlab.github.io/CSCE569/

Computer Graphics



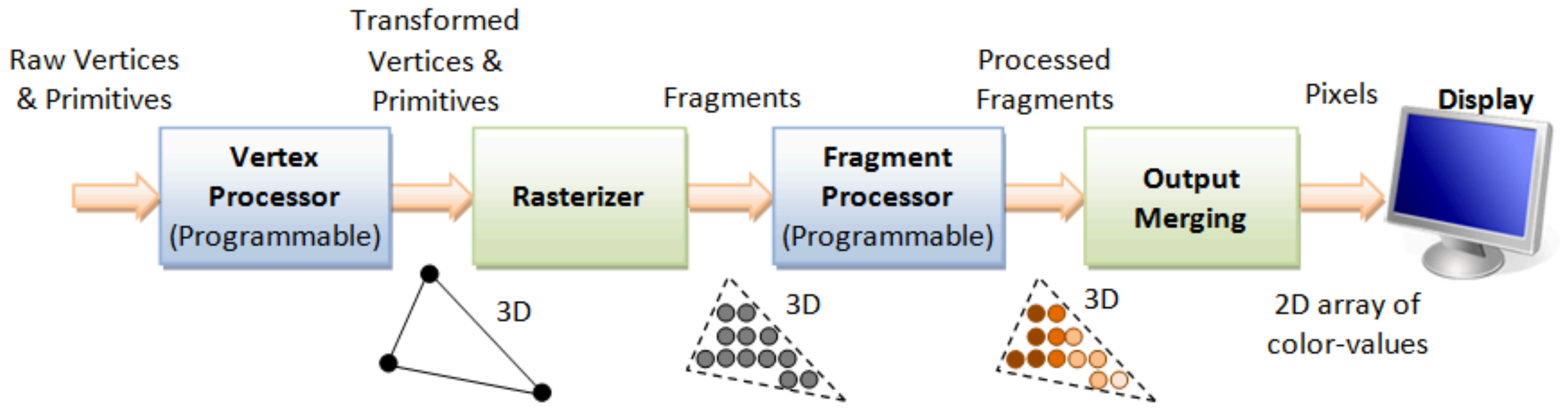


What is GPU Today?

- It is a **processor** optimized for 2D/3D graphics, video, visual computing, and display.
- It is **highly parallel, highly multithreaded multiprocessor** optimized for visual computing.
- It provide real-time visual interaction with **computed objects via graphics images, and video**.
- It serves as both a programmable graphics processor and a **scalable parallel computing platform**.
  - Heterogeneous systems: combine a GPU with a CPU
- It is called as **Many-core**

GPU Architecture Revolution

- **Unified Scalar Shader Architecture**
- **Highly Data Parallel Stream Processing**


Raw Vertices & Primitives
Vertex Processor (Programmable)
Transformed Vertices & Primitives
Rasterizer
Fragments
Fragment Processor (Programmable)
Processed Fragments
Output Merging
Pixels
Display
3D
3D
3D
2D array of color-values

3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Image: http://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

An Introduction to Modern GPU Architecture, Ashu Rege, NVIDIA Director of Developer Technology
ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

Unified Shader Architecture

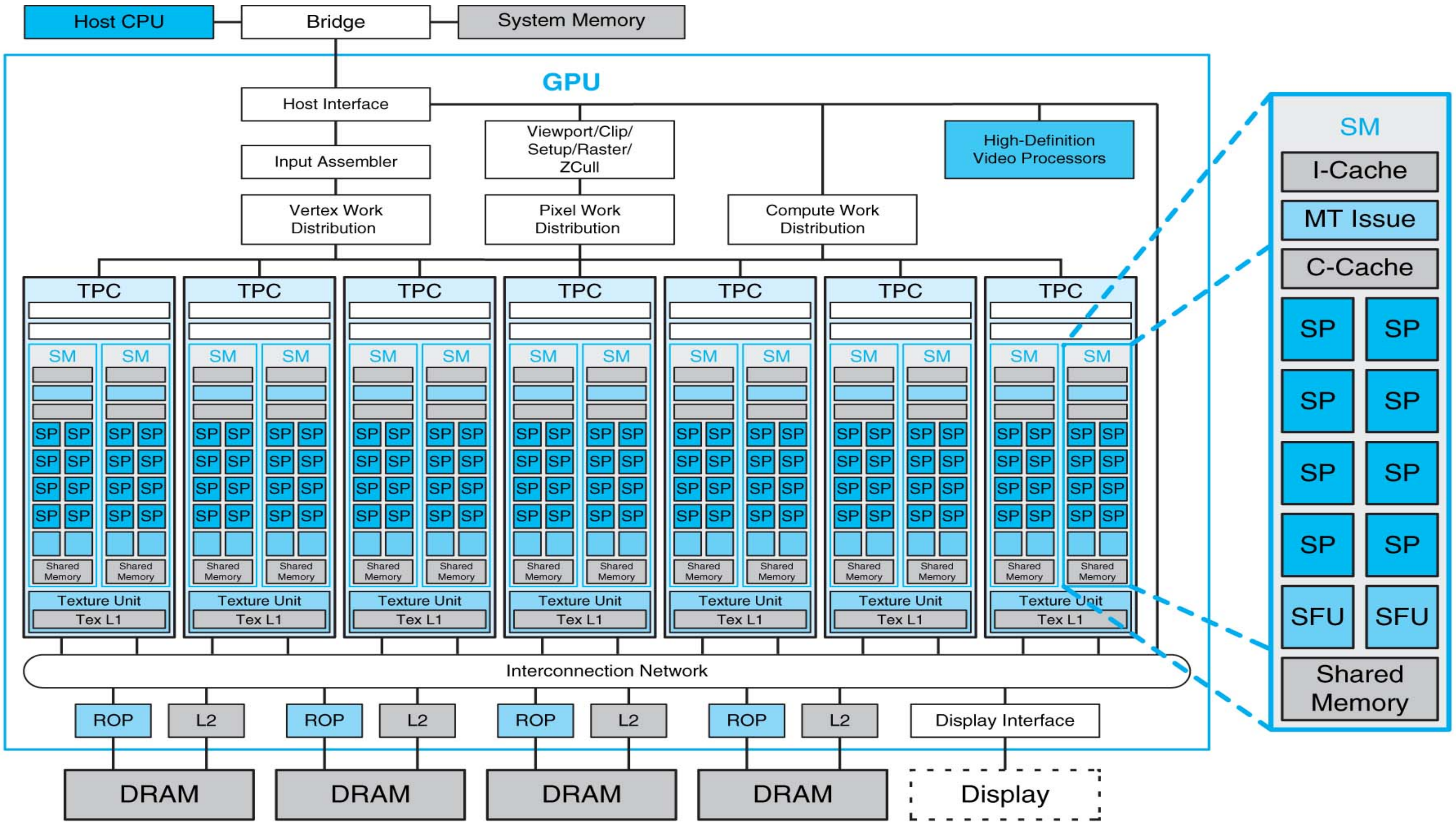
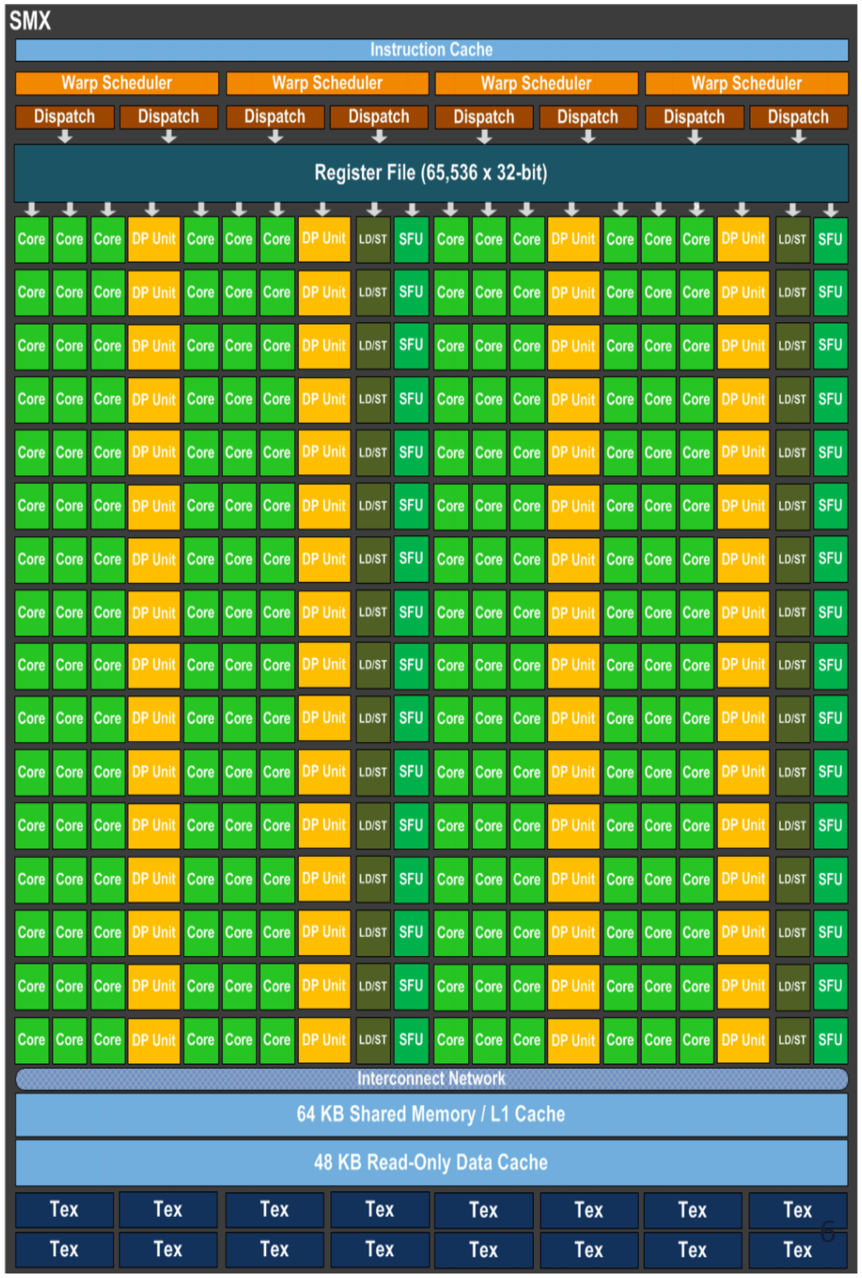


FIGURE A.2.5 Basic unified GPU architecture. Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory. Copyright © 2009 Elsevier, Inc. All rights reserved.

Multicores in NVIDIA GPUs

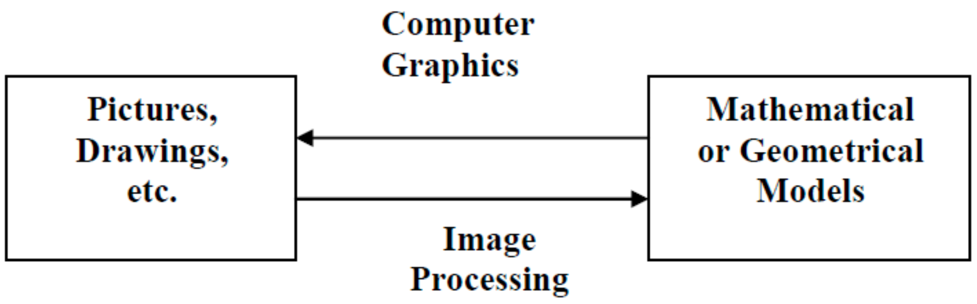
- NVIDIAGPU Streaming Multiprocessors (SM) are analogous to CPU cores
  - Single computational unit
  - Think of an SM as a single vector processor
  - Composed of multiple “cores”, load/store units, special function units (sin, cosine, etc.)
  - Each core contains integer and floating-point arithmetic logic units


SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

GPU Computing – The Basic Idea

- **Use GPU for more than just generating graphics**
  - The computational resources are there, they are most of the time underutilized

  - **The ironical fact:** It takes about 20 years (80/90s – 2007) to realize that a GPU that can do graphics well should do image processing well too.



Streaming Processing

To be efficient, GPUs must have *high throughput*, i.e. processing millions of pixels in a single frame, but may be high latency

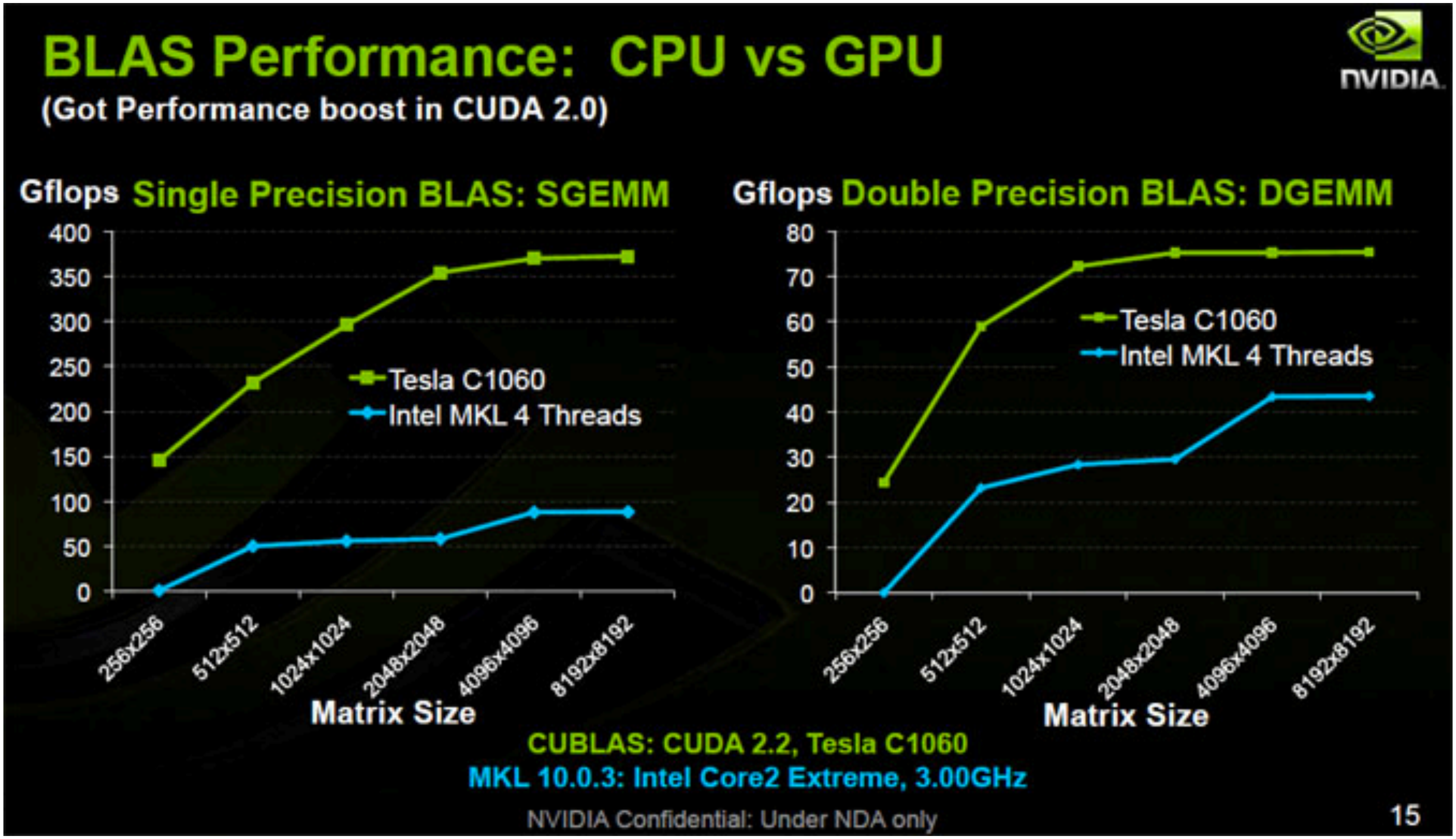
- "Latency is a *time delay* between the moment something is initiated, and the moment one of its effects begins or becomes detectable"
- For example, the time delay between a request for texture reading and texture data returns
- Throughput is the amount of work done in a given amount of time
  - CPUs are low latency low throughput processors
  - GPUs are high latency high throughput processors

Streaming Processing to Enable Massive Parallelism

- Given a (typically large) set of data(“stream”)
- Run the same series of operations (“kernel” or “shader”) on all of the data (SIMD)

- GPUs use various optimizations to improve throughput:
- Some on chip memory and local caches to reduce bandwidth to external memory
- Batch groups of threads to minimize incoherent memory access
  - Bad access patterns will lead to higher latency and/or thread stalls.
- Eliminate unnecessary operations by exiting or killing threads

GPU Performance Gains Over CPU



Parallelism in CPUs v. GPUs

- Multi-/many- core/CPUs use ***task parallelism***
  - MIMD, i.e. Multiple tasks map to multiple threads
  - Tasks run different instructions
  - 10s of relatively heavyweight threads run on 10s of cores
  - Each thread managed and scheduled explicitly
  - Each thread has to be individually programmed (MPMD)

- Manycore GPUs use ***data parallelism***
  - SIMD model (Single Instruction Multiple Data)
  - Same instruction on different data
  - 10,000s of lightweight threads on 100s of cores
  - Threads are managed and scheduled by hardware
  - Programming done for batches of threads (e.g. one pixel shader per group of pixels, or draw call)

Graphics Processing Unit (GPU)

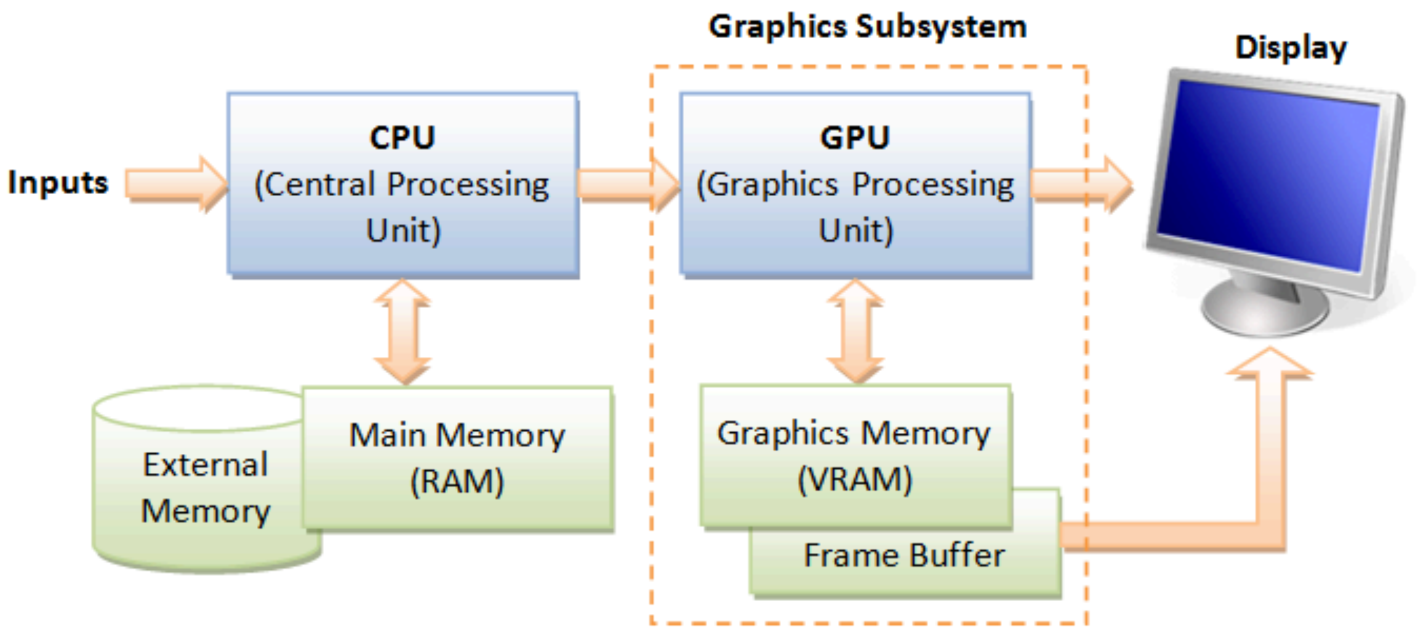
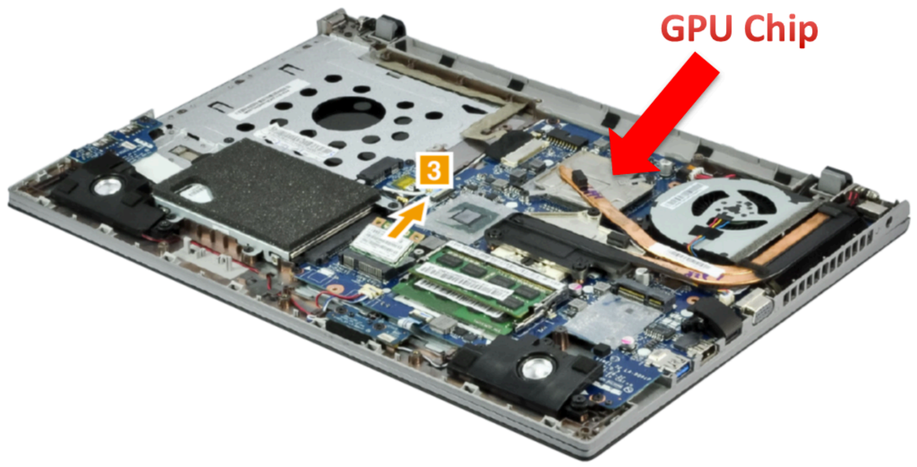
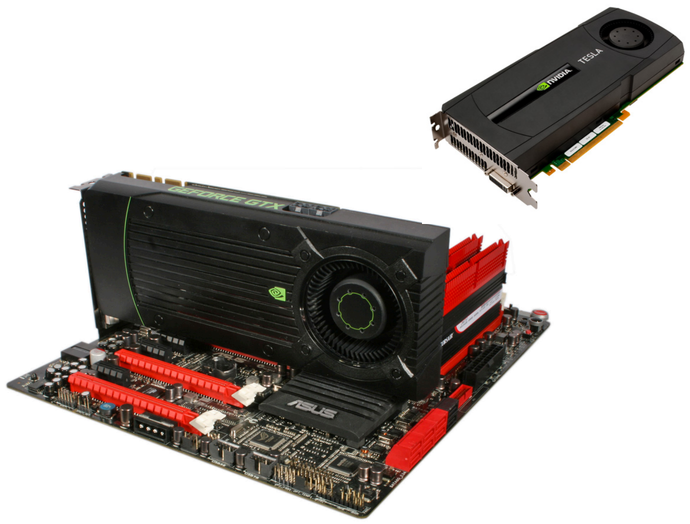
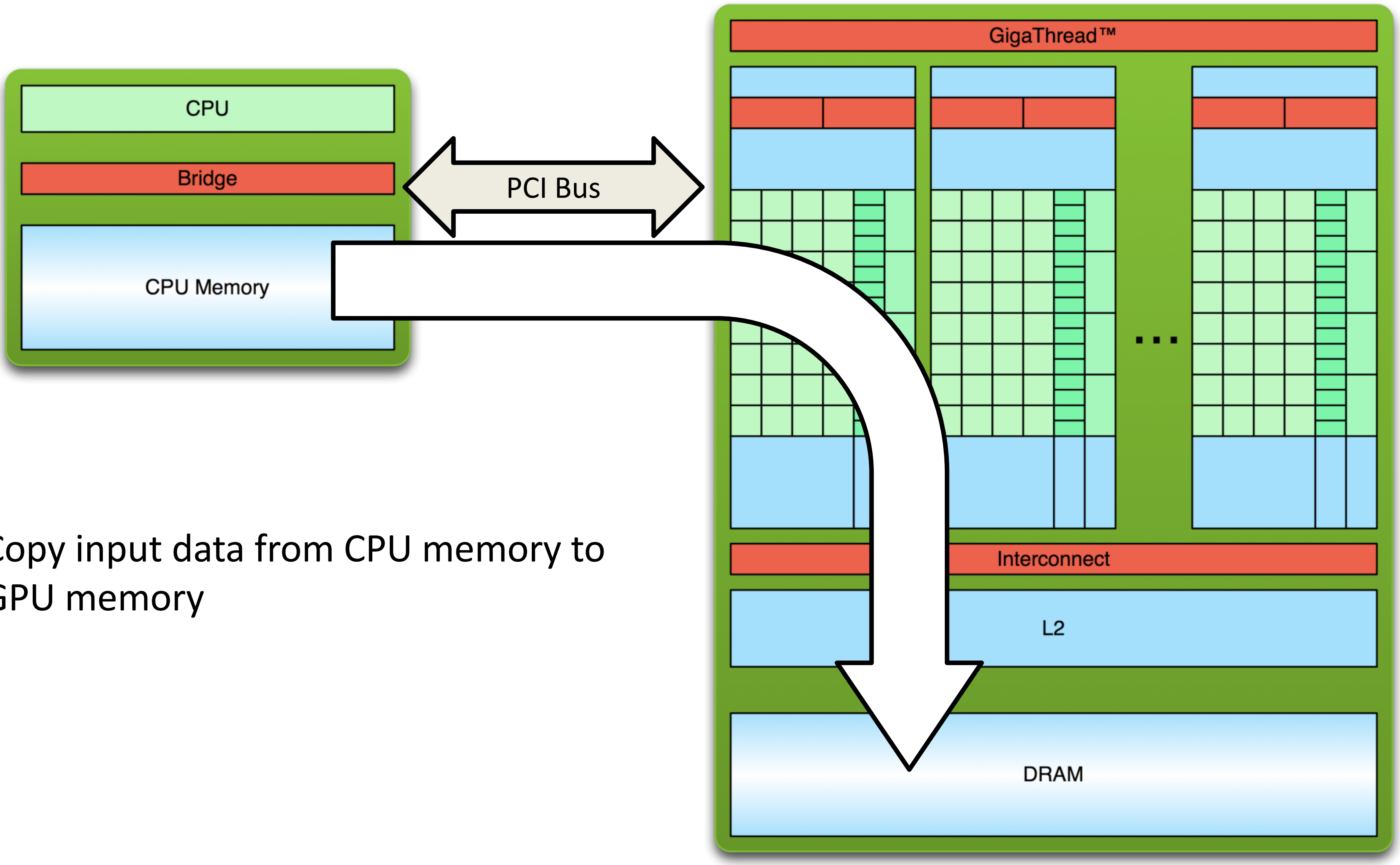


Image: http://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html



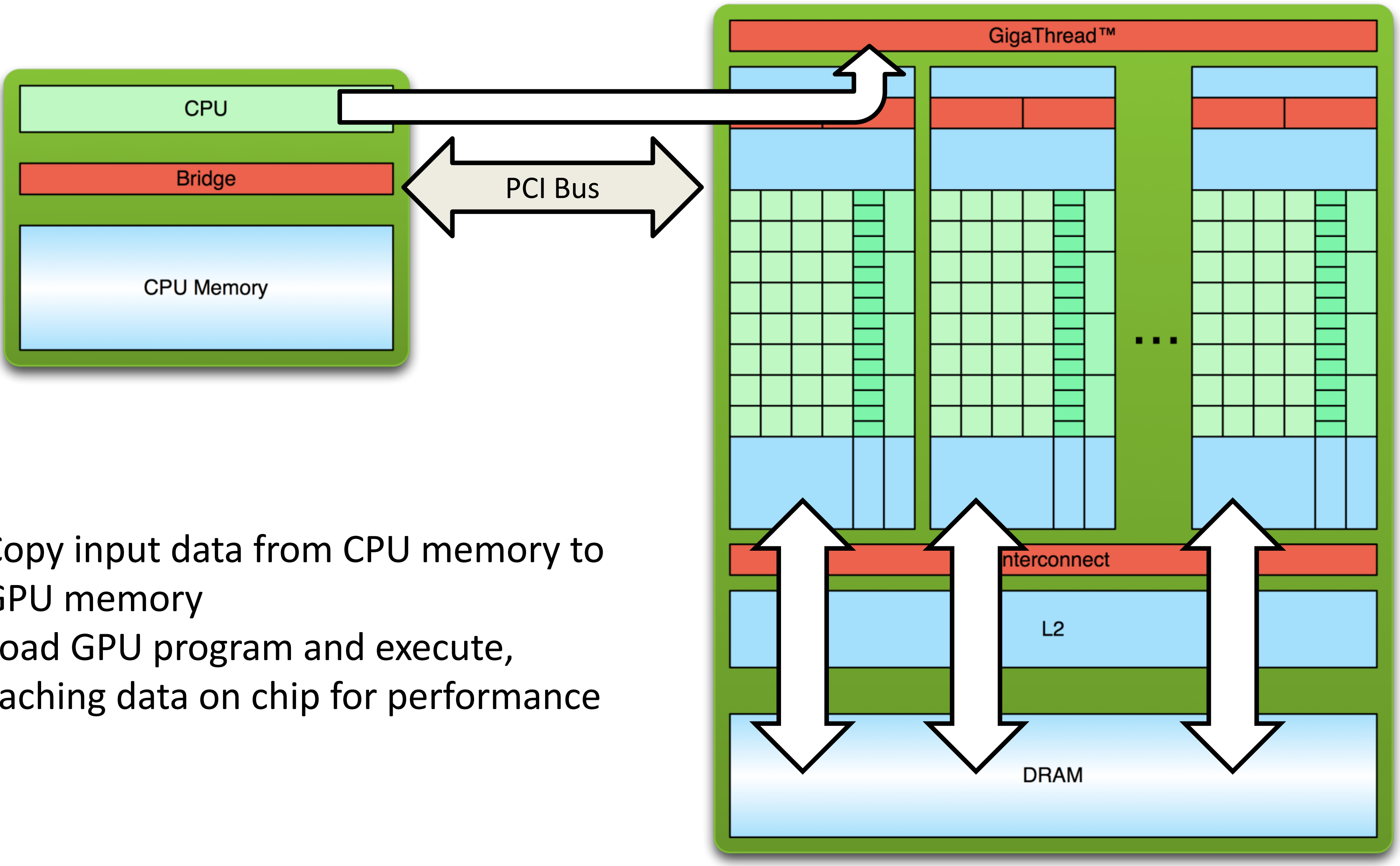


Simple Processing Flow



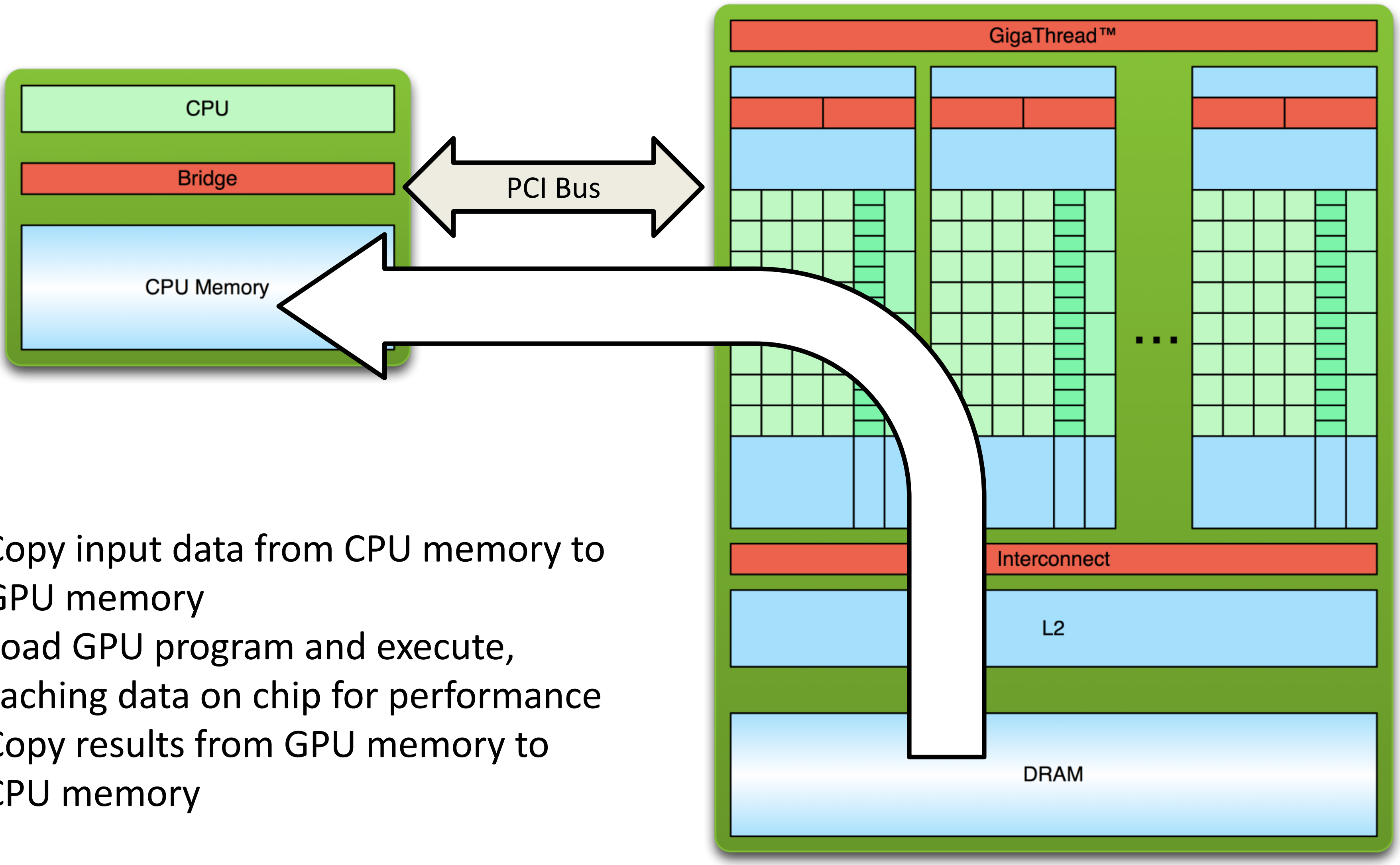
1. Copy input data from CPU memory to GPU memory

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

Offloading Computation

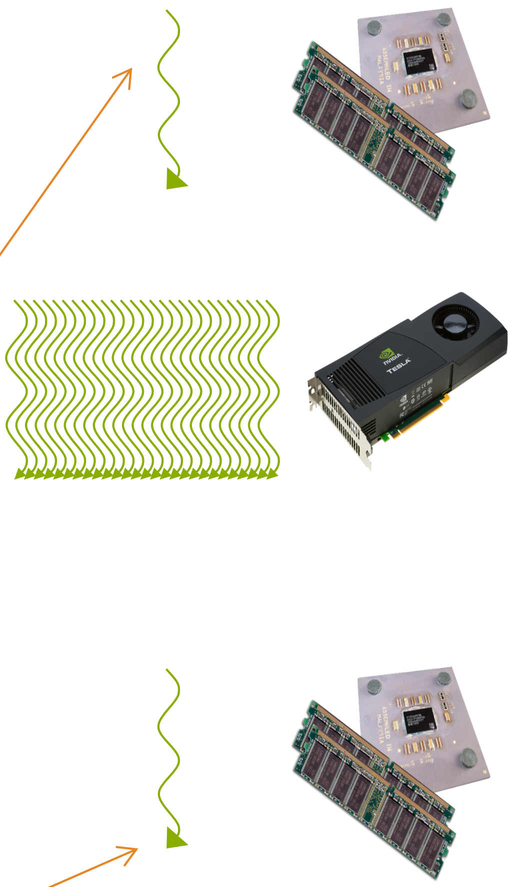
```
#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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 exe on GPU

serial code



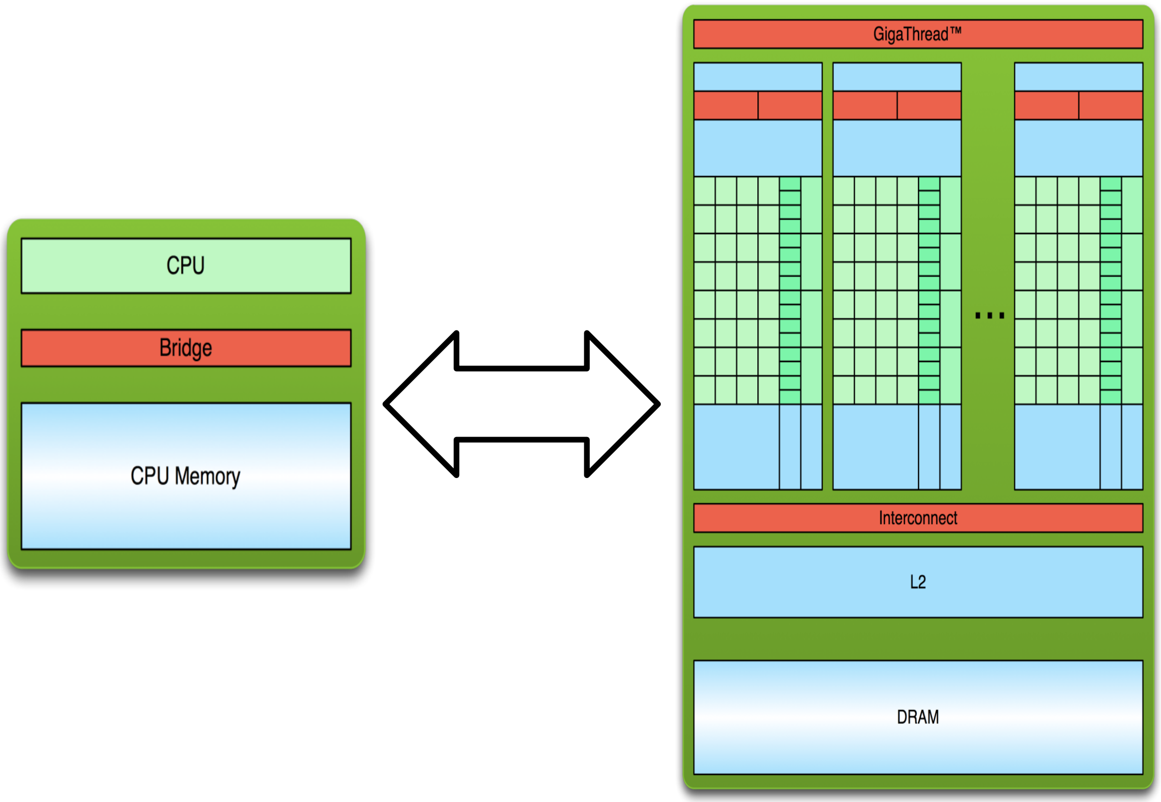
CUDA(Compute Unified Device Architecture)

Both an *architecture* and *programming model*

- Architecture and execution model
  - Introduced in NVIDIA in 2007
  - Get highest possible execution performance requires understanding of hardware architecture
- Programming model
  - Small set of extensions to C
  - Enables GPUs to execute programs written in C
  - Within C programs, call SIMT “kernel” routines that are executed on GPU.
- Hello world introduction today
  - More in later lectures

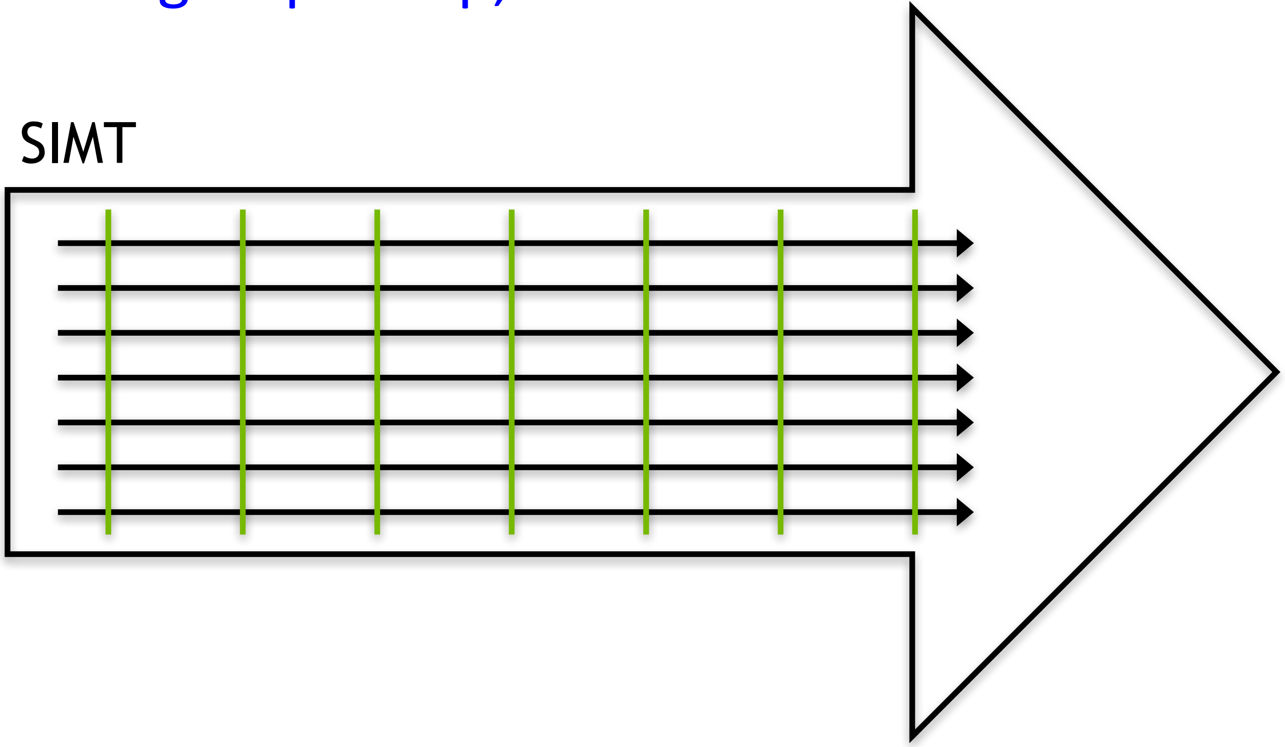
GPU Execution Model

- The GPU is a physically separate processor from the CPU
  - Discrete vs. Integrated
- The GPU Execution Model offers different abstractions from the CPU to match the change in architecture



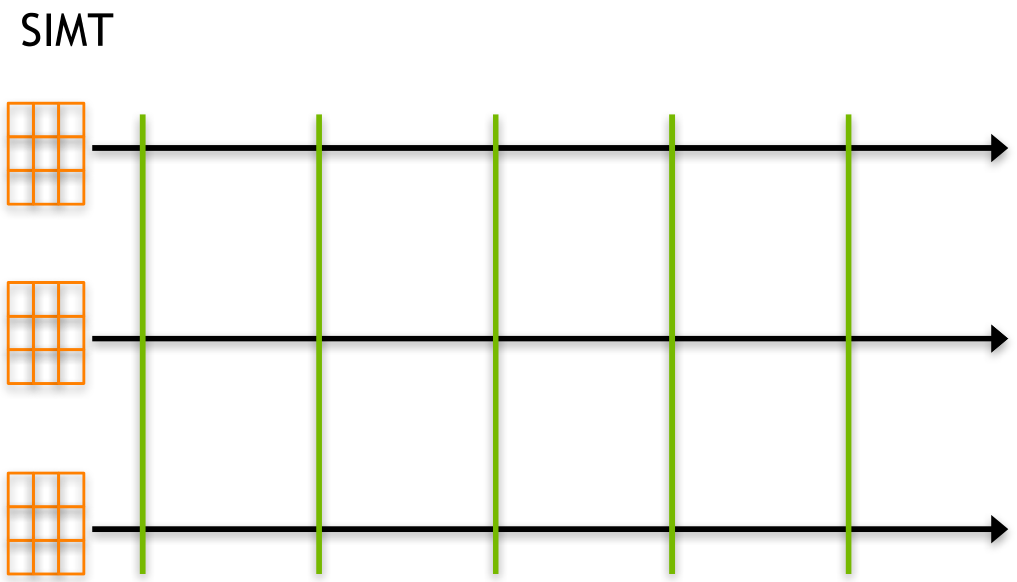
GPU Multi-Threading

- Uses the Single-Instruction, Multiple-Thread model
  - Many threads execute the same instructions in lock-step
  - Implicit synchronization after every instruction (think vector parallelism)
  - A SMIT group: wrap, 32 threads.



GPU Multi-Threading

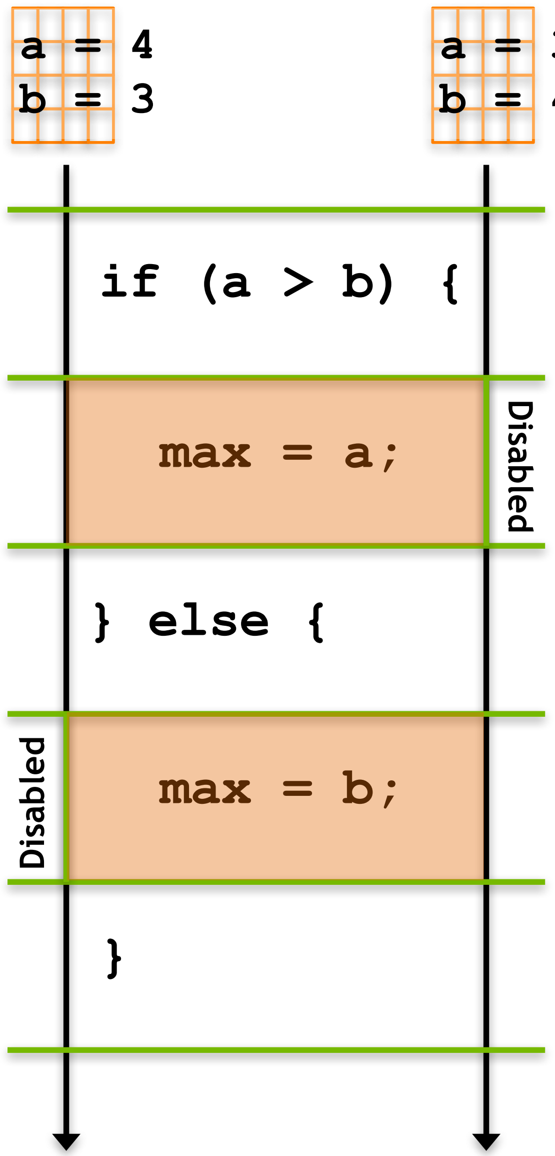
- In SIMT, all threads share instructions but operate on their own private registers, allowing threads to store thread-local state



GPU Multi-Threading

- SIMT threads can be "**disabled**" when they need to execute instructions different from others in their group

- Improves the flexibility of the SIMT model, relative to similar vector-parallel models (SIMD)



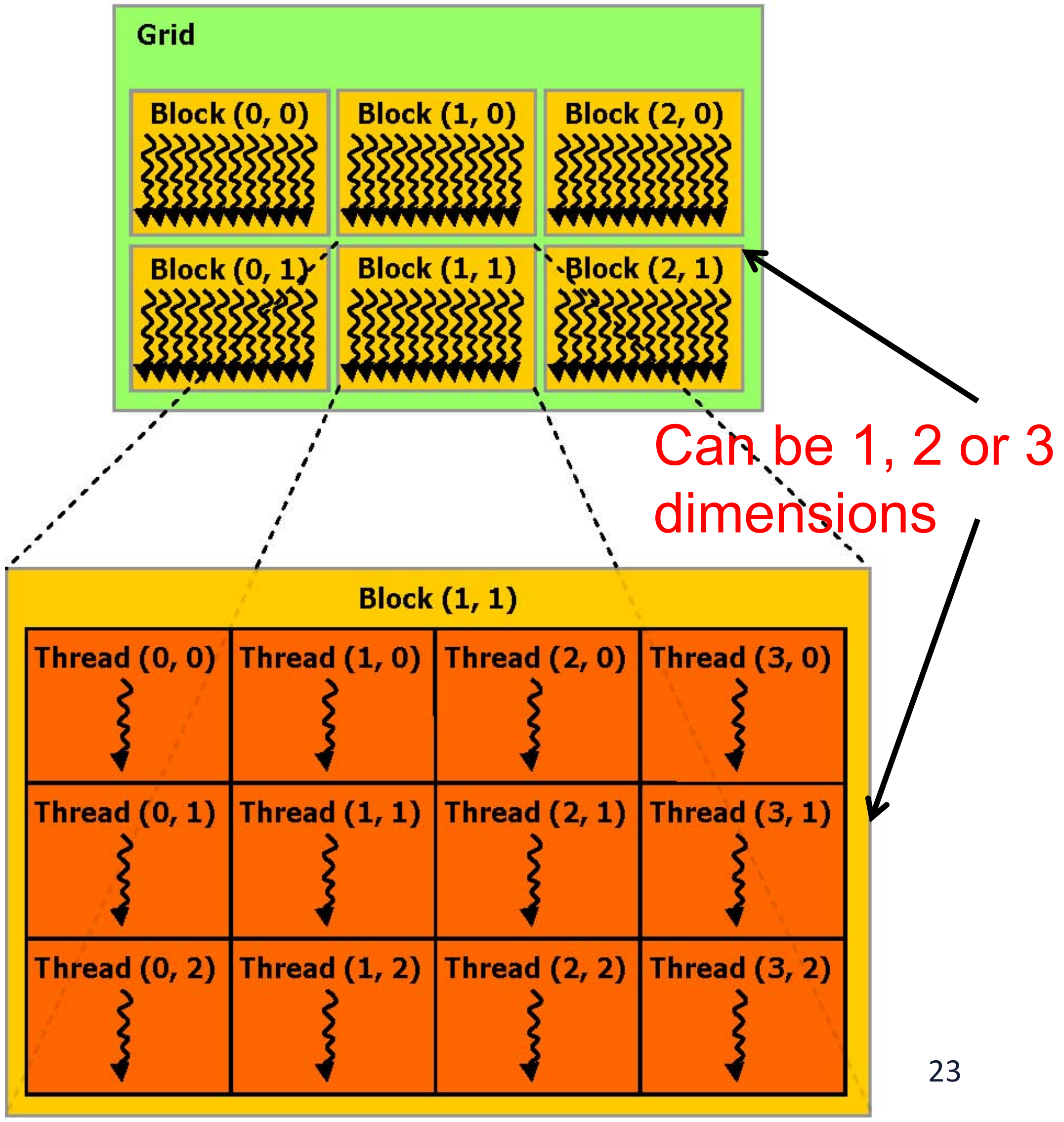
Execution Model to Hardware

- GPUs can execute multiple SIMT groups on each SM
  - For example: on NVIDIA GPUs a SIMT group is 32 threads, each Kepler SM has 192 CUDA cores → simultaneous execution of 6 SIMT groups on an SM
- SMs can support more concurrent SIMT groups than core count would suggest
  - Each thread persistently stores its own state in a private register set
  - Many SIMT groups will spend time blocked on I/O, not actively computing
  - Keeping blocked SIMT groups scheduled on an SM would waste cores
  - Groups can be swapped in and out without worrying about losing state

CUDA Thread Hierarchy

```
stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);
```

- Allows flexibility and efficiency in processing 1D, 2-D, and 3-D data on GPU.
- Linked to internal organization
- Threads in one block execute together.



How are GPU threads organized?

- On the GPU, the number of blocks and threads per block is exposed through intrinsic thread coordinate variables:
  - **Dimensions**
  - **IDs**

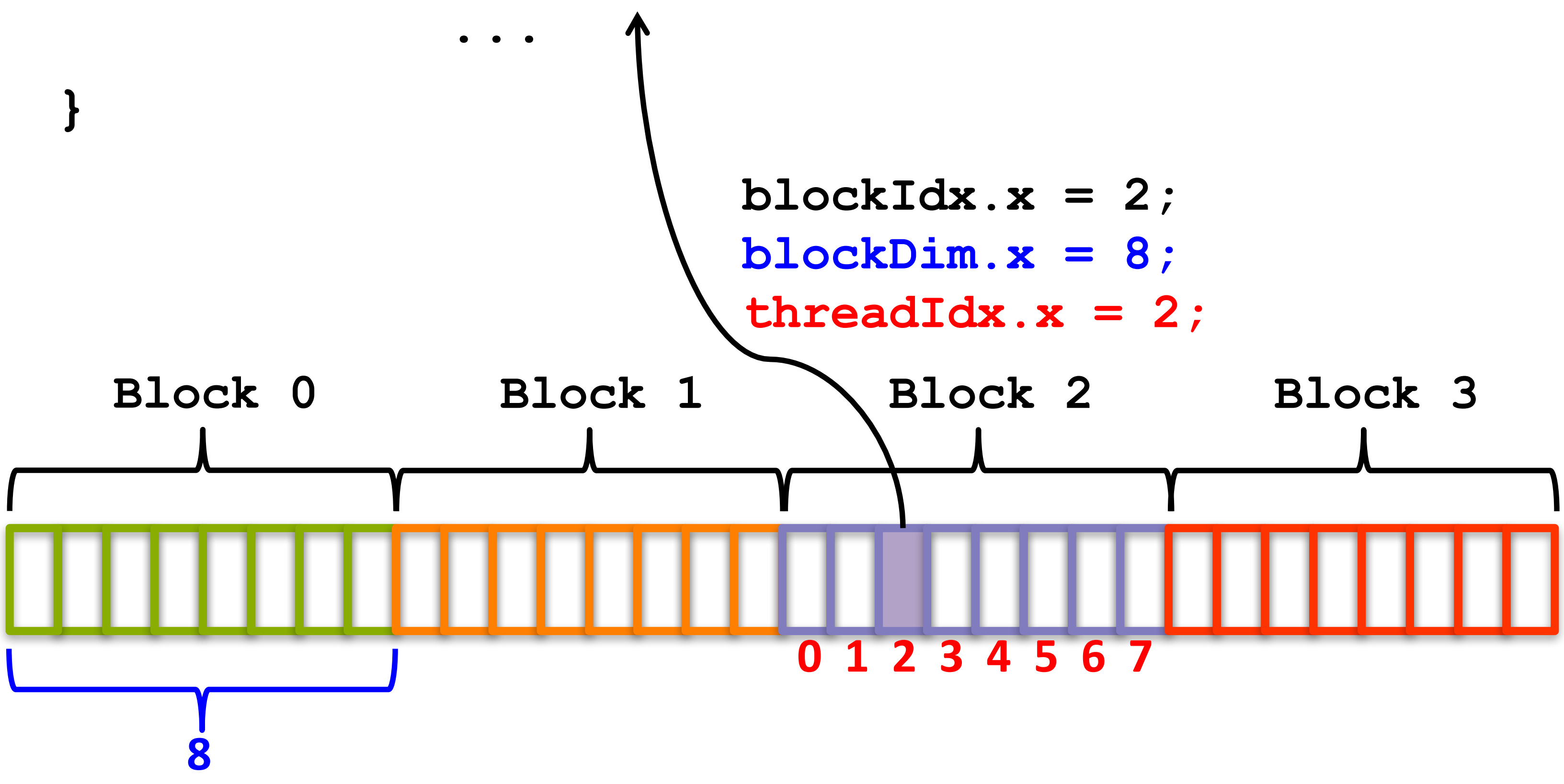
Variable	Meaning
`gridDim.x, gridDim.y, gridDim.z`	Number of blocks in a kernel launch.
`blockIdx.x, blockIdx.y, blockIdx.z`	Unique ID of the block that contains the current thread.
`blockDim.x, blockDim.y, blockDim.z`	Number of threads in each block.
`threadIdx.x, threadIdx.y, threadIdx.z`	Unique ID of the current thread within its block.

How are GPU threads organized?

to calculate a **globally unique ID** for a thread on the GPU inside a one-dimensional grid and one-dimensional block:

```
kernel<<<4, 8>>>(...);
   __global__ void kernel(...) {
      int id = blockIdx.x * blockDim.x + threadIdx.x;
                  ...
   }
```

```
blockIdx.x = 2;
blockDim.x = 8;
threadIdx.x = 2;
```



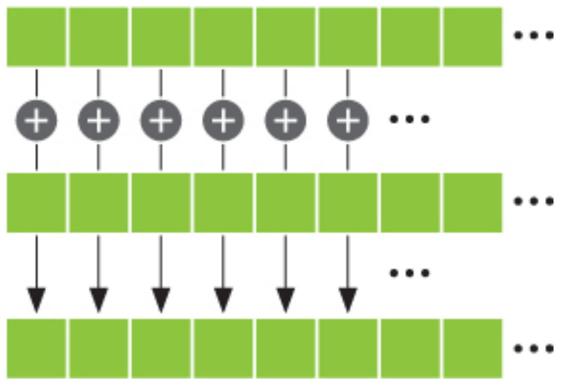
How is GPU memory managed?

- CUDA Memory Management API
  - Allocation of GPU memory
  - Transfer of data from the host to GPU memory
  - Free-ing GPU memory
  - Foo(int A[][N]) { }

Host Function	CUDA Analogue
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`free`	`cudaFree`

AXPY Offloading To a GPU using CUDA

```
// CUDA kernel. Each thread takes care of one element of c
__global__ void axpy(REAL *x, REAL *y, int n, REAL a) {
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    if (id < n) y[id] += a * x[id];
}

int main( int argc, char* argv[] ) {

    // ... init host a, x and y
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_x, size);
    cudaMalloc(&d_y, size);

    // Copy host vectors to device
    cudaMemcpy( d_x, h_x, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_y, h_y, size, cudaMemcpyHostToDevice);

    int blockSize, gridSize;
    blockSize = 1024;
    gridSize = (int)ceil((float)n/blockSize);
    axpy<<<gridSize, blockSize>>>(d_x, d_y, n, a);

    // Copy array back to host
    cudaMemcpy( h_y, d_y, size, cudaMemcpyDeviceToHost );

    // Release device memory
    cudaFree(d_x);
    cudaFree(d_y);
}
```



Memory allocation on device

Memcpy from host to device

Launch parallel execution

Memcpy from device to host

Deallocation of dev memory

More Examples and Exercises

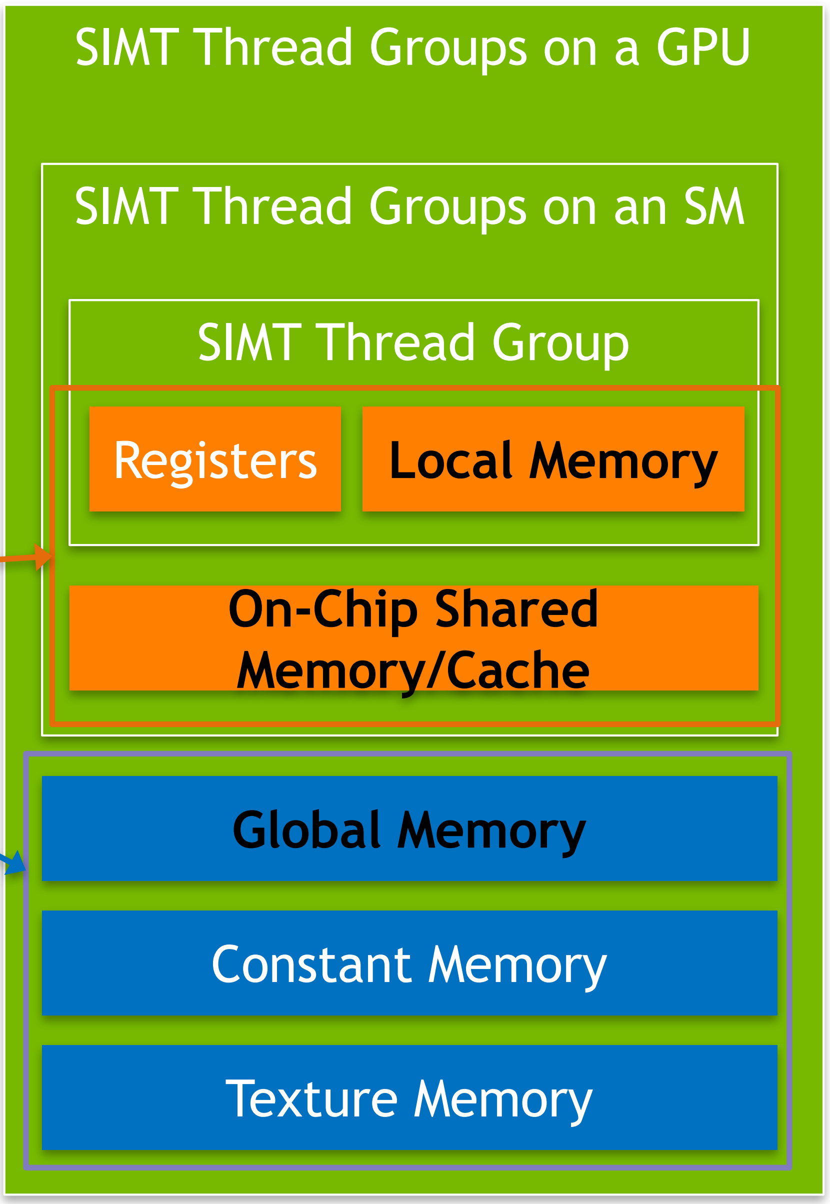
- Matvec:
  - Version 1: each thread computes one element of the final vector
  - Version 2:
- Matmul in assignment #4
  - Version 1: each thread computes one row of the final matrix C

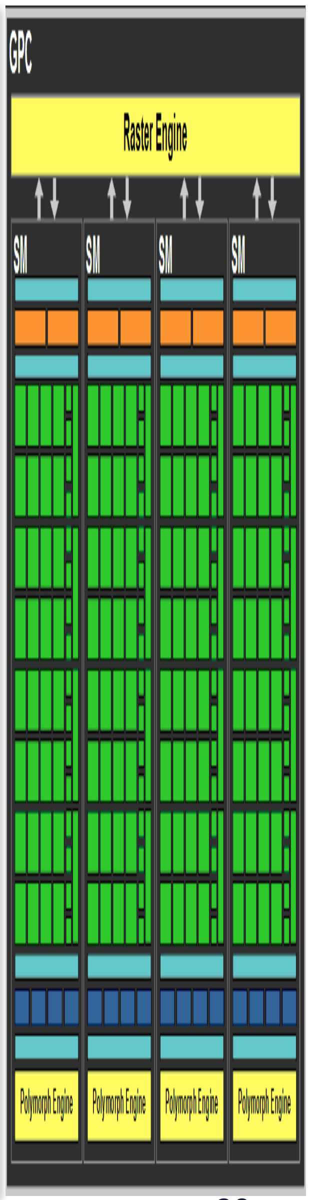
Inspecting CUDA Programs

- Debugging CUDA program:
  - `cuda-gdb` debugging tool, like gdb
- Profiling a program to examine the performance
  - `Nvprof` tool, like `gprof`
  - `Nvprof ./vecAdd`

GPU Memory Hierarchy

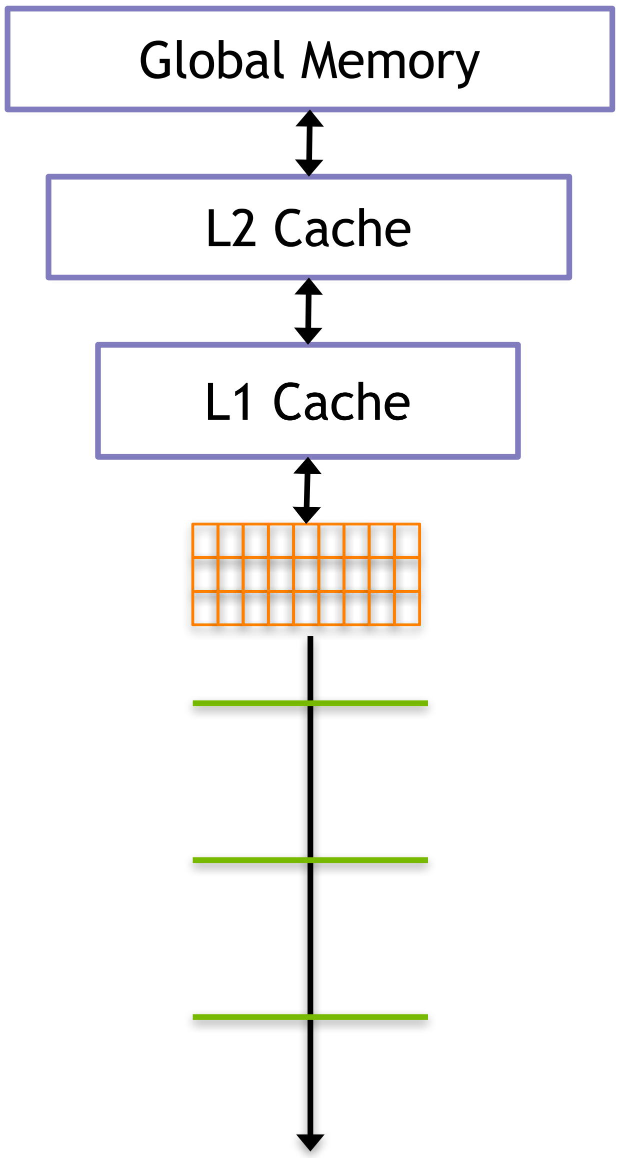
- More complex than the CPU memory
  - ***Many different*** types of memory, each with special-purpose characteristics
    - **SRAM**
    - **DRAM**
  - More ***explicit*** control over data movement


SIMT Thread Groups on a GPU
SIMT Thread Groups on an SM
SIMT Thread Group
Registers
Local Memory
On-Chip Shared Memory/Cache
Global Memory
Constant Memory
Texture Memory



Storing Data on the GPU

- **Global Memory (DRAM)**
  - **Large, high-latency memory**
  - **Stored in device memory (along with constant and texture memory)**
  - **Can be declared statically with `__device__`**
  - **Can be allocated dynamically with `cudaMalloc`**
  - **Explicitly managed by the programmer**
  - Optimized for all threads in a warp accessing neighbouring memory cells

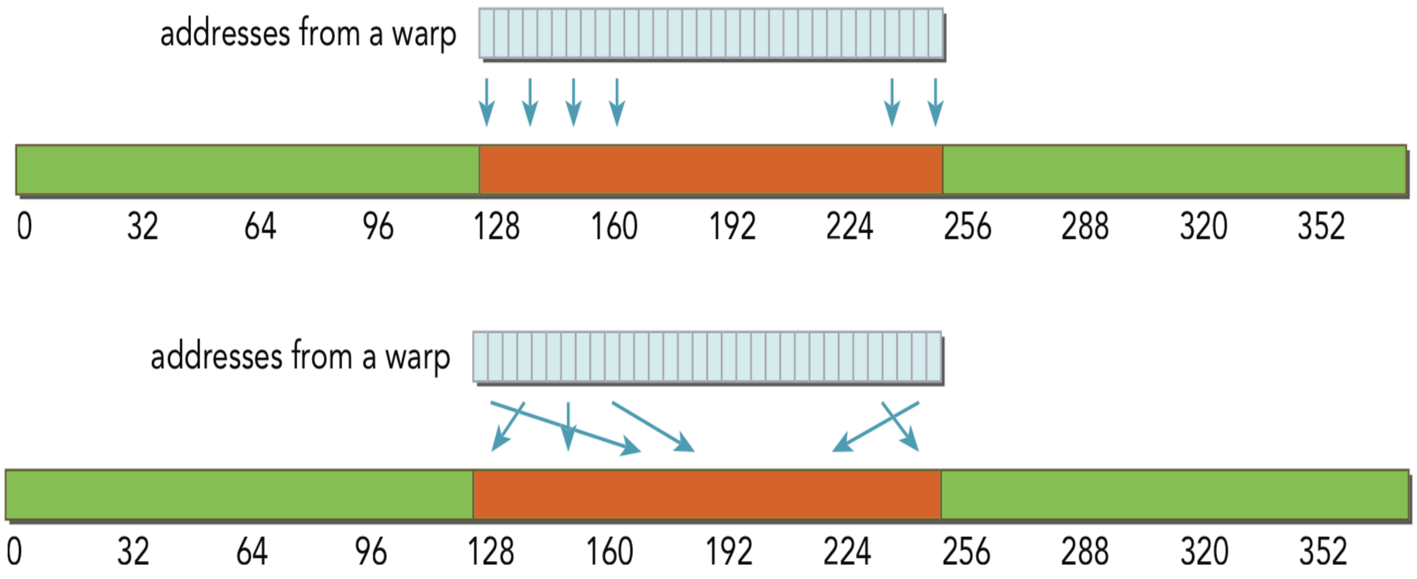


Global Memory Access Patterns

- Achieving **aligned** and **coalesced** global memory accesses is key to optimizing an application’s use of global memory bandwidth
  - Coalesced: the threads within a warp reference memory addresses that can all be serviced by a single global memory transaction (think of a memory transaction as the process of bring a cache line into the cache)
  - Aligned: the global memory accesses by threads within a warp start at an address boundary that is an even multiple of the size of a global memory transaction

Global Memory Access Patterns

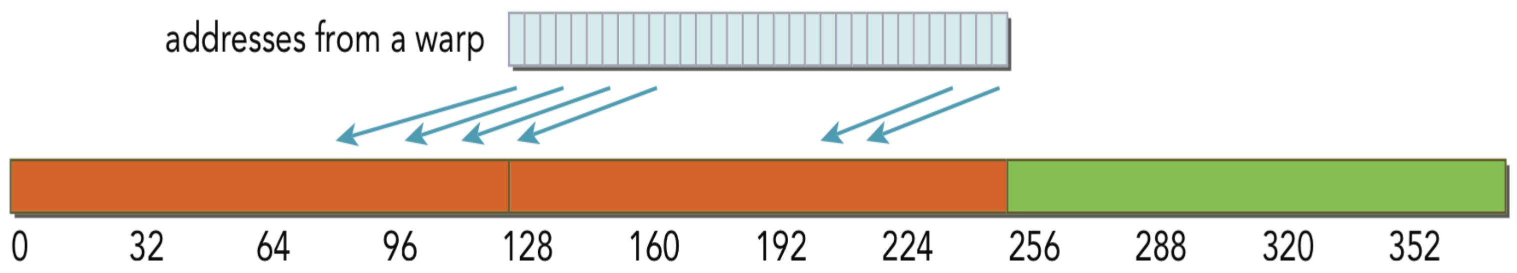
- Aligned and Coalesced Memory Access (*w/ L1 cache*)
  - 32-thread wrap, 128-bytes memory transaction



- With 128-byte access, a single transaction is required and all of the loaded bytes are used

Global Memory Access Patterns

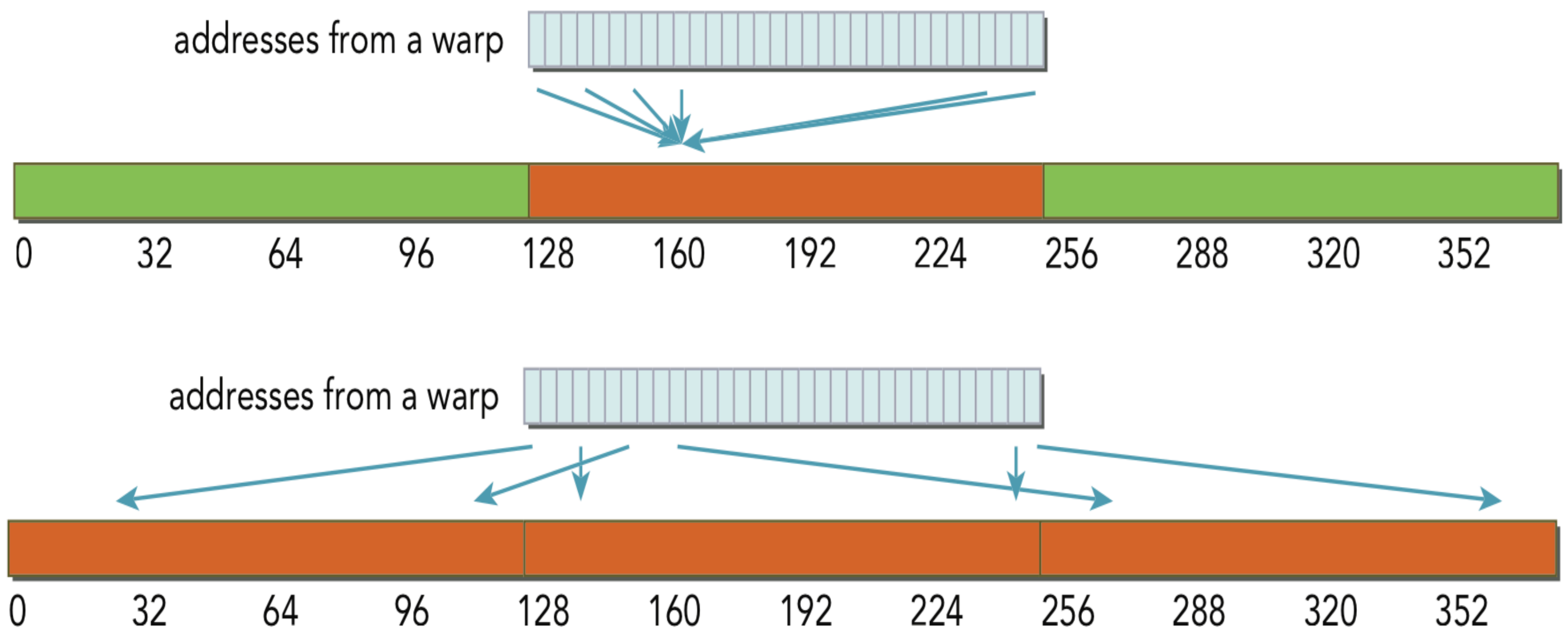
- Misaligned and Coalesced Memory Access (*w/ L1 cache*)



- With 128-byte access, two memory transactions are required to load all requested bytes. Only half of the loaded bytes are used.

Global Memory Access Patterns

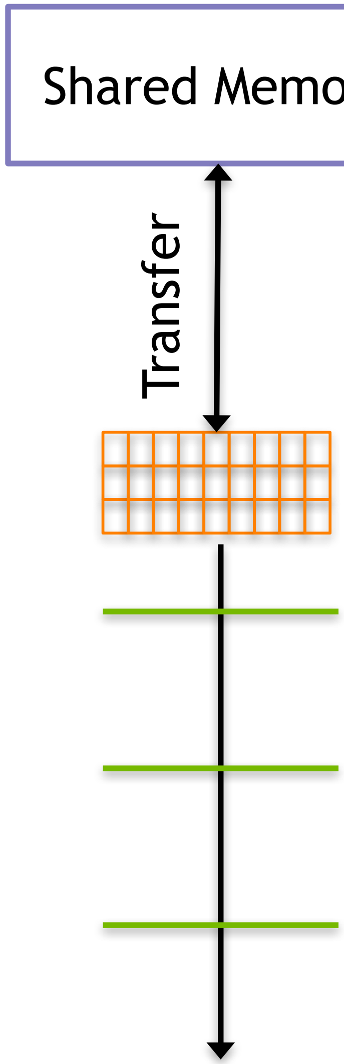
- Misaligned and Uncoalesced Memory Access (*w/ L1 cache*)



- With uncoalesced loads, many more bytes loaded than requested

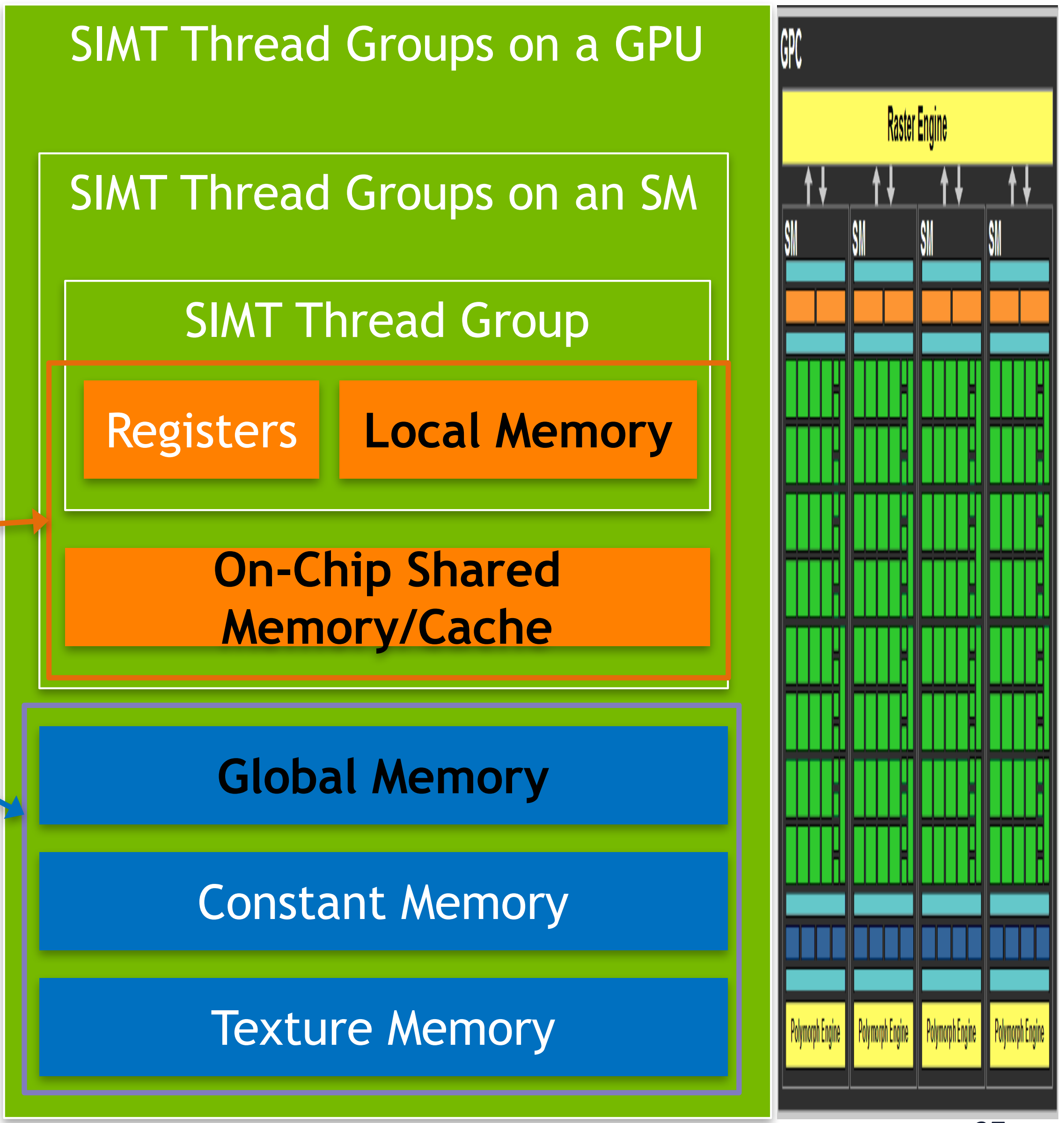
Shared Memory on the GPU

- **Shared Memory (SRAM)**
  - **Declared with the `__shared__` keyword**
  - Low-latency, high bandwidth
  - **Shared by all threads in a thread block**
  - Explicitly allocated and managed by the programmer, manual L1 cache
  - Stored on-SM, same physical memory as the GPU L1 cache
  - **On-SM memory is statically partitioned between L1 cache and shared memory**



GPU Memory Hierarchy

- More complex than the CPU memory
  - ***Many different*** types of memory, each with special-purpose characteristics
    - SRAM
    - DRAM
  - More ***explicit*** control over data movement



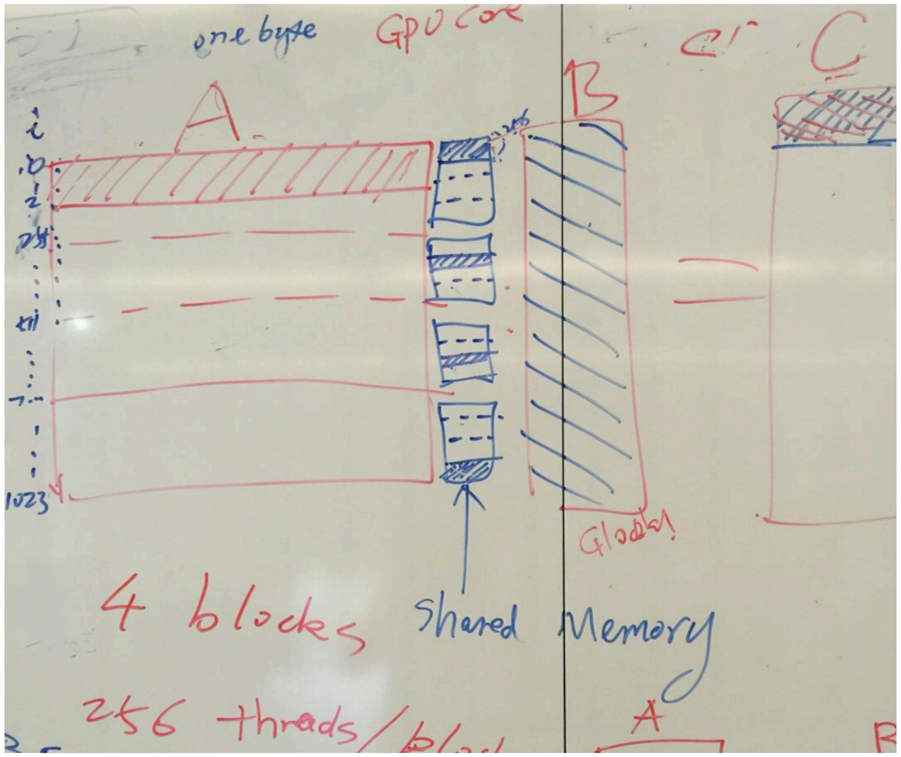
Shared Memory Allocation

- Dynamically Allocated Shared Memory
  - Size in bytes is set at kernel launch with a third kernel launch configurable
  - Can only have one dynamically allocated shared memory array per kernel
  - Must be one-dimensional arrays

```
__global__ void kernel(...) {
    extern __shared__ int s_arr[];
    ...
}

kernel<<<nblocks, threads_per_block,
shared_memory_bytes>>>(...);
```

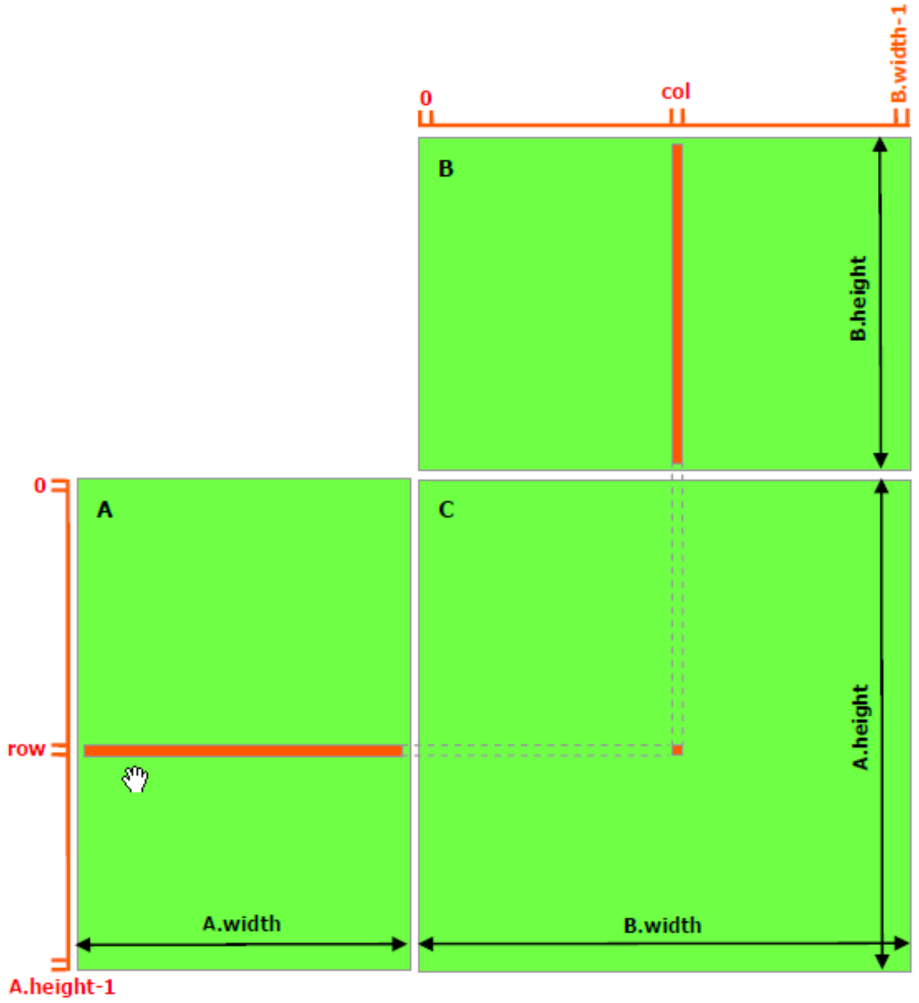
Matrix Vector Multiplication

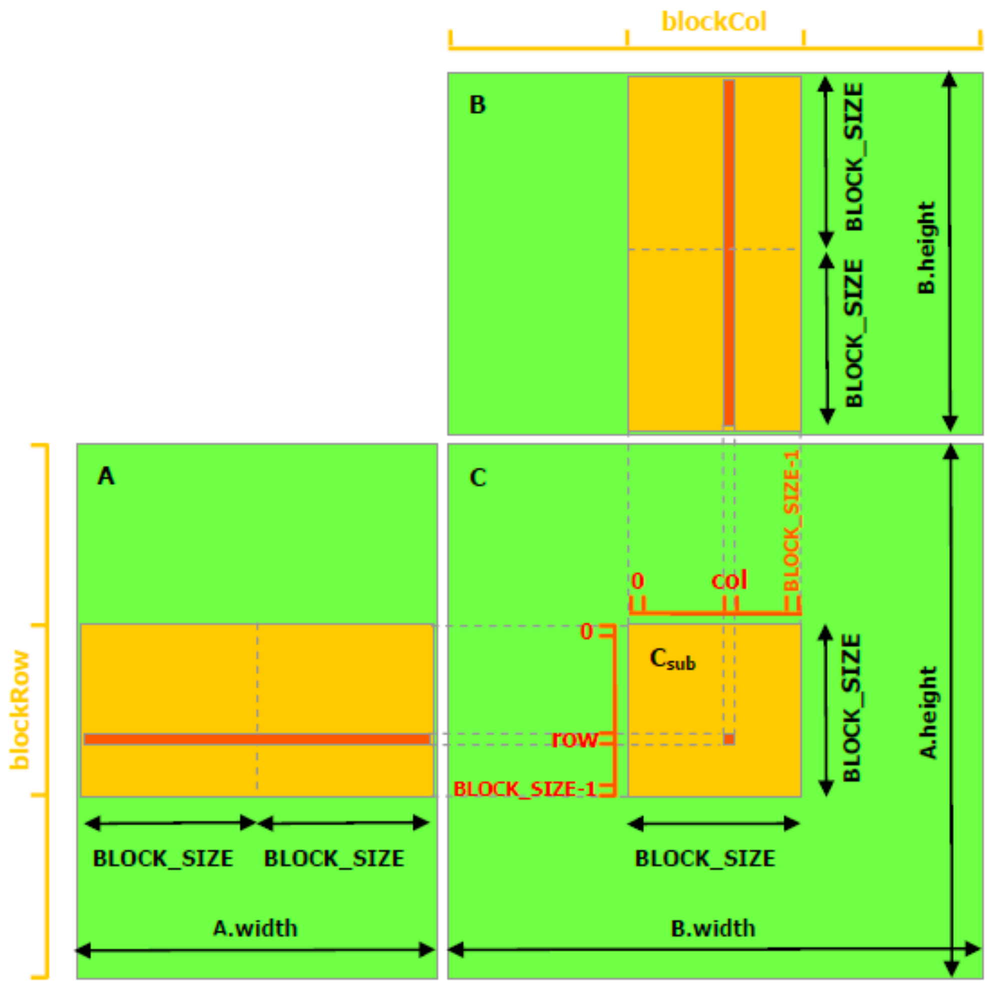
```
/** N =1024, 4 blocks, 256 threads/per block  */
__global__ void
matvec_kernel_shared(float * A, float * B, float * C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x; /* 0 - 1023 */
    int j;

    extern __shared__ float B_shared[]; /* the same size as B[1024] */
    B_shared[i] = B[i];
    /* for block 0: 0-255 are filled */
    /* for block 1: 256-511 are filled */
    /* for block 2: 512-767 are filled */
    /* for block 3: 768 - 1023 are filled */

    B_shared[(i+256)%1024] = B[(i+256)%1024];
    B_shared[(i+512)%1024] = B[(i+512)%1024];
    B_shared[(i+768)%1024] = B[(i+768)%1024];

    __syncthreads();

    if (i < N) {
      float temp = 0.0;
      for (j=0; j<N; j++)
        temp += A[i*N+j] * B_shared[j];

      C[i] = temp;
    }
}
```



Matrix Multiplication V1 and V2 in Assignment #4

- https://docs.nvidia.com/cuda/cuda-c-programming-guide/#shared-memory





GPU Memory Performance

- Data transfer from CPU to GPU over the PCI bus adds
  - Conceptual complexity
  - Performance overhead

Communication Medium	Latency	Bandwidth
On-Chip Shared Memory	A few clock cycles	Thousands of GB/s
GPU Global Memory	Hundreds of clock cycles	Hundreds of GB/s
PCI Bus	Hundreds to thousands of clock cycles	Tens of GB/s