Lecture: Manycore GPU Architectures and Programming, Part 1

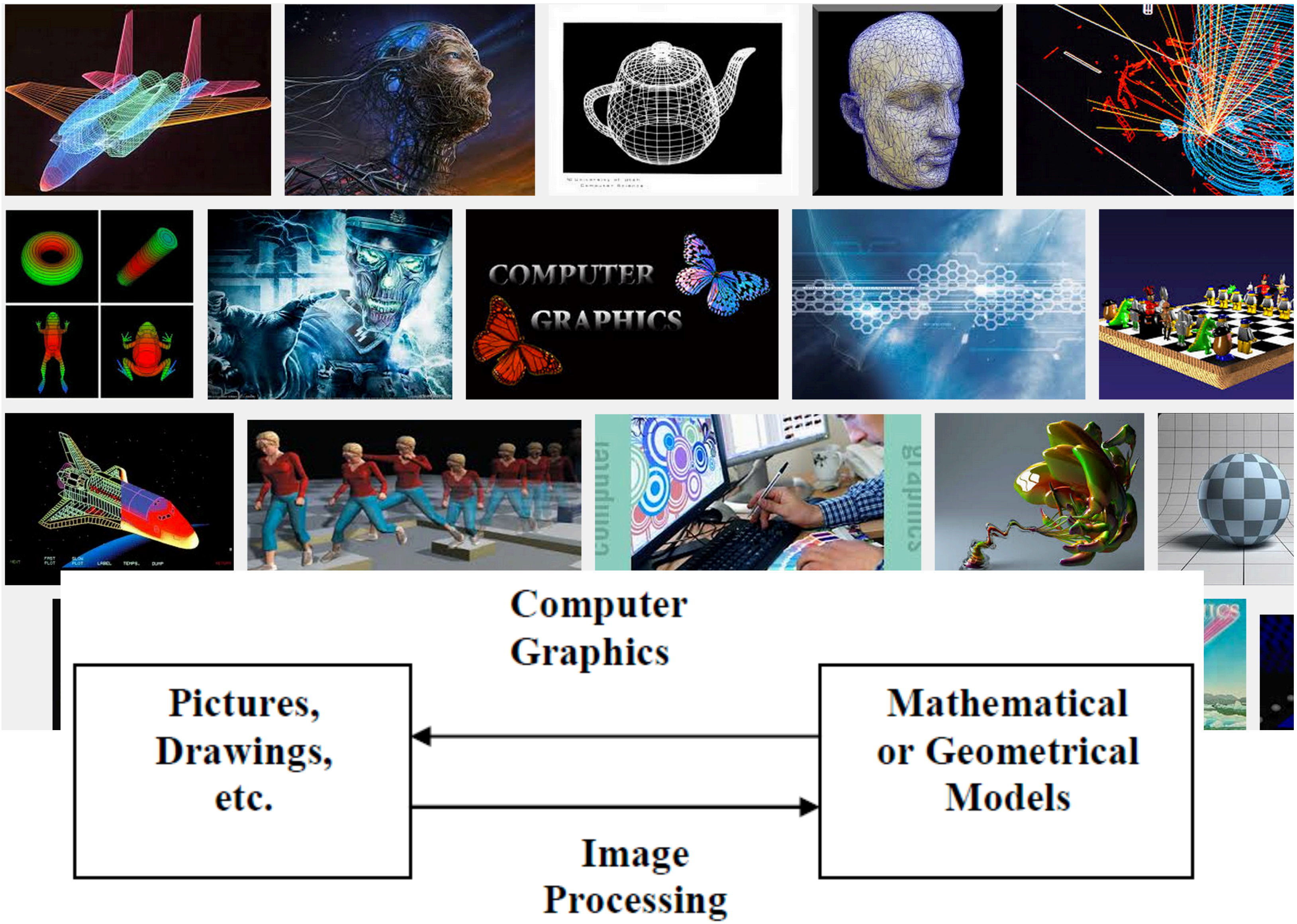
CSCE 569 Parallel Computing

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

Manycore GPU Architectures and Programming: Outline

- ☛ Introduction
  - GPU architectures, GPGPUs, and CUDA
- GPU Execution model
- CUDA Programming model
- Working with Memory in CUDA
  - Global memory, shared and constant memory
- Streams and concurrency
- CUDA instruction intrinsic and library
- Performance, profiling, debugging, and error handling
- Directive-based high-level programming model
  - OpenACC and OpenMP

Computer Graphics



Graphics Processing Unit (GPU)

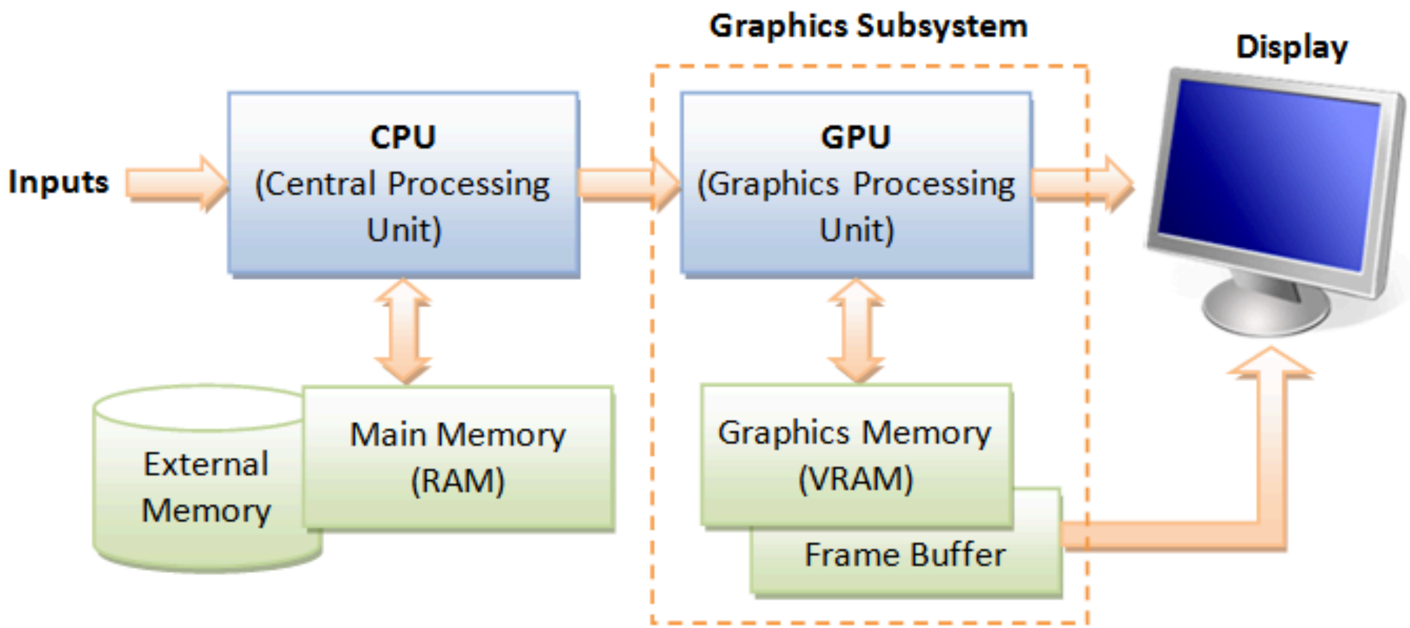
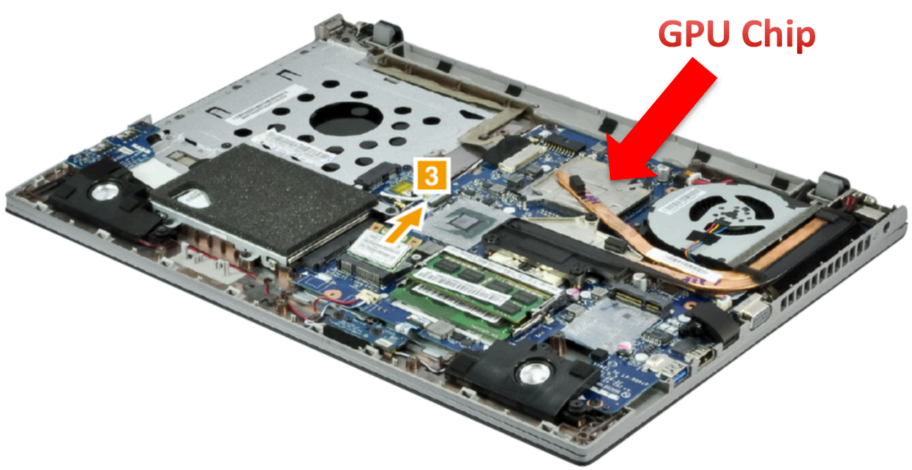
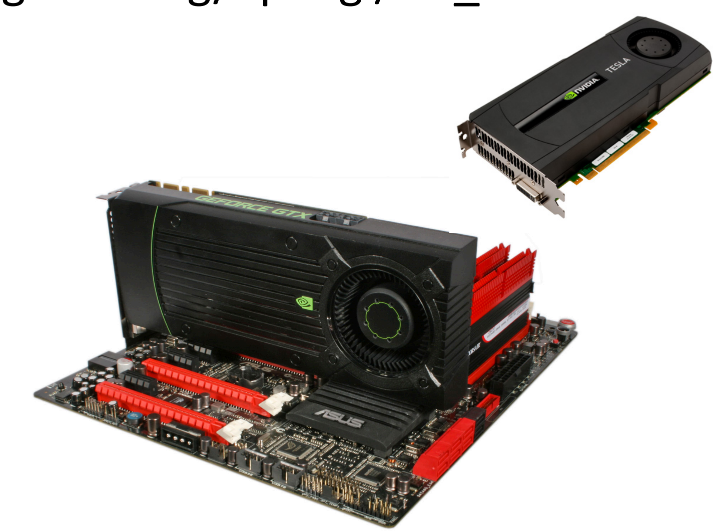


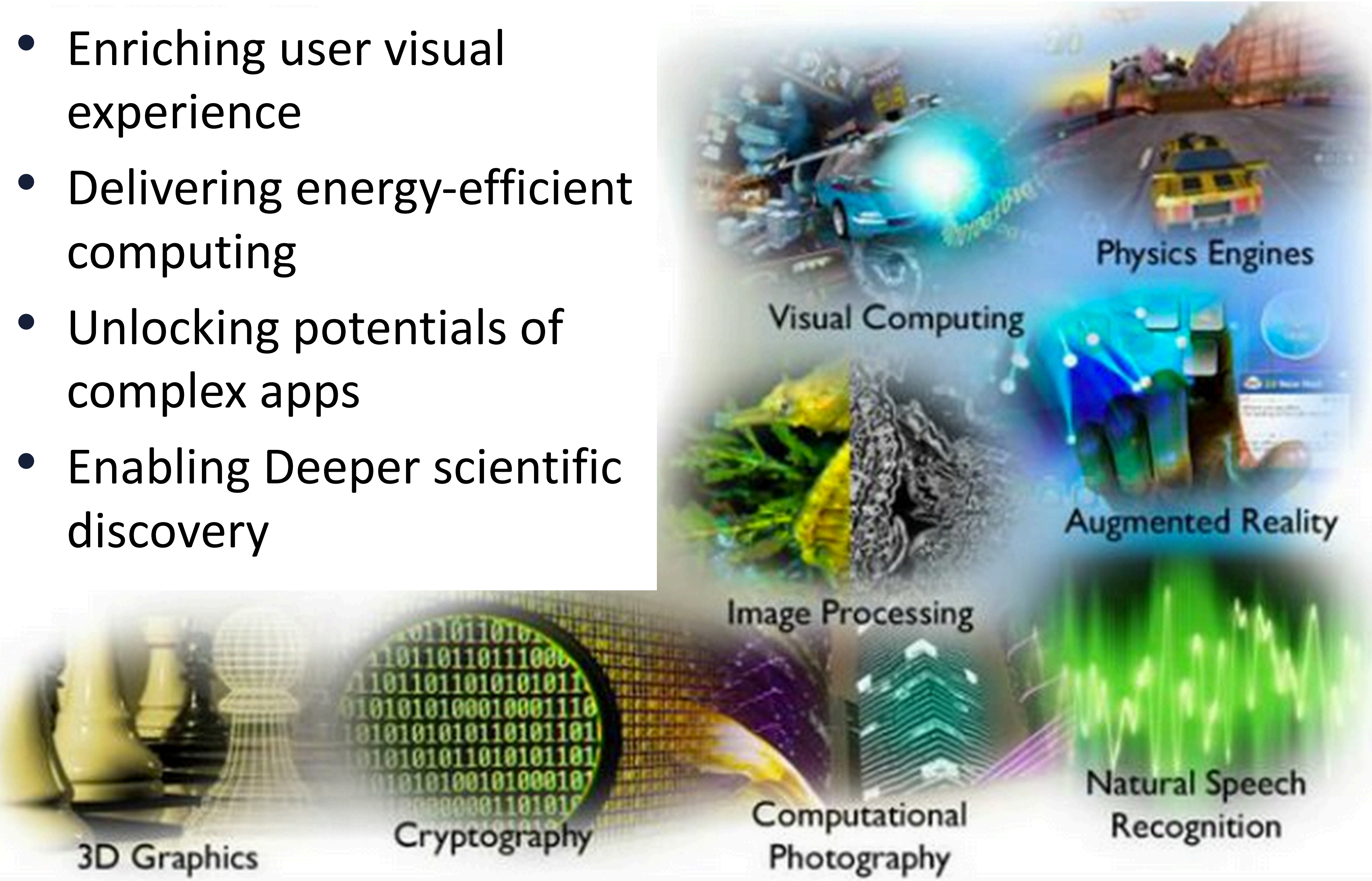
Image: http://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html





Graphics Processing Unit (GPU)

- Enriching user visual experience
- Delivering energy-efficient computing
- Unlocking potentials of complex apps
- Enabling Deeper scientific discovery

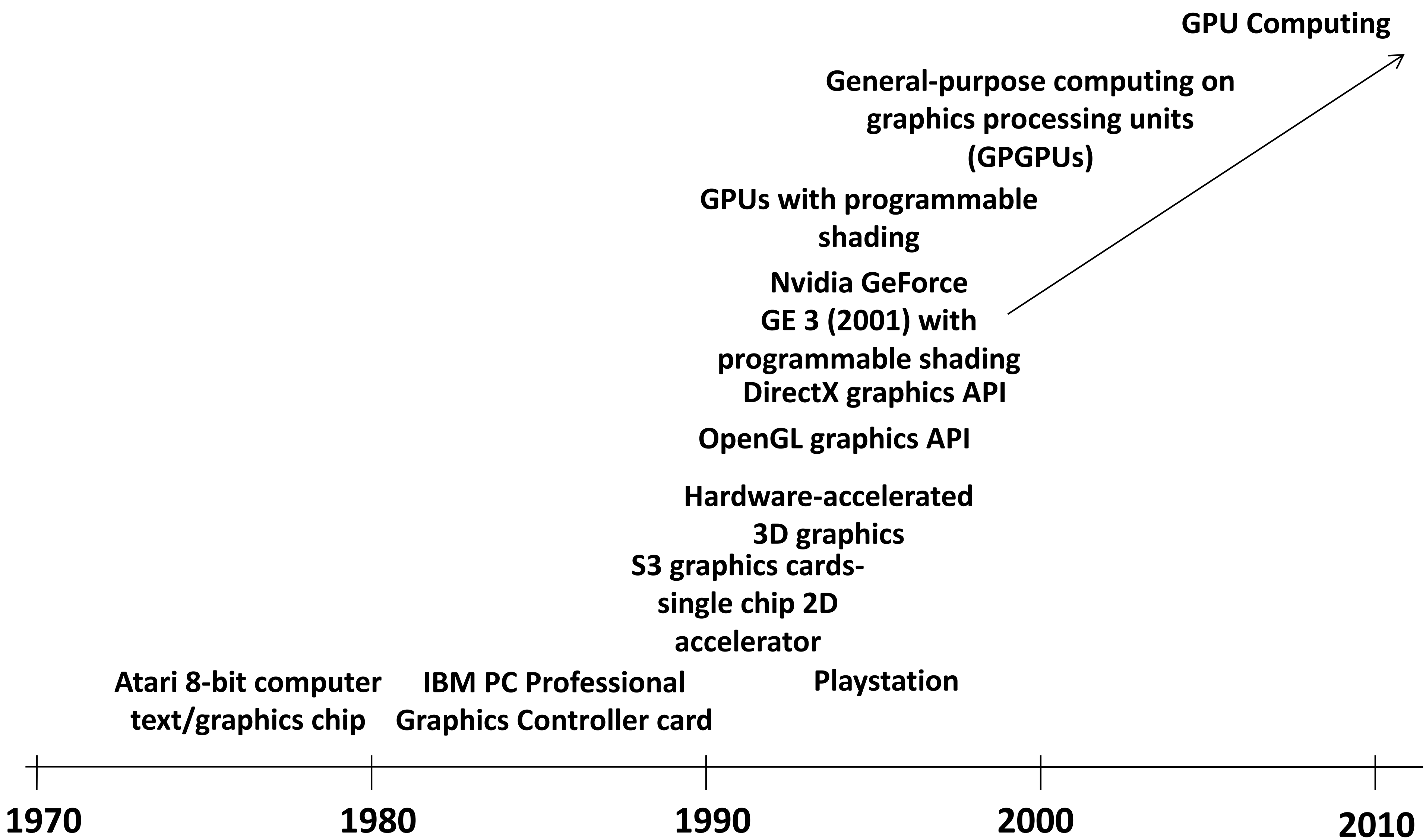


What is GPU Today?

- 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**

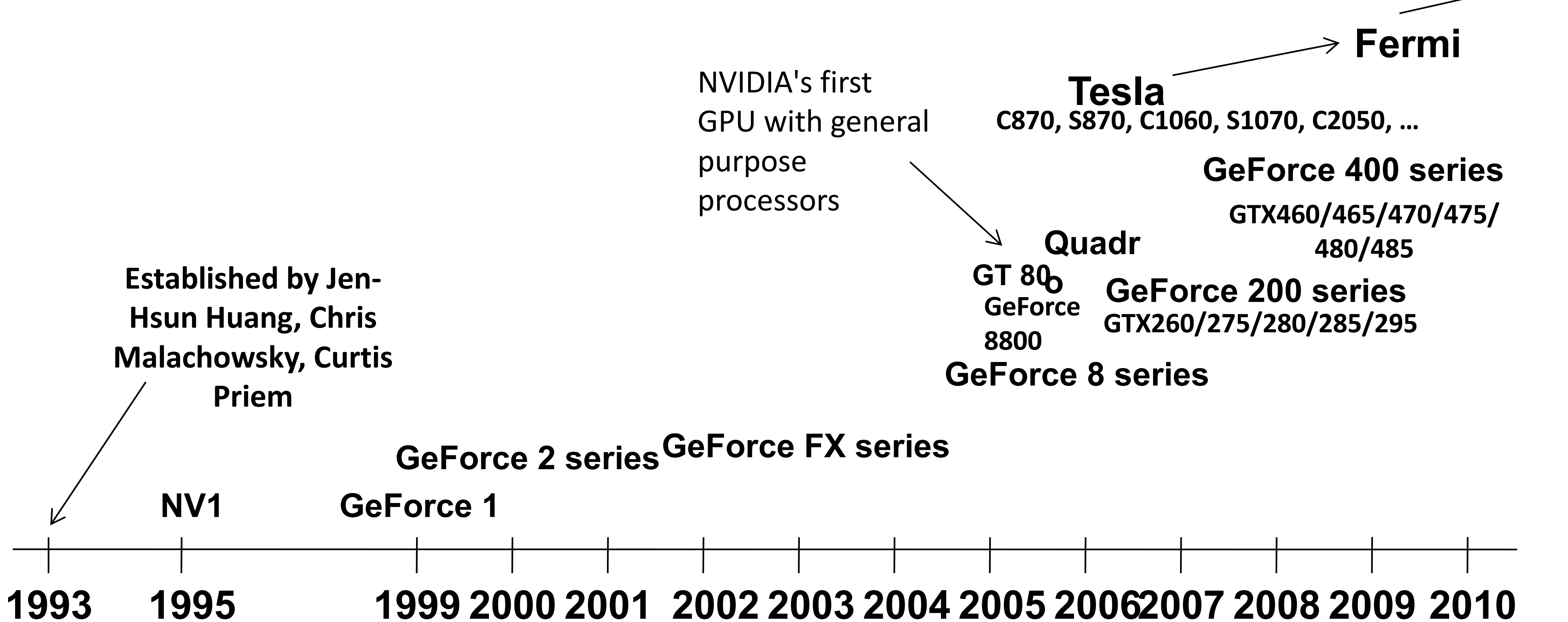
Graphics Processing Units (GPUs): Brief History



Source of information http://en.wikipedia.org/wiki/Graphics_Processing_Unit

NVIDIA Products

- NVIDIA Corp. is the leader in GPUs for HPC
- We will concentrate on NVIDIA GPU
  - Others AMD, ARM, etc

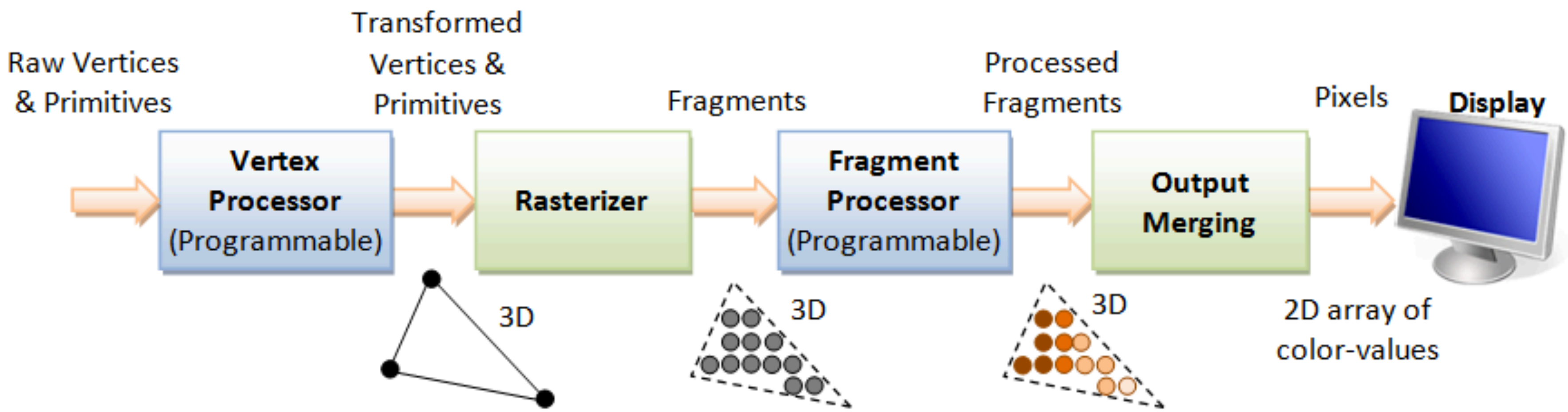


http://en.wikipedia.org/wiki/GeForce

GPU Architecture Revolution

- **Unified Scalar Shader Architecture**

- **Highly Data Parallel Stream Processing**



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.

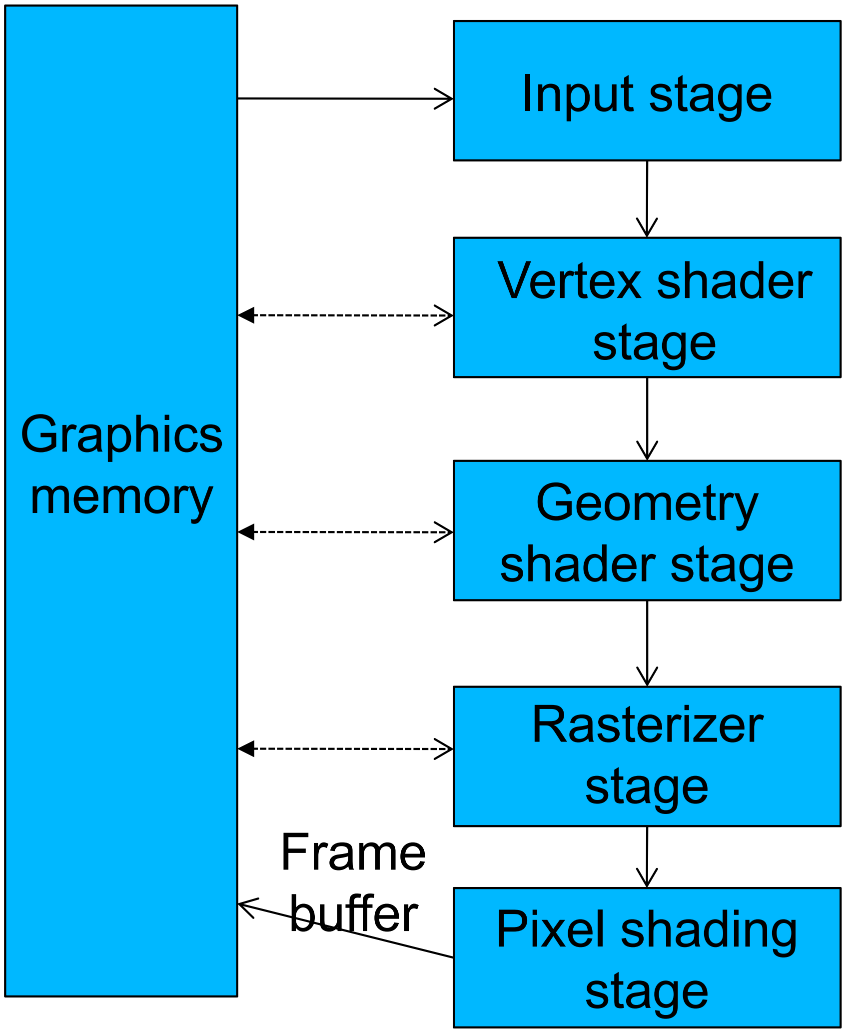
Image: http://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

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

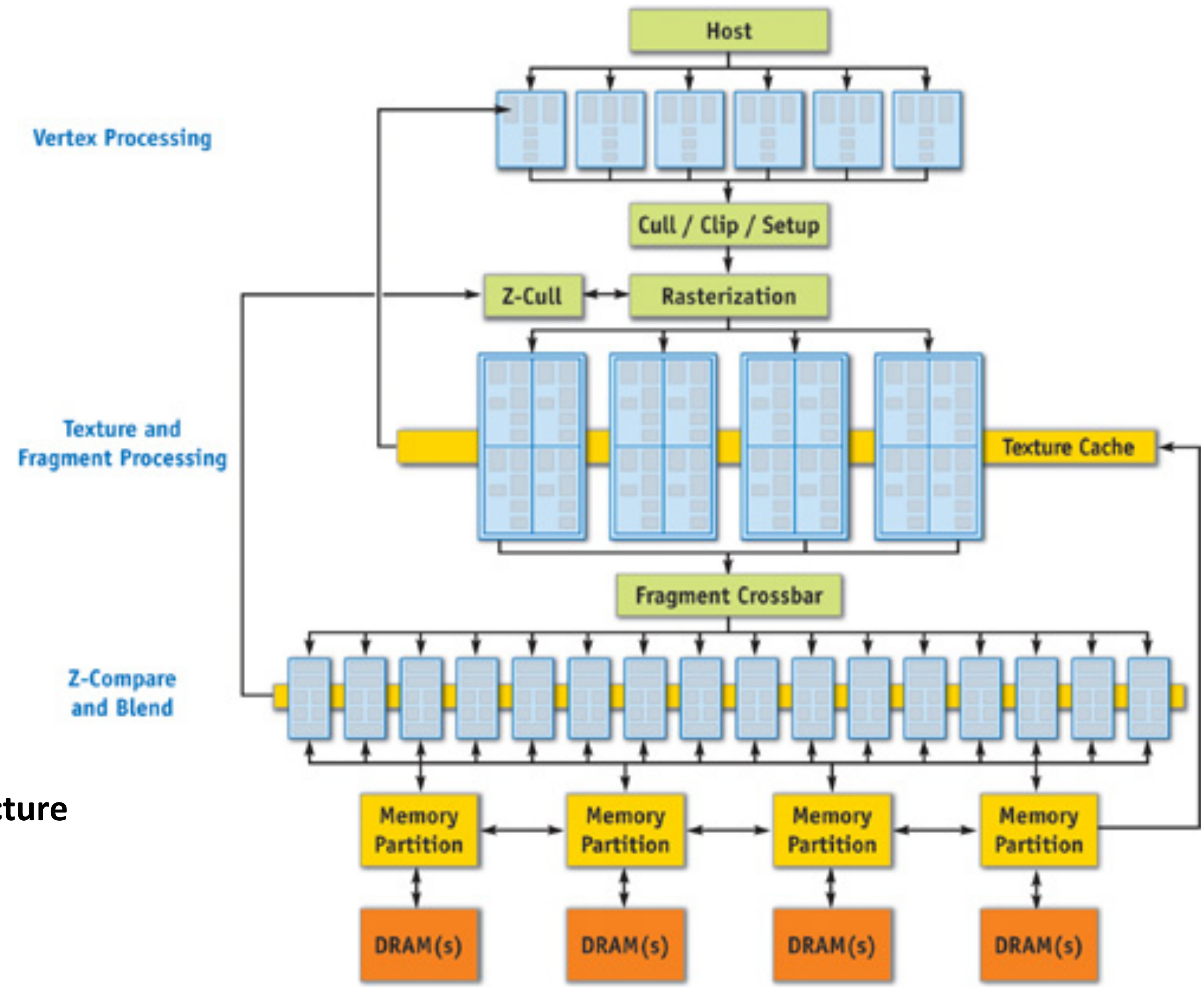
GPUs with Dedicated Pipelines -- late 1990s-early 2000s

- Graphics chips generally had a pipeline structure with individual stages performing specialized operations, finally leading to loading frame buffer for display.

- Individual stages may have access to graphics memory for storing intermediate computed data.



Specialized Pipeline Architecture

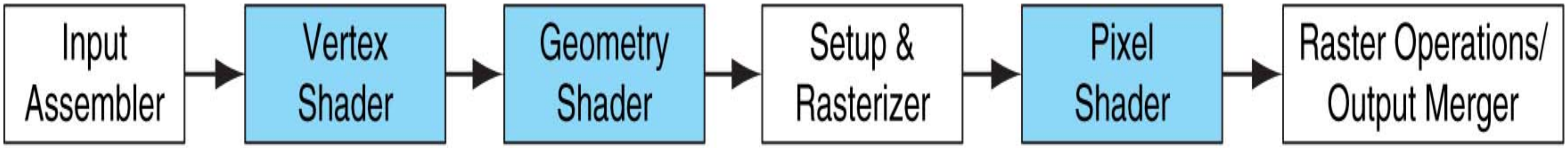


GeForce 6 Series Architecture

(2004-5)

From GPU Gems 2

Graphics Logical Pipeline



Graphics logical pipeline. Programmable graphics shader stages are blue, and fixed-function blocks are white.

Processor Per Function, each could be vector

Unbalanced and inefficient utilization

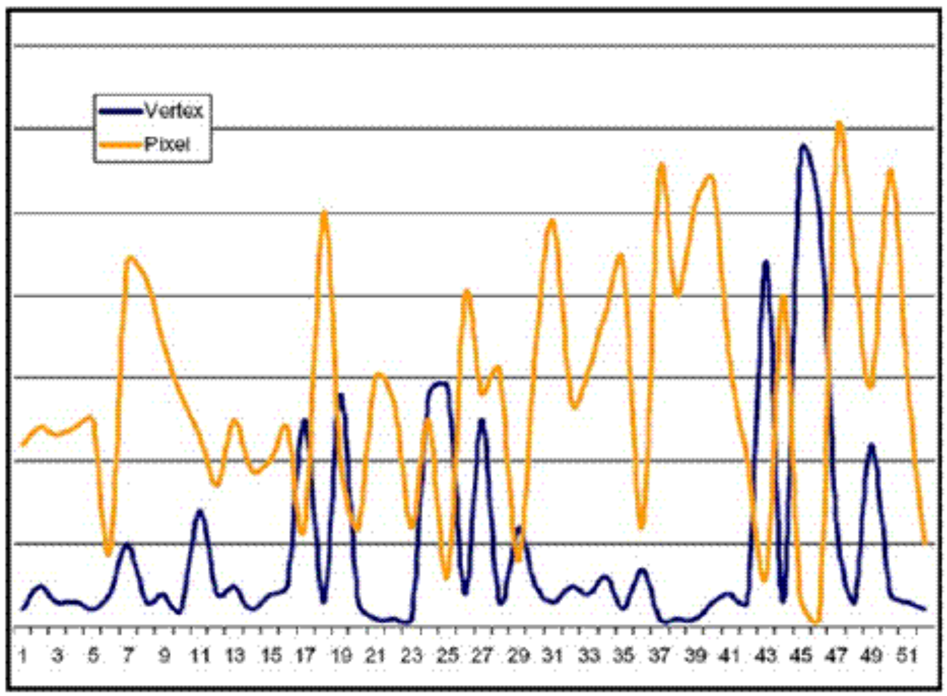


Figure 14. Characteristic pixel and vertex shader workload variation over time

Unified Shader

- Optimal utilization in unified architecture

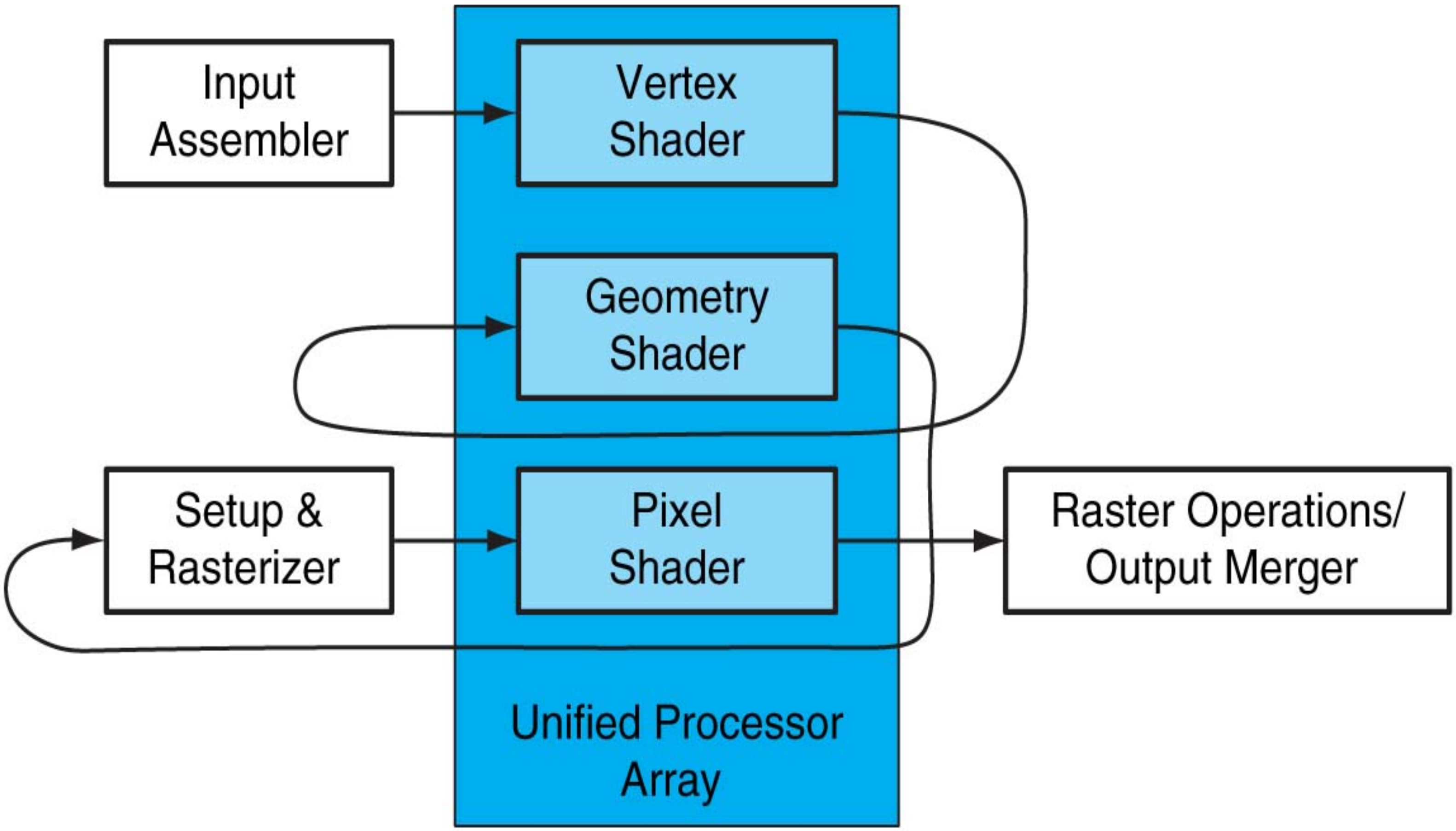


FIGURE A.2.4 Logical pipeline mapped to physical processors. The programmable shader stages execute on the array of unified processors, and the logical graphics pipeline dataflow recirculates through the processors.

Unified Shader Architecture

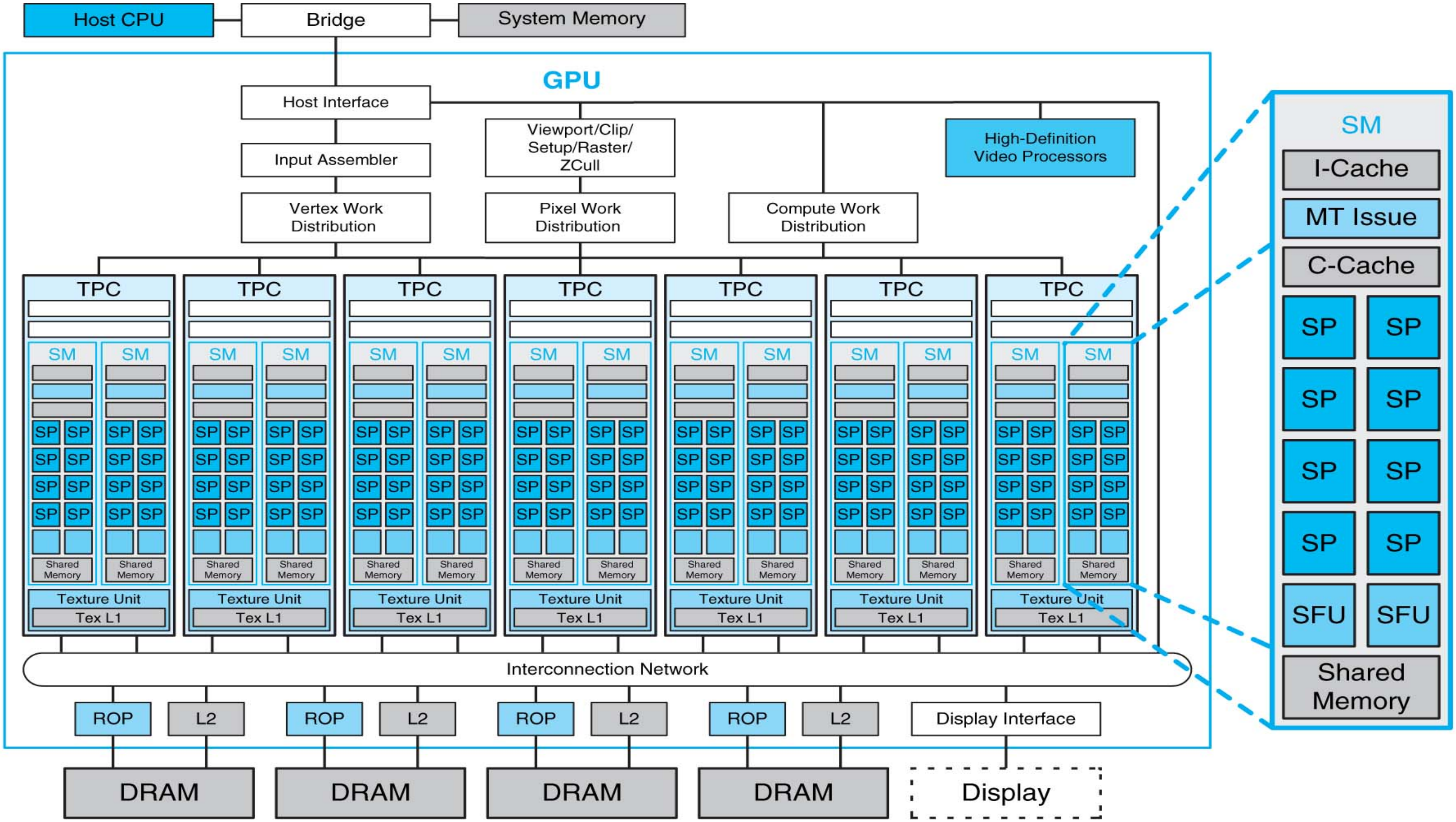


FIGURE A.2.5 Basic unified GPU architecture. Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions 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.

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

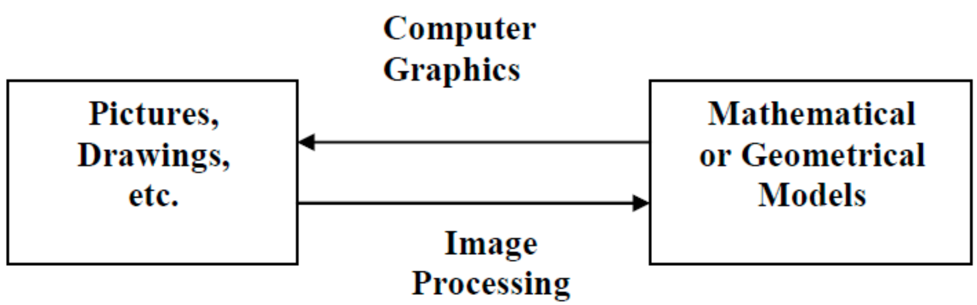
- Given a (typically large) set of data(“stream”)
- Run the same series of operations (“kernel” or “shader”) on all of the data (SIMD)

- 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

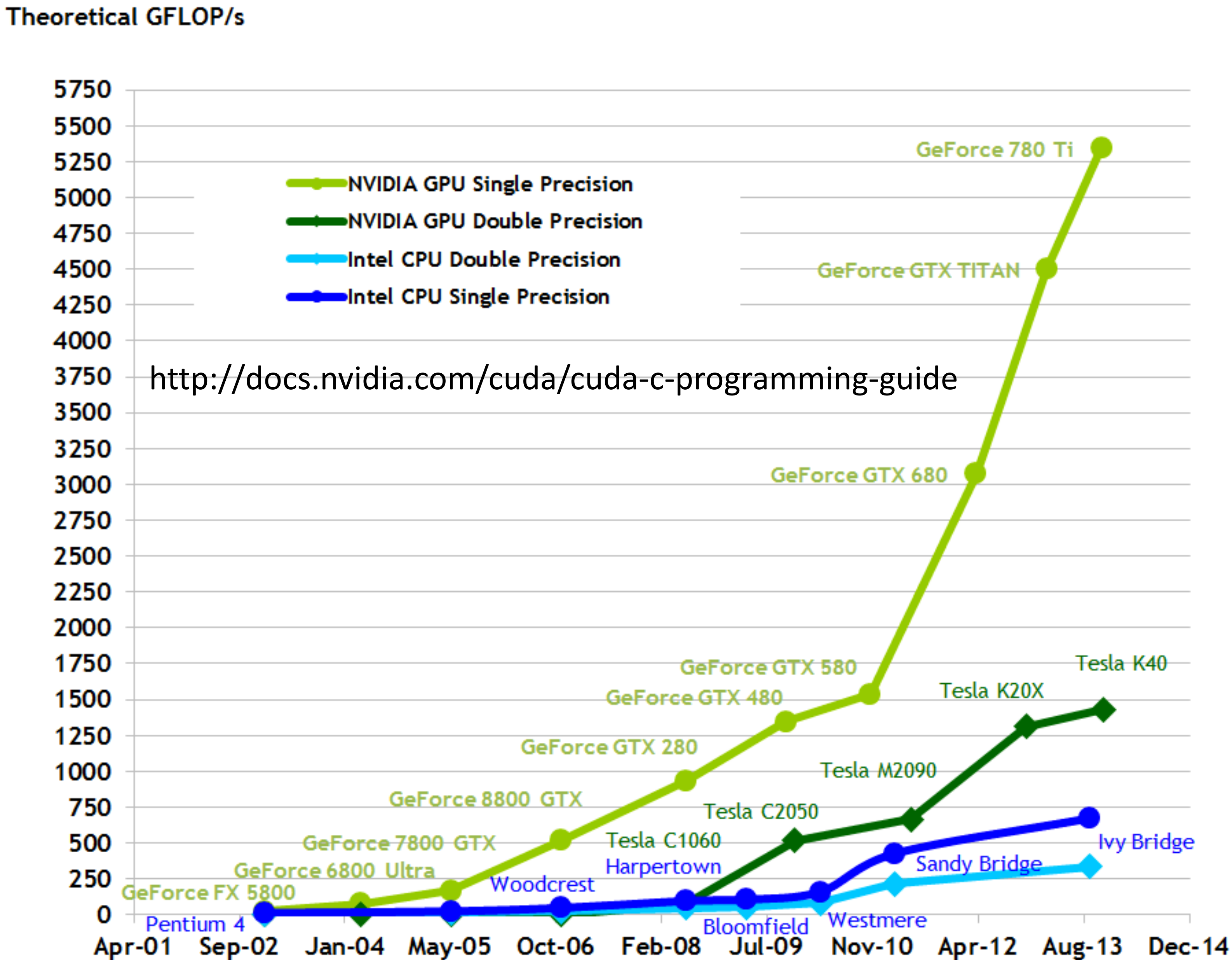
GPU Computing – The Basic Idea

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

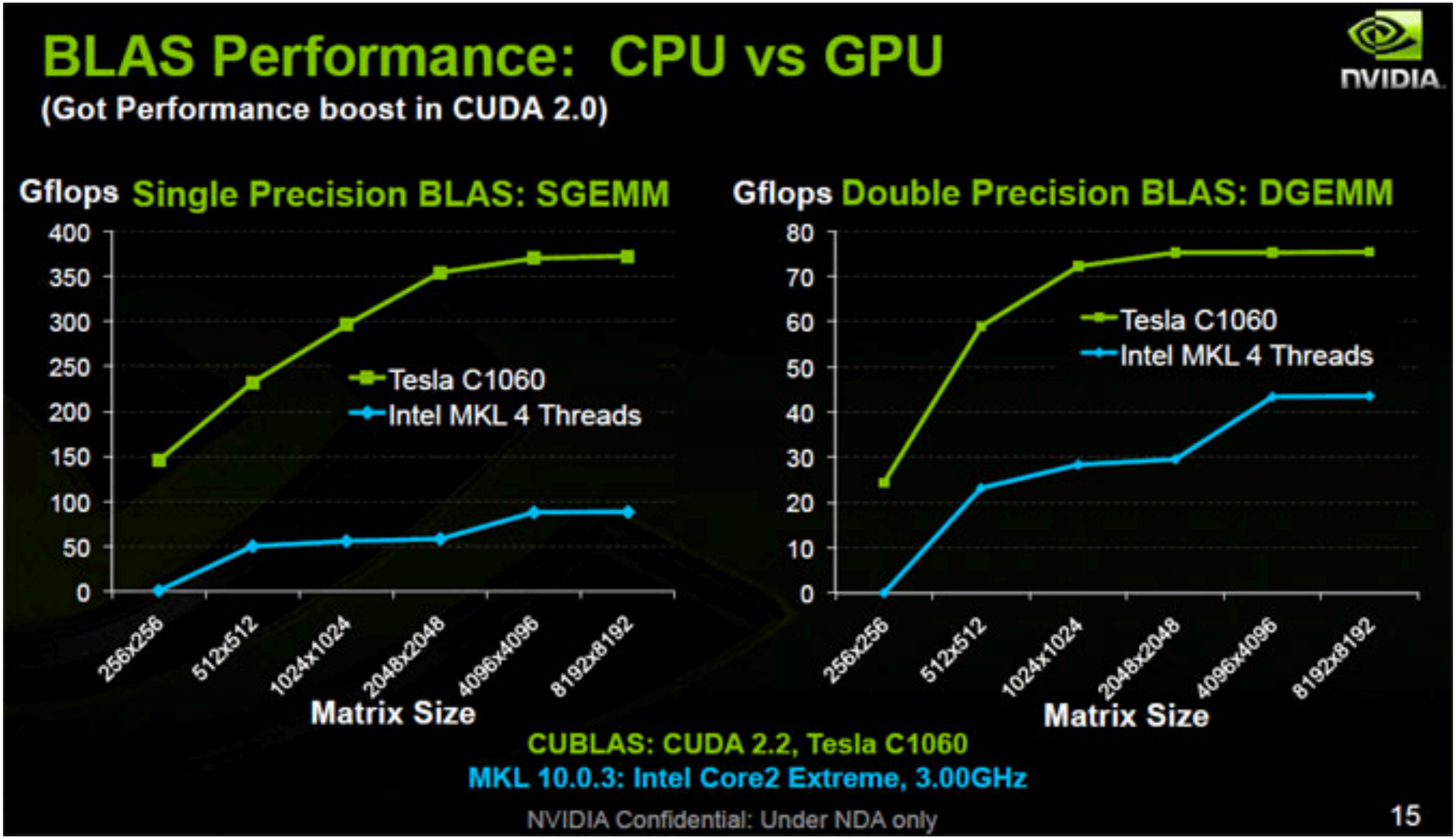
  - **The ironical fact:** It takes about 20 years (80/90s – 2007) to realize that a GPU that can do graphics well should do image processing well too.



GPU Performance Gains Over CPU



GPU Performance Gains Over CPU

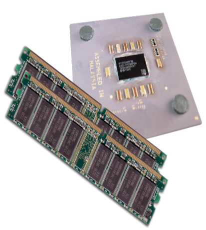


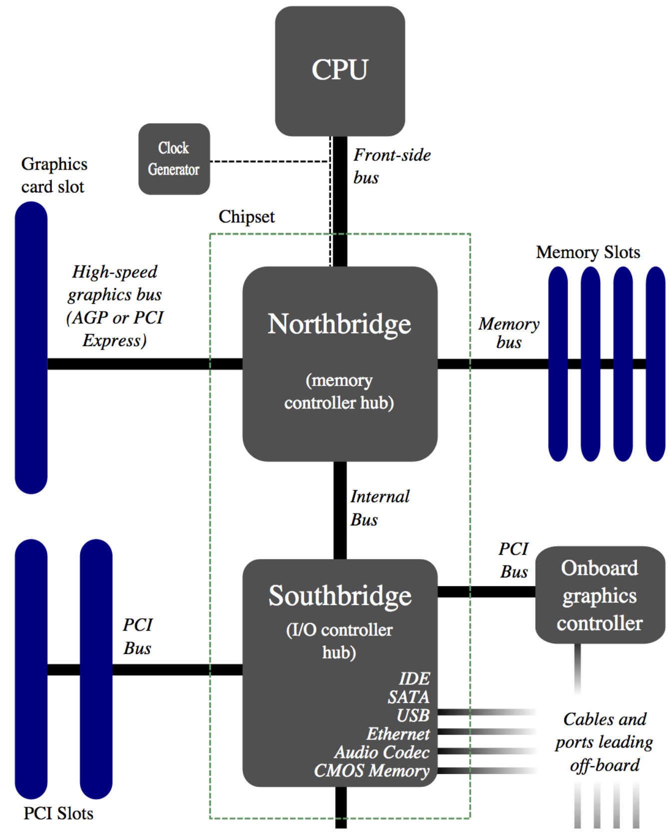
Parallelism in CPUs v. GPUs

- 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)

GPU Computing – Offloading Computation

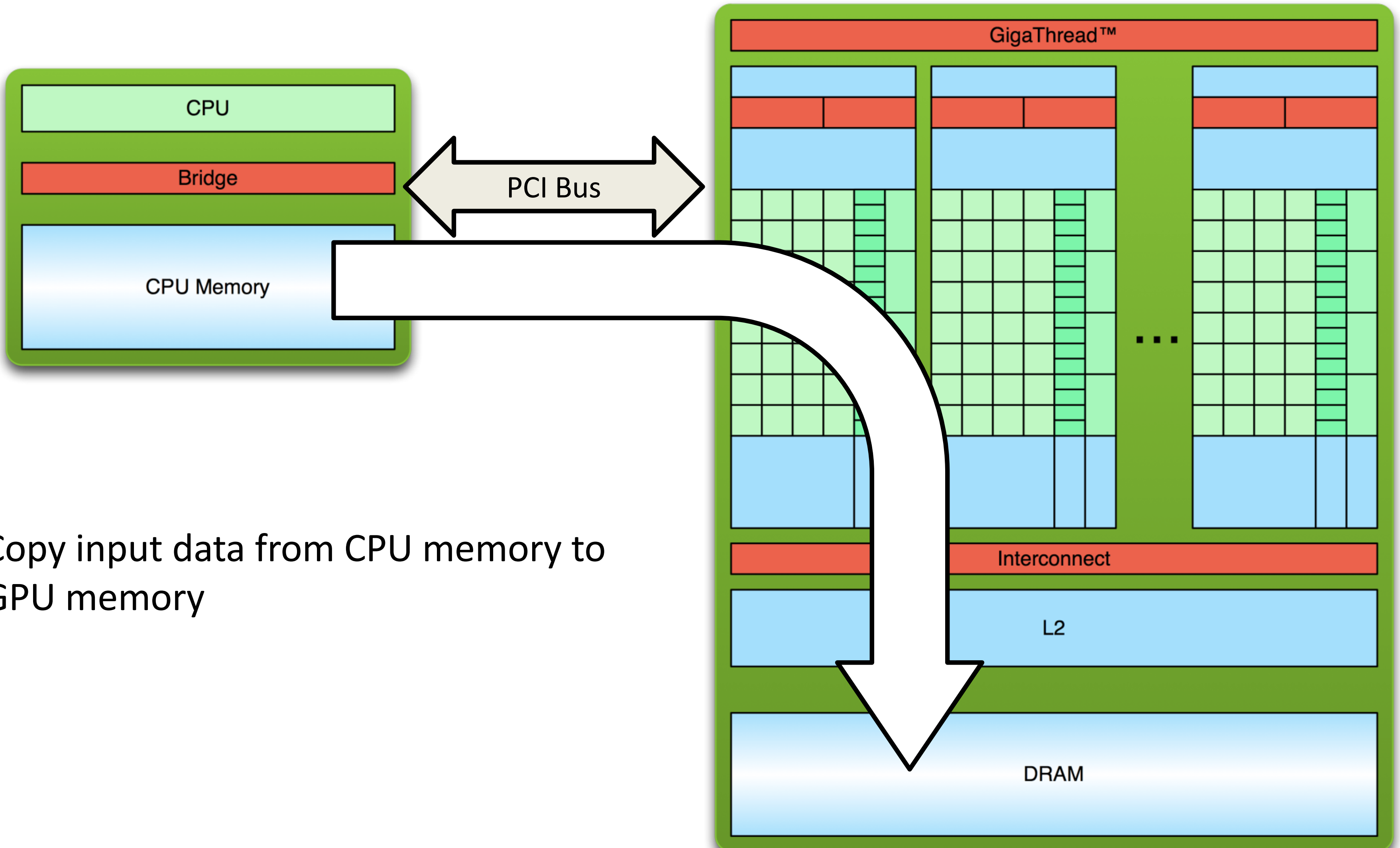
- The GPU is connected to the CPU by a reasonable fast bus (8 GB/s is typical today): PCIe





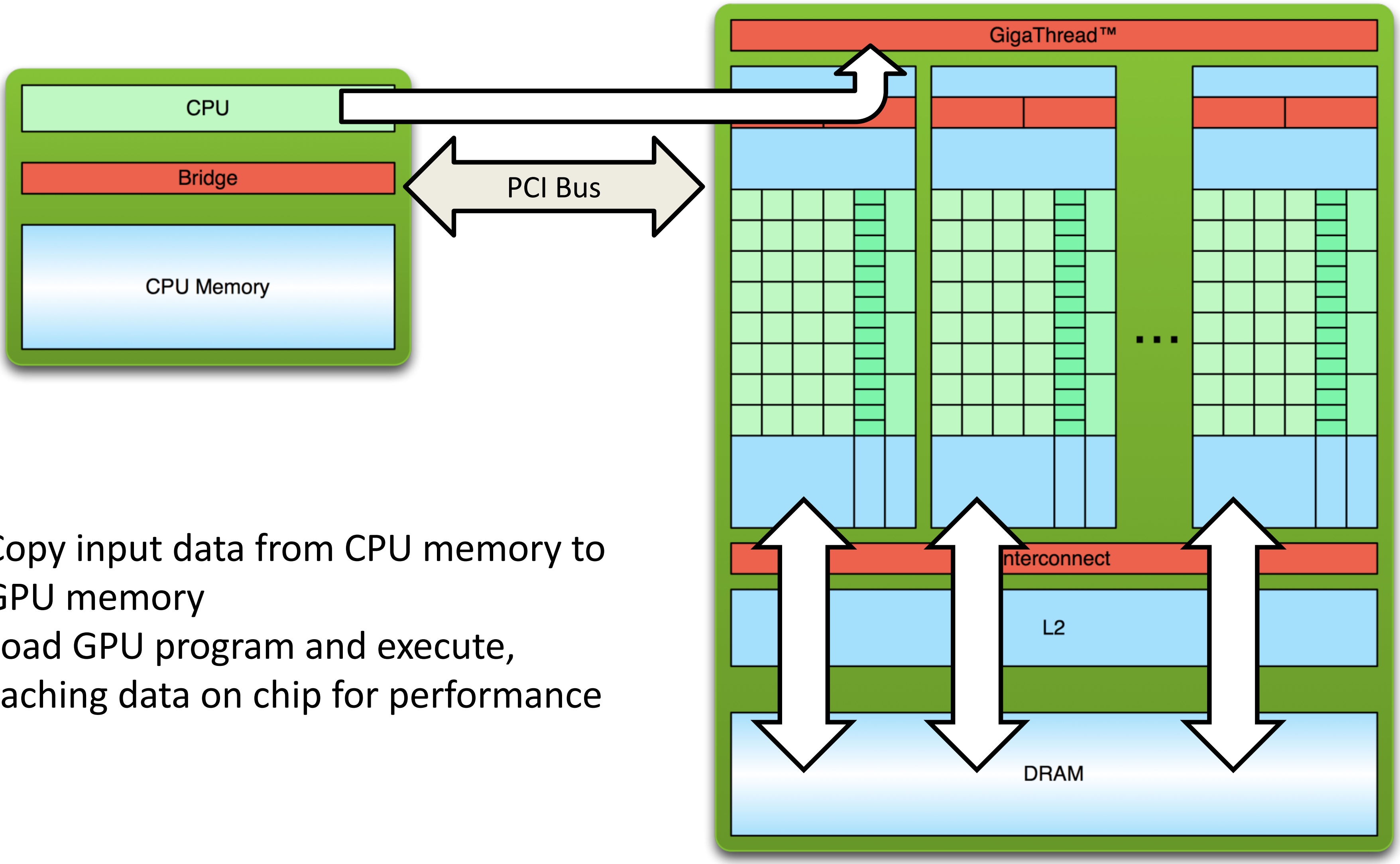
- Terminology
  – **Host**: The CPU and its memory (host memory)
  – **Device**: The GPU and its memory (device memory)

Simple Processing Flow



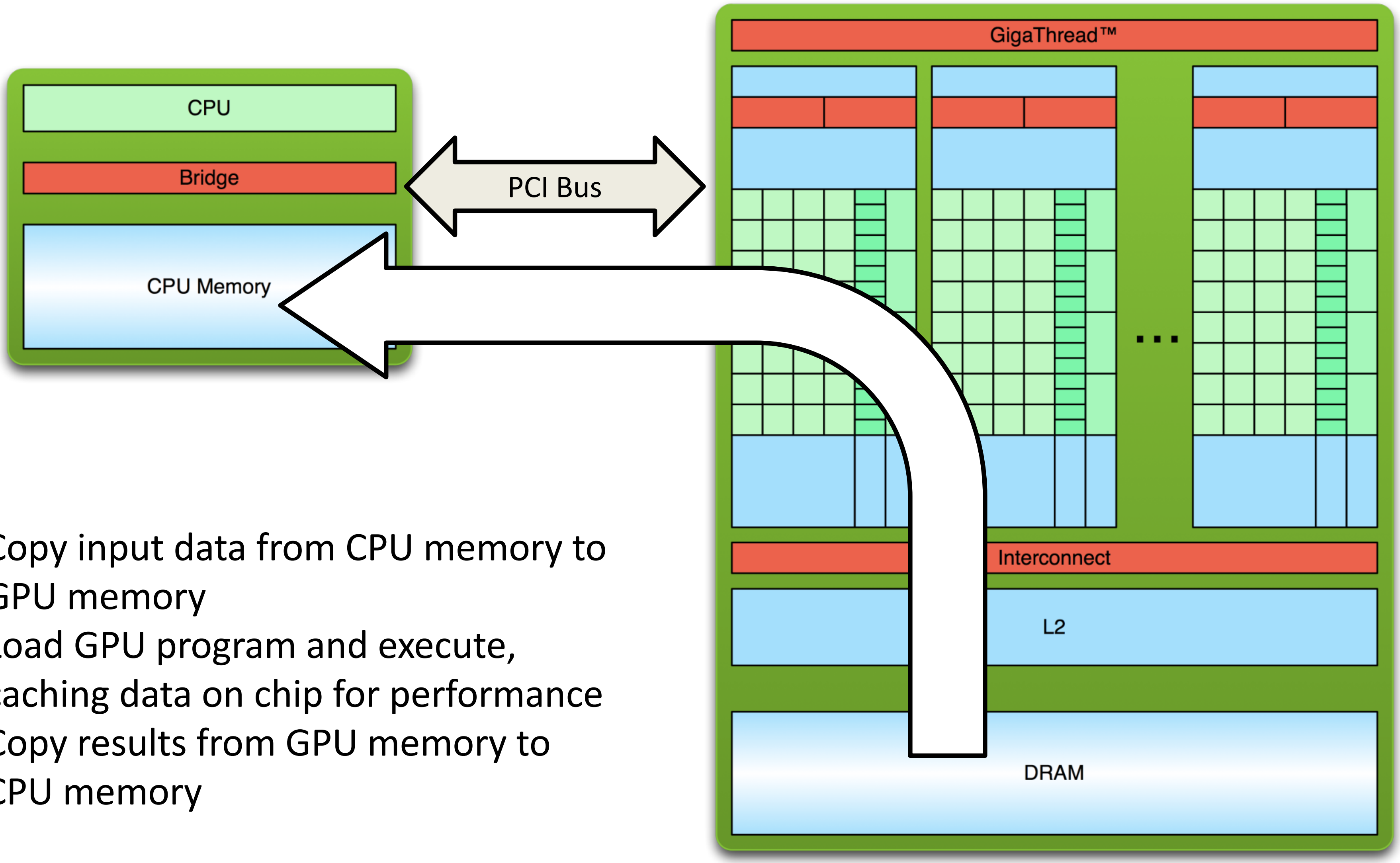
1. Copy input data from CPU memory to GPU memory

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

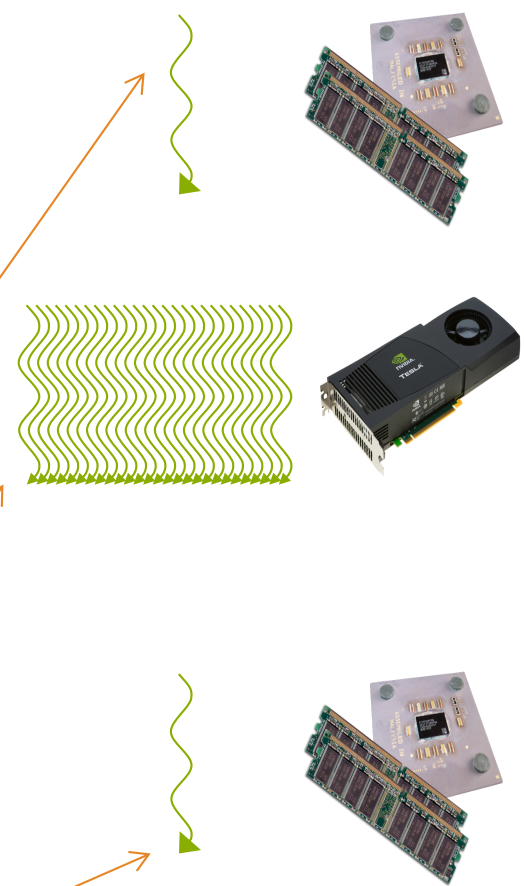
```
#define N        1024
#define RADIUS    3
#define BLOCK_SIZE 16

__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

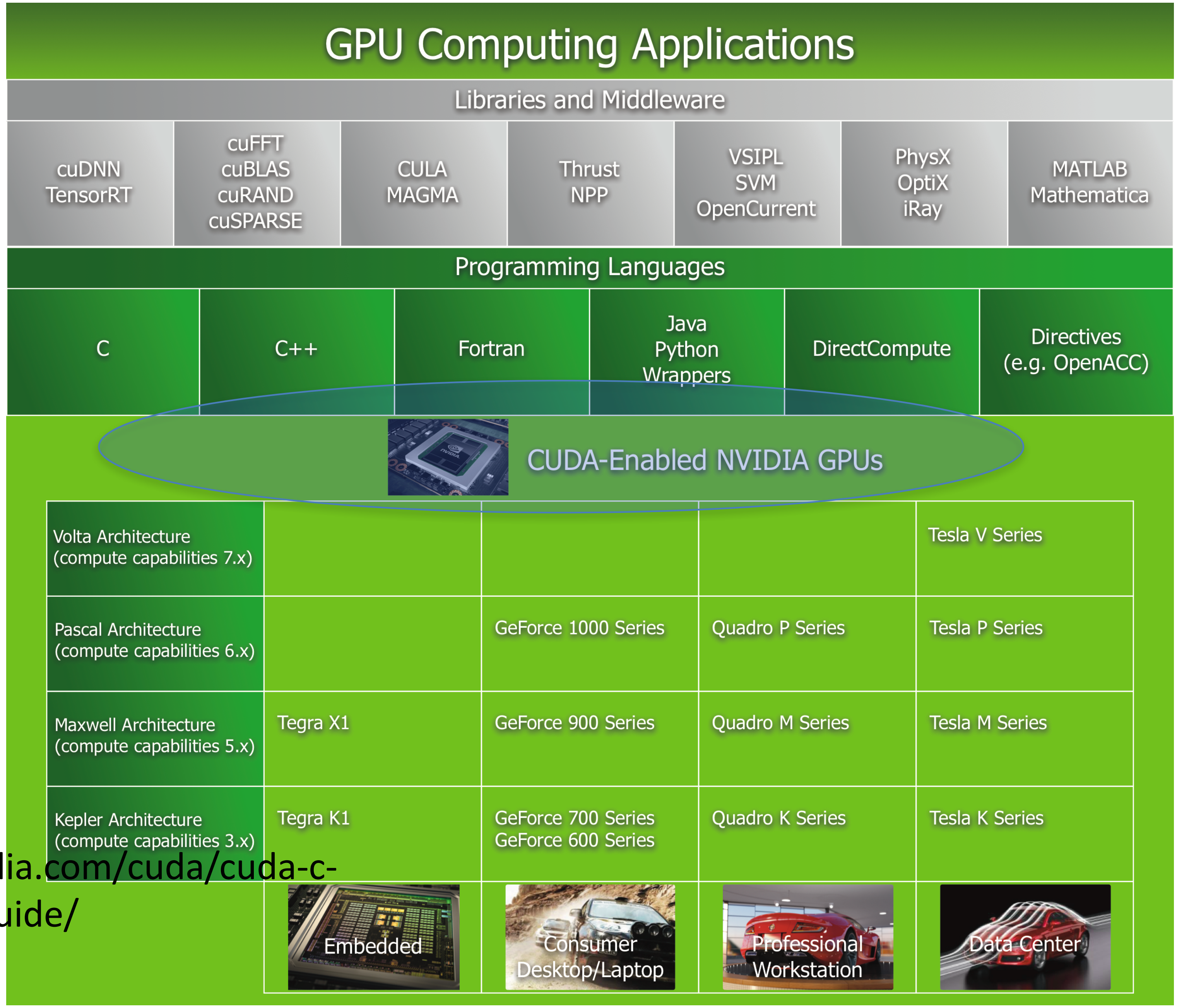
serial code

parallel exe on GPU

serial code



Programming for NVIDIA GPUs



http://docs.nvidia.com/cuda/cuda-c-programming-guide/

CUDA(Compute Unified Device Architecture)

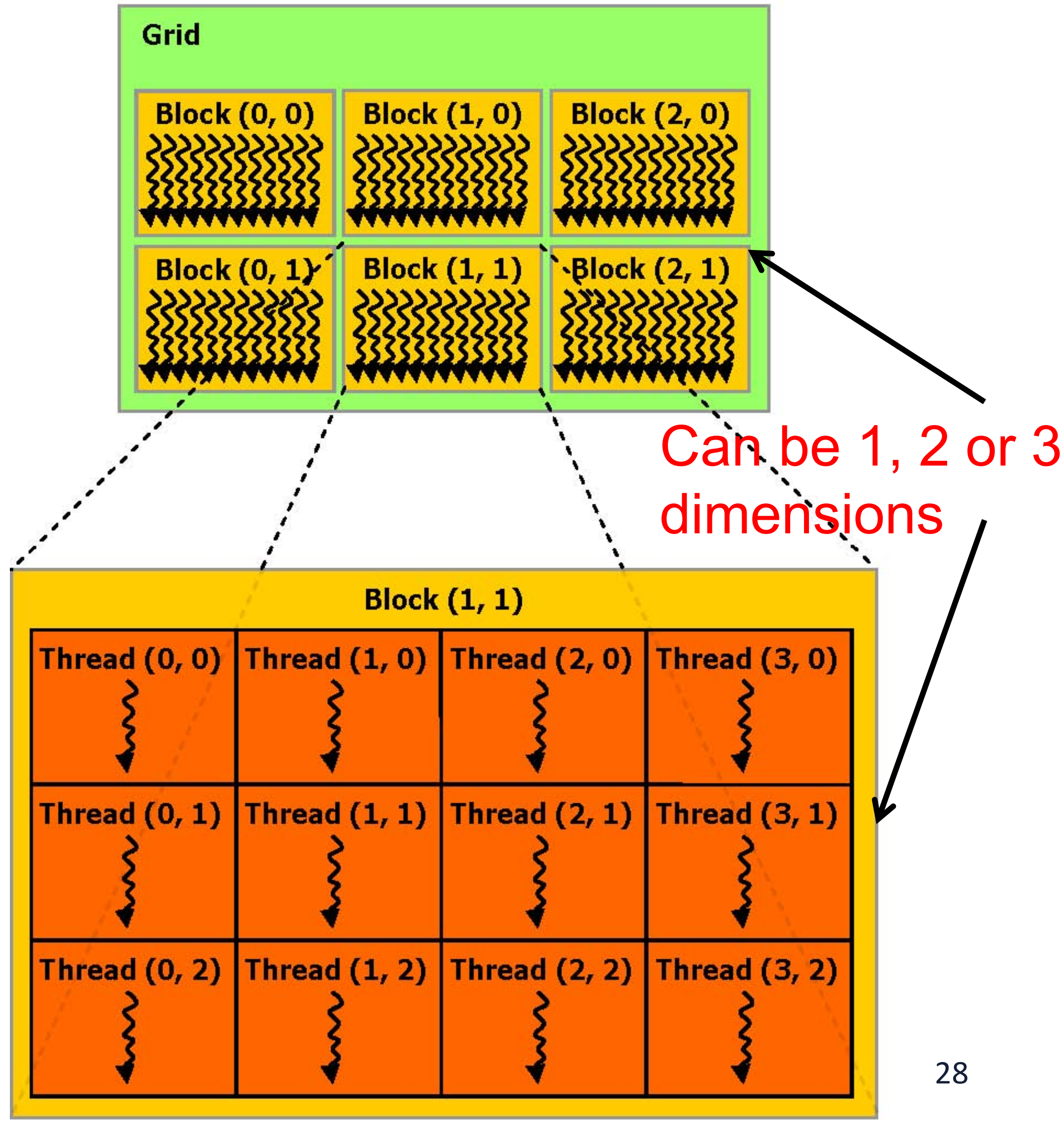
Both an *architecture* and *programming model*

- 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

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.



Hello World!

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host

- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code
  - Try on bridges, using interactive mode
  - On your computer that has NVIDIA GPU
    - You need to install CUDA SDK and NVIDIA graphics driver

Output:

```
$ nvcc
hello.cu
$ ./a.out
Hello World!
$
```

Hello World! with Device Code

```
__global__ void hellokernel() {
    printf("Hello World!\n");
}
int main(void){
    int num_threads = 1;
    int num_blocks = 1;
    hellokernel<<<num_blocks,num_threads>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

- Two new syntactic elements…

Output:

```
$ nvcc
hello.cu
$ ./a.out
Hello World!
$
```

GPU code examples and try on Bridges

- GPU code examples:
  - https://passlab.github.io/CSCE569/resources/gpu_code_examples
  - You can download by yourself or copy from my home folder on bridges
- Bridge instruction:
  - https://passlab.github.io/CSCE569/resources/HardwareSoftware.html#interactive

- Bridges:
  - interact -gpu
  - module load gcc/5.3.0 cuda/8.0 opencv/3.2.0
  - cp -r ~yan/gpu_code_examples ~

  - cd gpu_code_examples
  - nvcc hello-1.cu –o hello-1
  - ./hello-1
  - nvcc hello-2.cu –o hello-2
  - ./hello-2

Hello World! with Device Code

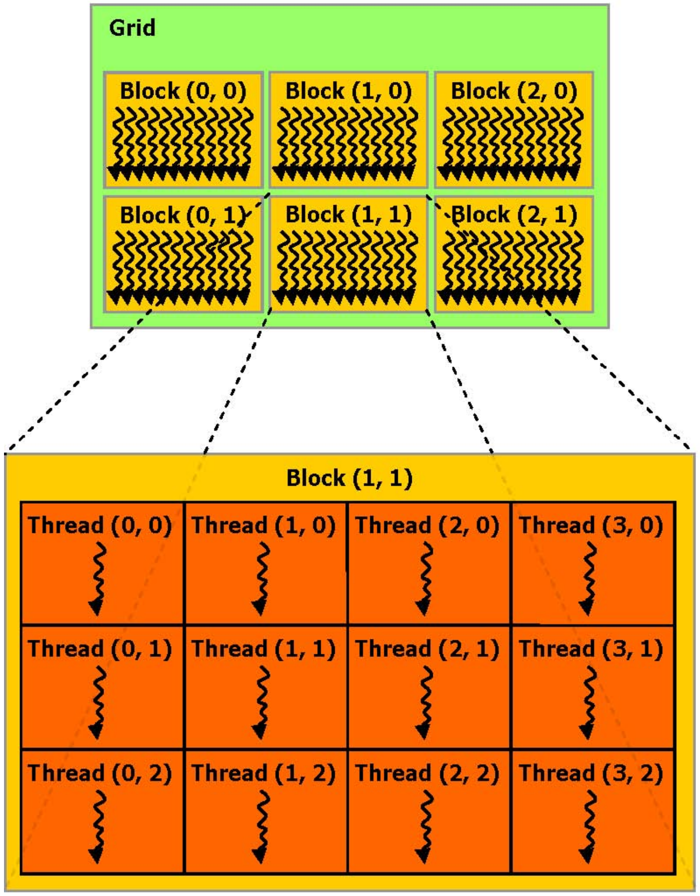
```
__global__ void hellokernel(void)
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `hellokernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`

Hello World! with Device COde

```
hellokernel<<<num_blocks,num_threads>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - <<< ... >>> parameters are for thread dimensionality
- That’s all that is required to execute a function on the GPU!



Hello World! with Device Code

```
__device__ const char *STR = "Hello World!";
const char STR_LENGTH = 12;


__global__ void hellokernel(){
  printf("%c", STR[threadIdx.x % STR_LENGTH]);
}
int main(void){
  int num_threads = STR_LENGTH;
  int num_blocks = 1;
  hellokernel<<<num_blocks,num_threads>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Output:

```
$ nvcc
hello.cu
$ ./a.out
Hello World!
$
```

Hello World! with Device Code

```
__device__ const char *STR = "Hello World!";
const char STR_LENGTH = 12;

__global__ void hellokernel(){
  printf("%c", STR[threadIdx.x % STR_LENGTH]);
}
int main(void){
  int num_threads = STR_LENGTH;
  int num_blocks = 2;
  hellokernel<<<num_blocks,num_threads>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

`__device__`: Identify device-only data

`threadIdx.x`: the thread ID

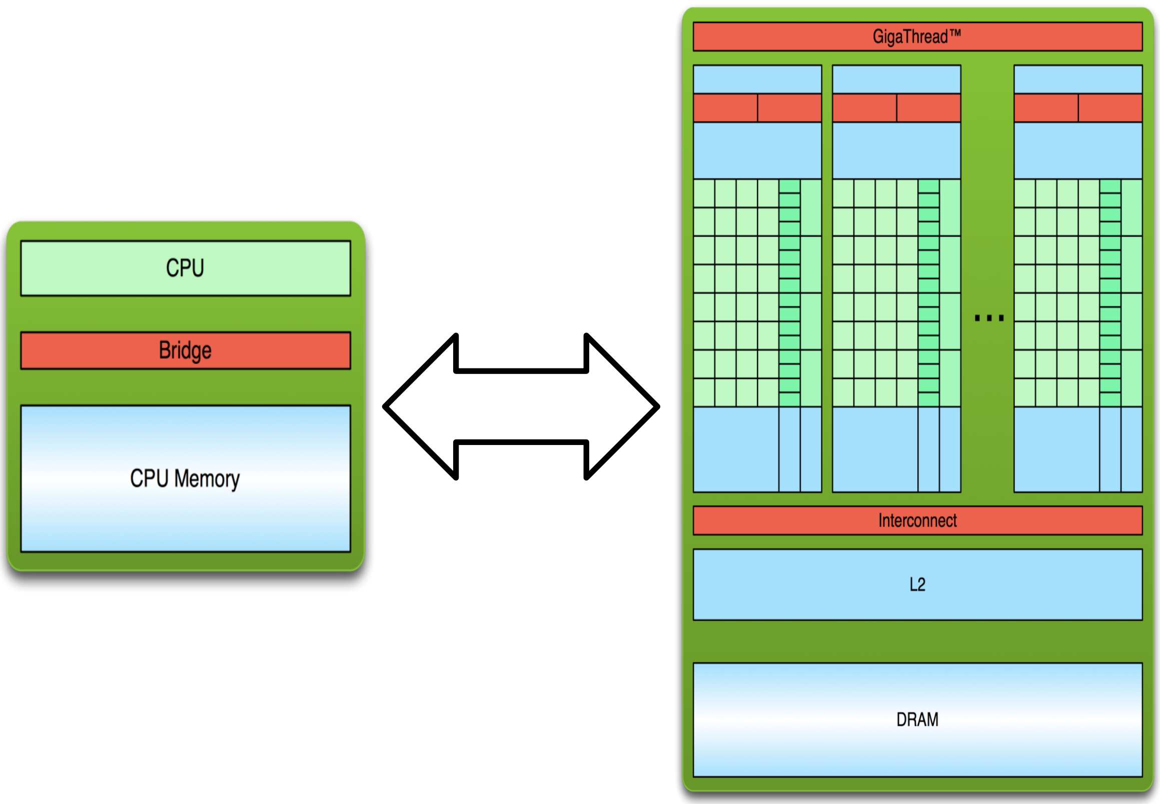
Each thread only prints one character

Manycore GPU Architectures and Programming

- GPU architectures, graphics and GPGPUs
- ☛ GPU Execution model
- CUDA Programming model
- Working with Memory in CUDA
  - Global memory, shared and constant memory
- Streams and concurrency
- CUDA instruction intrinsic and library
- Performance, profiling, debugging, and error handling
- Directive-based high-level programming model
  - OpenACC and OpenMP

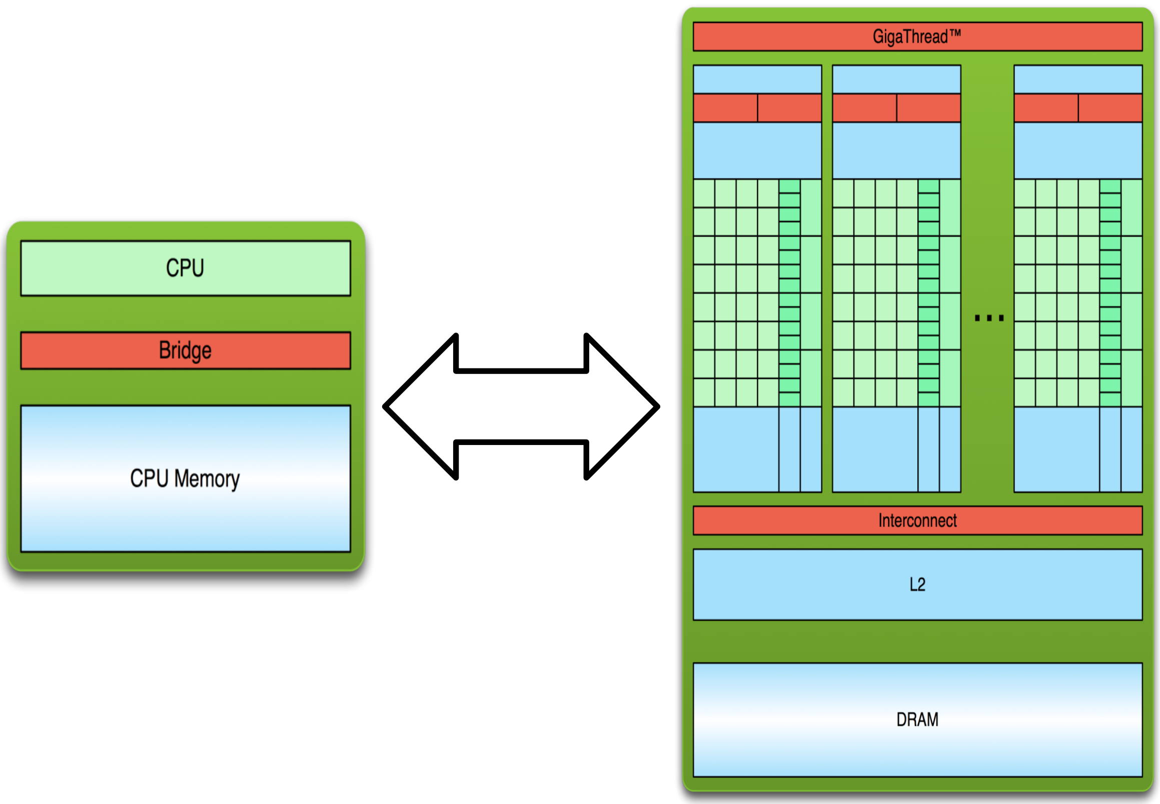
GPU Execution Model

- 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 Execution Model

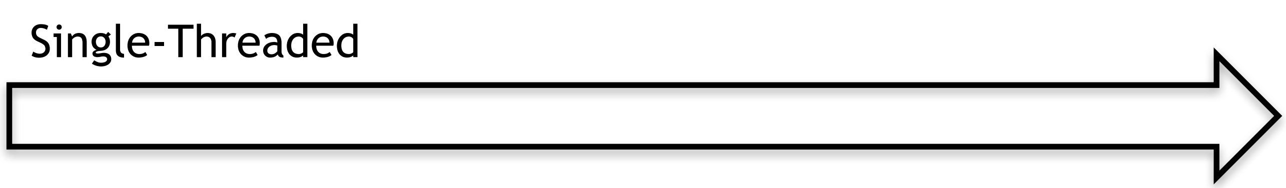
- The GPU is a physically separate processor from the CPU
  - Discrete vs. Integrated
- The GPU Execution Model offers different abstractions from the CPU to match the change in architecture



The Simplest Model: Single-Threaded

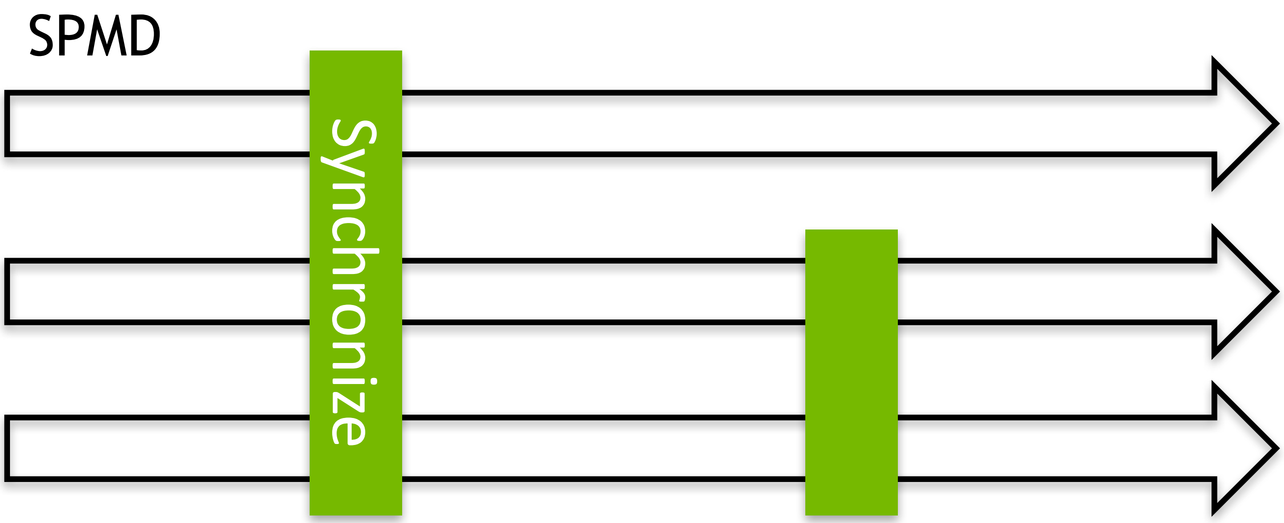
- Single-threaded Execution Model
  - Exclusive access to all variables
  - Guaranteed in-order execution of loads and stores
  - Guaranteed in-order execution of arithmetic instructions

- Also the most common execution model, and simplest for programmers to conceptualize and optimize



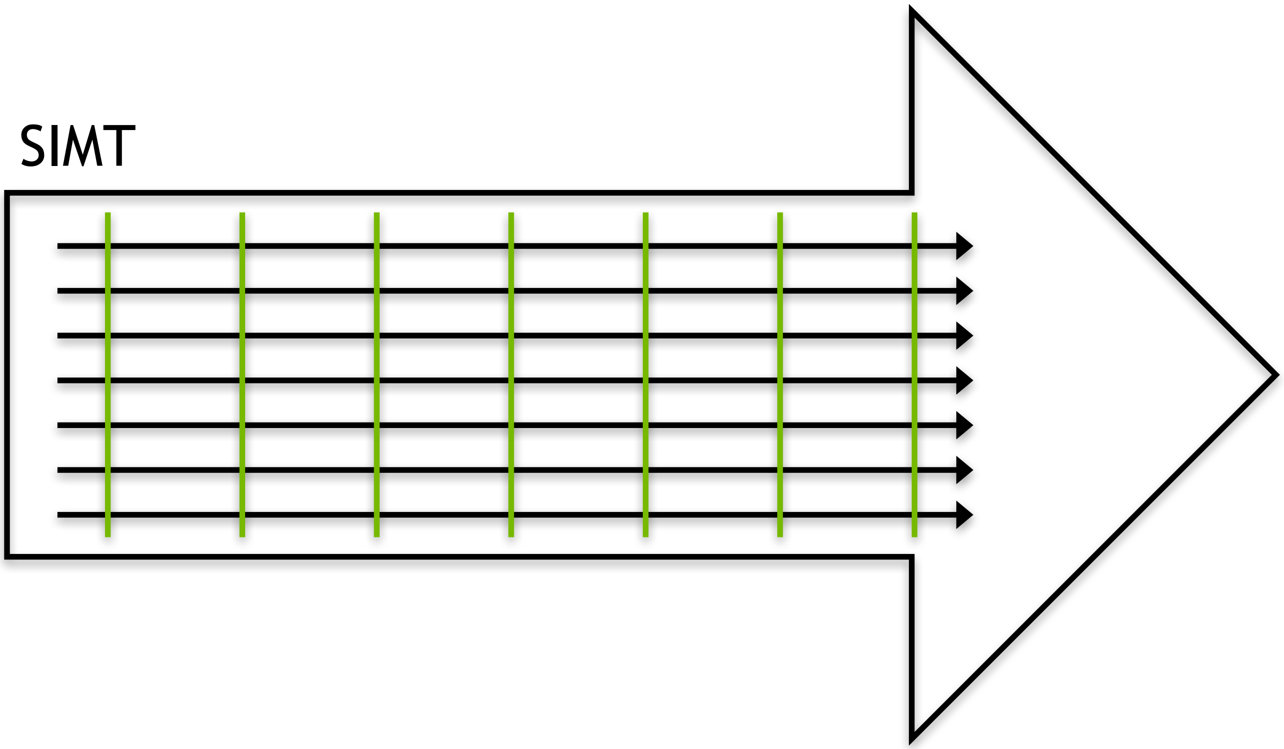
CPU SPMD Multi-Threading

- Single-Program, Multiple-Data (SPMD) model
  - Makes the same in-order guarantees within each thread
  - Says little or nothing about inter-thread behaviour or exclusive variable access without **explicit inter-thread synchronization**



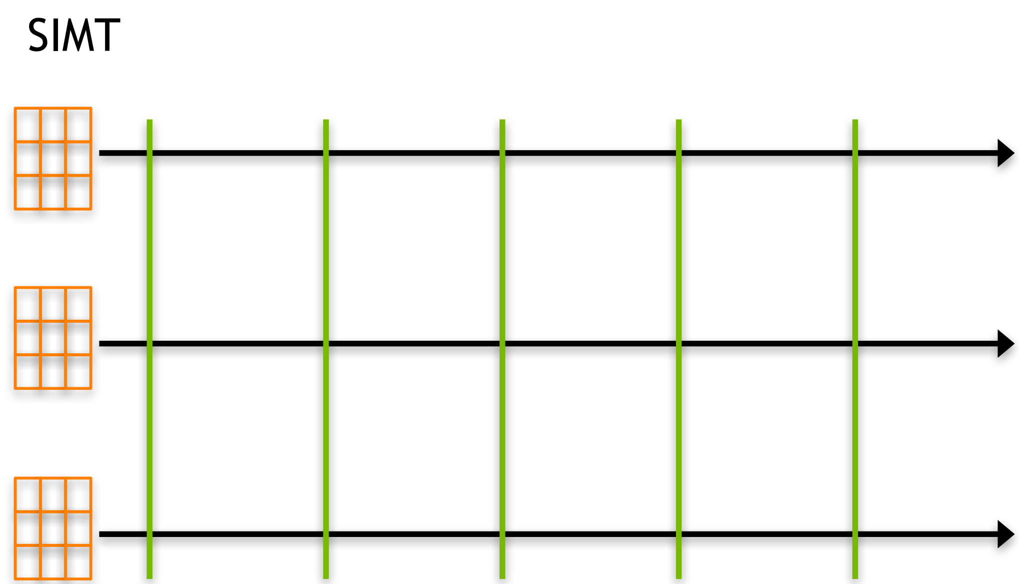
GPU Multi-Threading

- Uses the Single-Instruction, Multiple-Thread model
  - Many threads execute the same instructions in lock-step
  - Implicit synchronization after every instruction (think vector parallelism)



GPU Multi-Threading

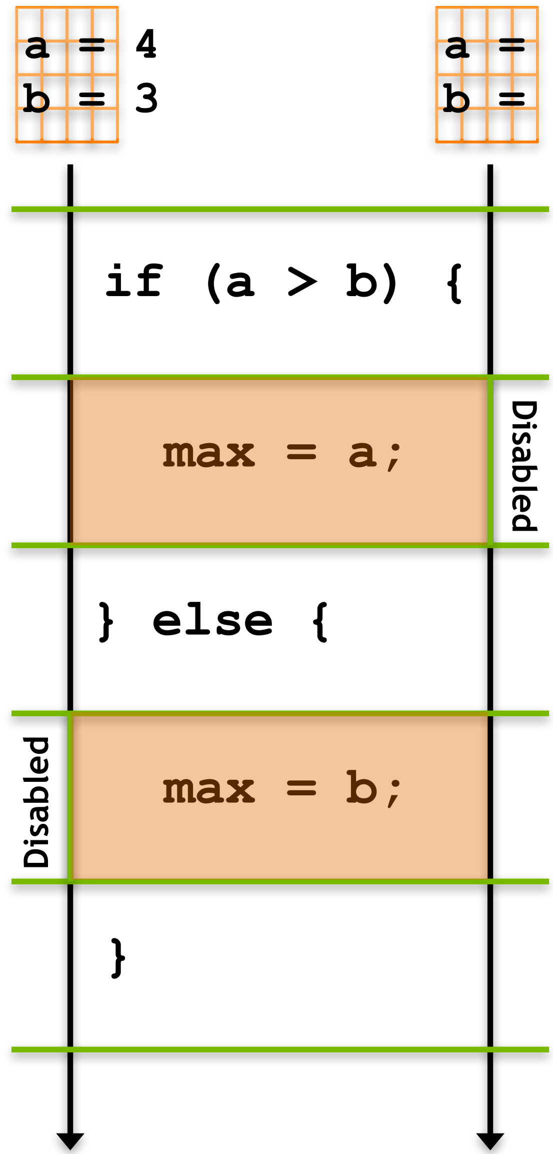
- In SIMT, all threads share instructions but operate on their own private registers, allowing threads to store thread-local state



GPU Multi-Threading

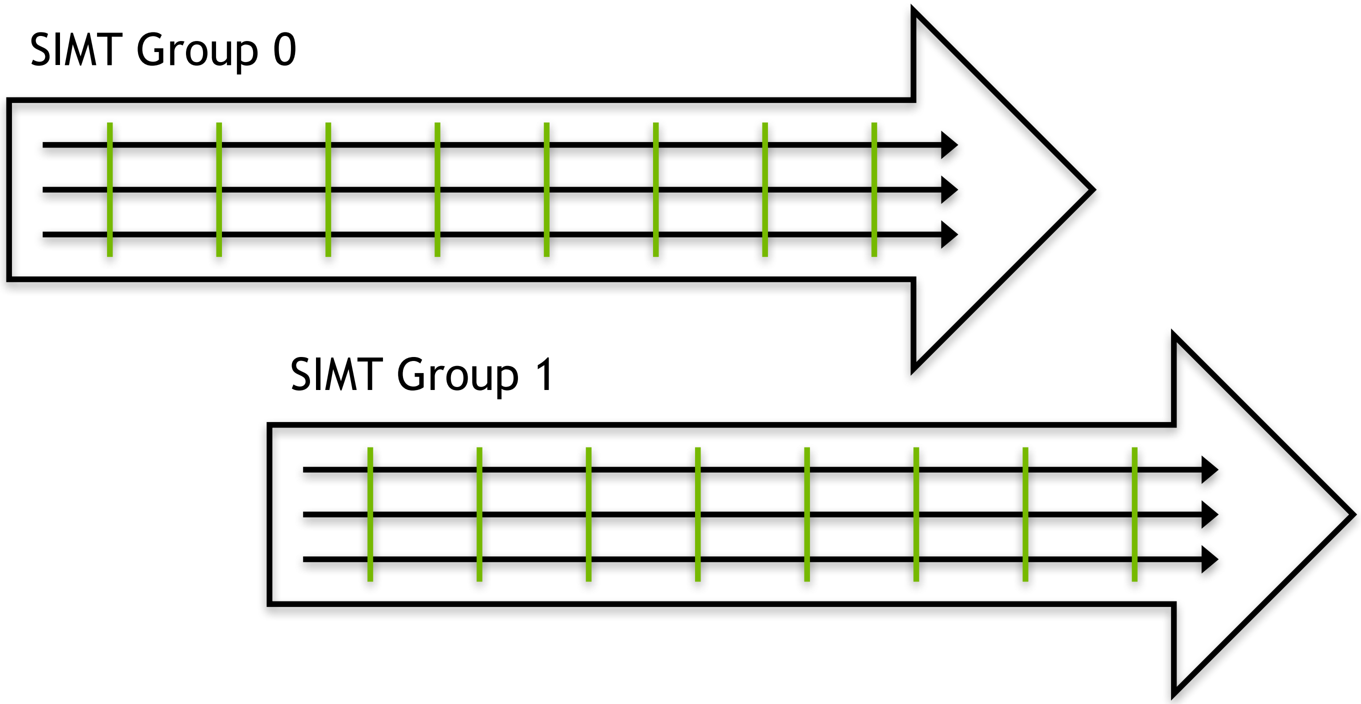
- SIMT threads can be "**disabled**" when they need to execute instructions different from others in their group

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



GPU Multi-Threading

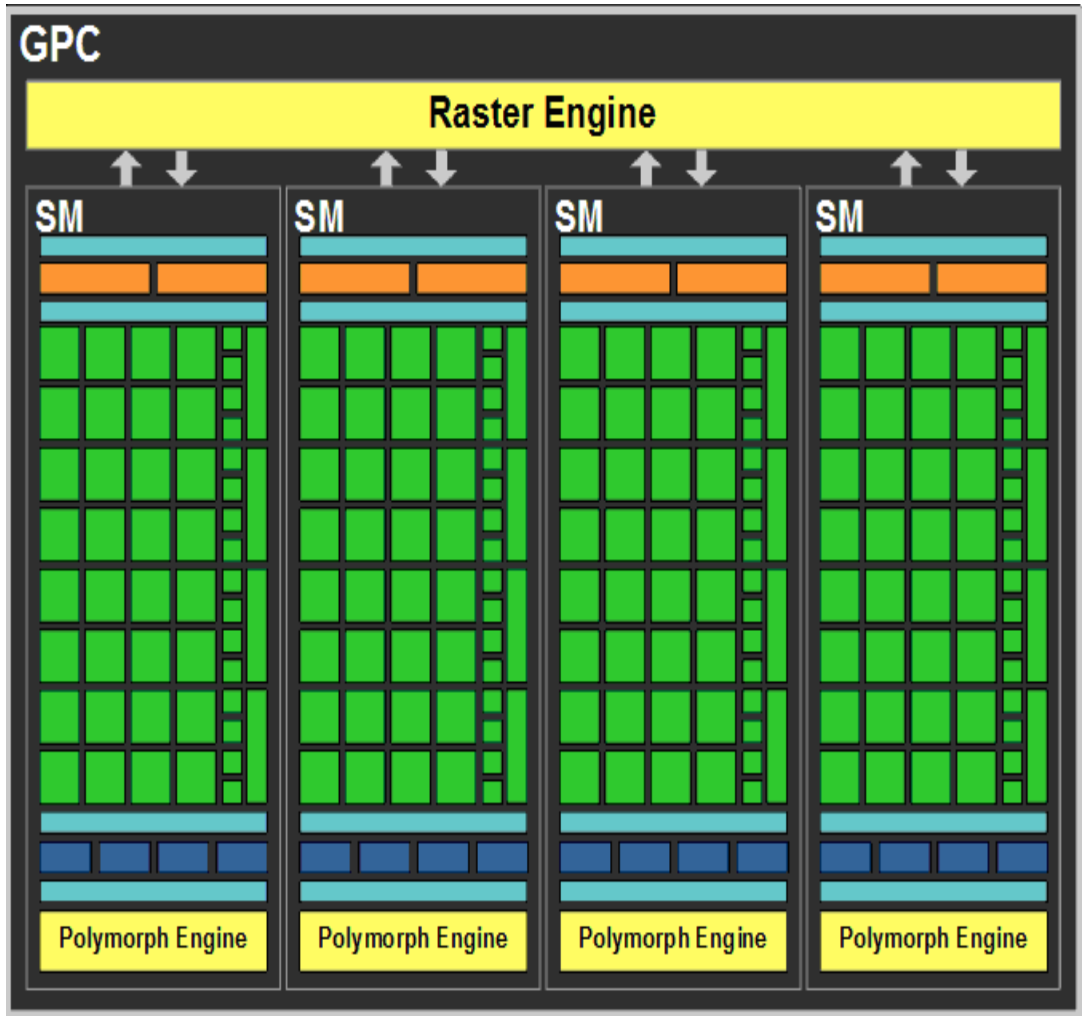
- GPUs execute many groups of SIMT threads in parallel
  - Each executes instructions independent of the others



Execution Model to Hardware

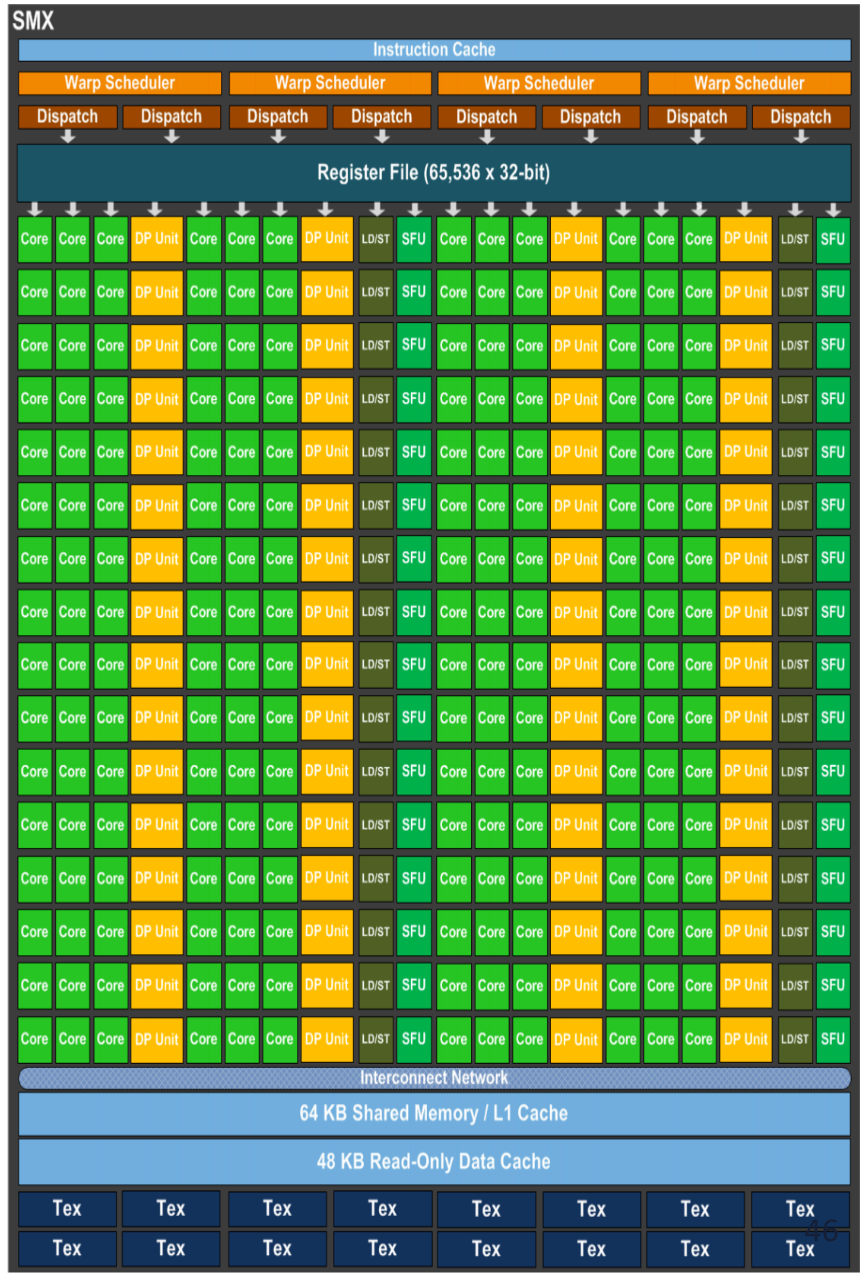
- How does this execution model map down to actual GPU hardware?

- NVIDIA GPUs consist of many streaming multiprocessors (SM)



Execution Model to Hardware

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



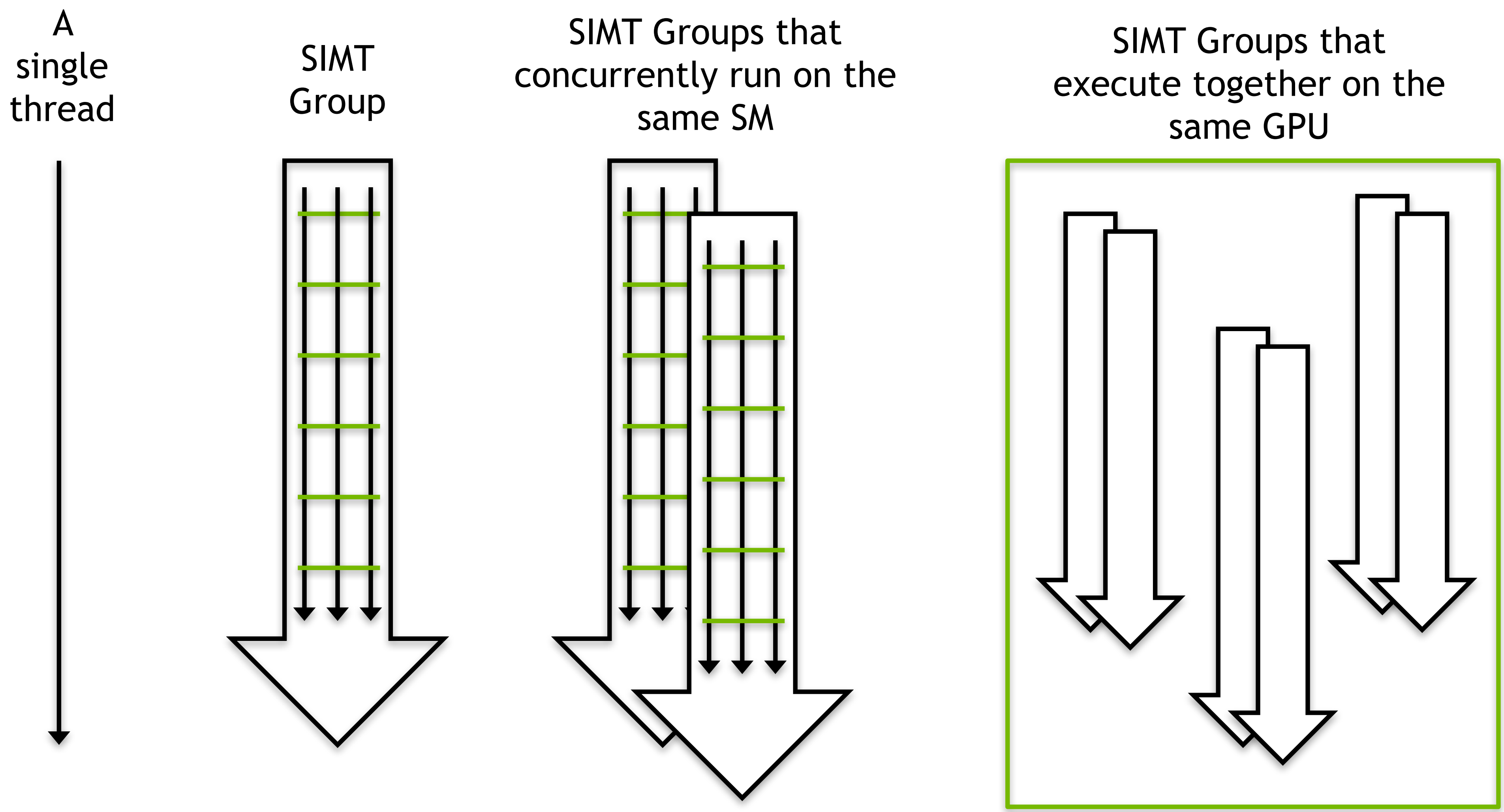
Execution Model to Hardware

- 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

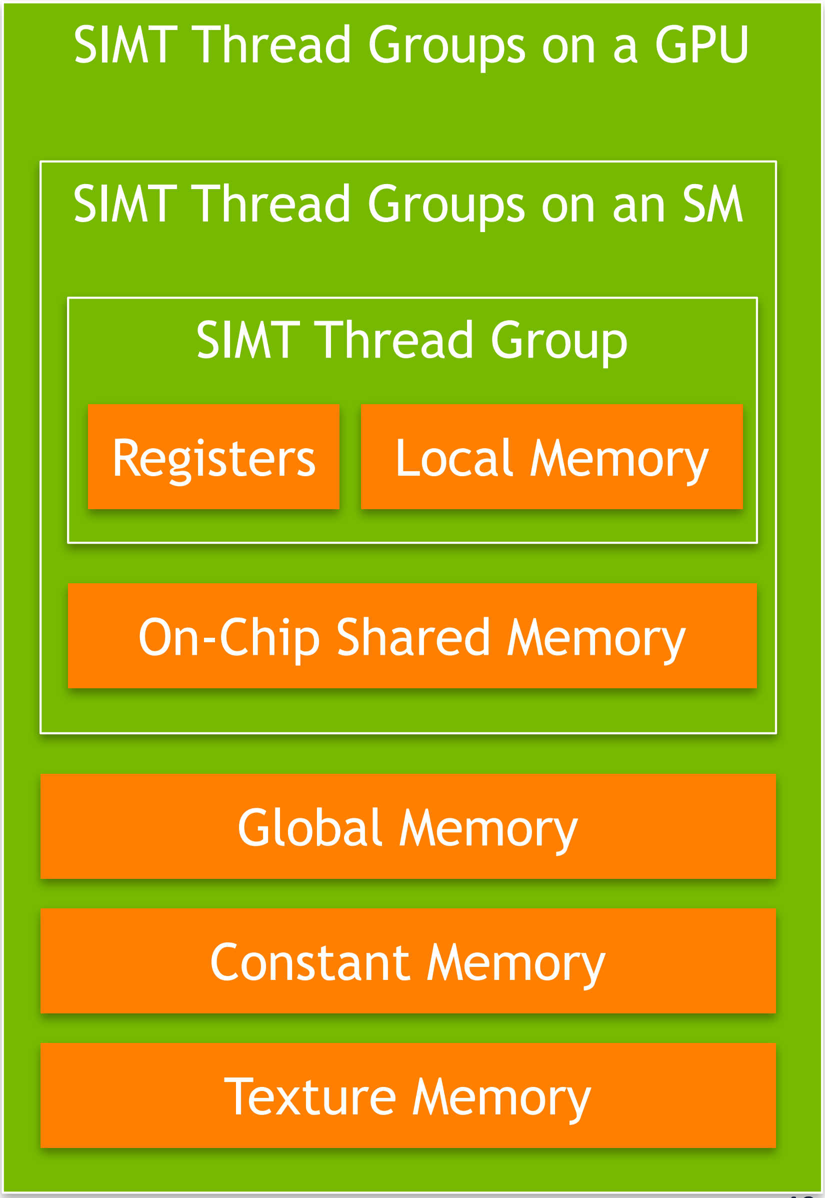
Execution Model to Hardware

- This leads to a nested thread hierarchy on GPUs



GPU Memory Model

- Now that we understand how abstract threads of execution are mapped to the GPU:
  - How do those threads store and retrieve data?
  - What rules are there about memory consistency?
  - How can we efficiently use GPU memory?



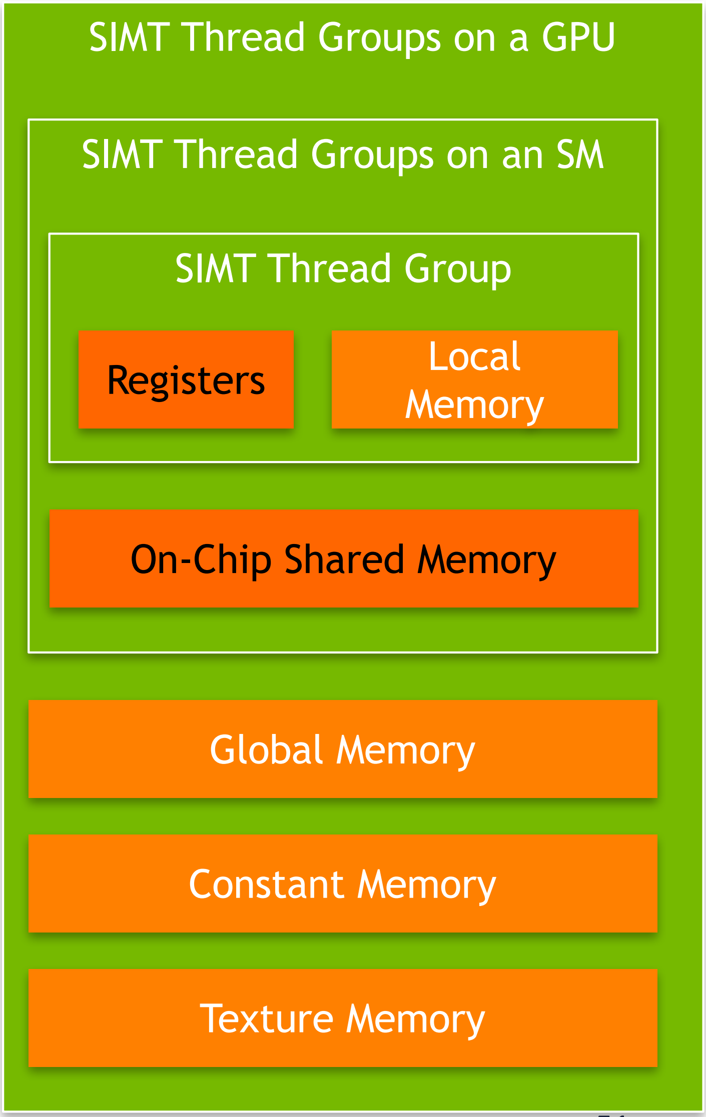
GPU Memory Model

- There are many levels and types of GPU memory, each of which has special characteristics that make it useful
  - Size
  - Latency
  - Bandwidth
  - Readable and/or Writable
  - Optimal Access Patterns
  - Accessibility by threads in the same SIMT group, SM, GPU

- Later lectures will go into detail on each type of GPU memory

GPU Memory Model

- For now, we focus on two memory types: on-chip shared memory and registers
  - These memory types affect the GPU execution model
- Each SM has a limited set of registers, each thread receives its own private set of registers
- Each SM has a limited amount of Shared Memory, all SIMT groups on an SM share that Shared Memory



GPU Memory Model

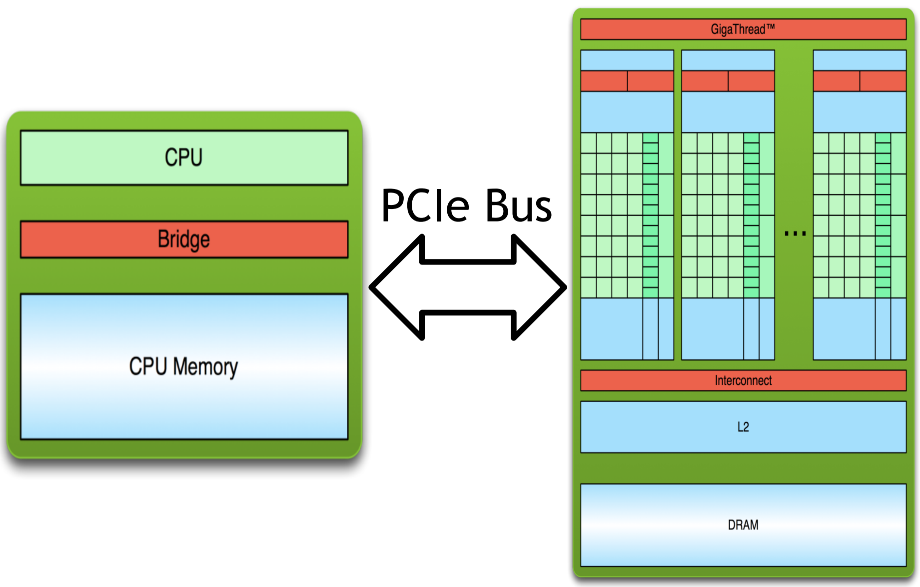
- → Shared Memory and Registers are limited
  - Per-SM resources which can impact how many threads can execute on an SM

- For example: consider an imaginary SM that supports executing 1,024 threads concurrently (32 SIMT groups of 32 threads)
  - Suppose that SM has a total of 16,384 registers
  - Suppose each thread in an application requires 64 registers to execute
  - Even though we can theoretically support 1,024 threads, we can only simultaneously store state for 16,384 registers / 64 registers per thread = **256 threads**

GPU Communication

- Communicating between the host and GPU is a piece of added complexity, relative to homogeneous programming models

- Generally, CPU and GPU have physically and logically separate address spaces (though this is changing)



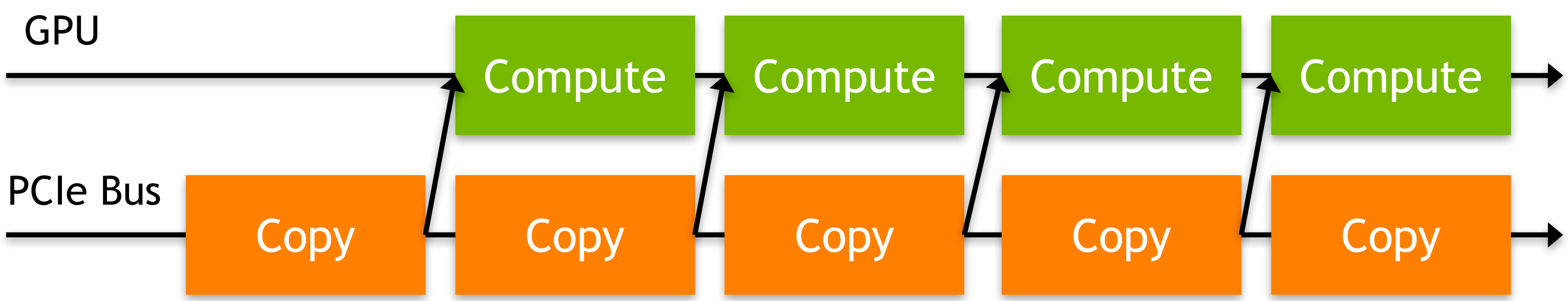
GPU Communication

- 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 Memory	Hundreds of clock cycles	Hundreds of GB/s
PCI Bus	Hundreds to thousands of clock cycles	Tens of GB/s

GPU Communication

- As a result, computation-communication overlap is a common technique in GPU programming
  - Asynchrony is a first-class citizen of most GPU programming frameworks



GPU Execution Model

- GPUs introduce a new conceptual model for programmers used to CPU single- and multi-threaded programming
- While the concepts are different, they are no more complex than those you would need to learn to extract optimal performance from CPU architectures
- GPUs offer programmers more control over how their workloads map to hardware, which makes the results of optimizing applications more predictable

References

1. The sections on *Introducing the CUDA Execution Model*, *Understanding the Nature of Warp Execution*, and *Exposing Parallelism* in Chapter 3 of *Professional CUDA C Programming*
2. Michael Wolfe. *Understanding the CUDA Data Parallel Threading Model*. https://www.pgroup.com/lit/articles/insider/v2n1a5.htm
3. Will Ramey. *Introduction to CUDA Platform*. http://developer.download.nvidia .com/compute/developertrainingmaterials/presentations/general/Why_GPU_ Computing.pptx
4. Timo Stich. *Fermi Hardware & Performance Tips.* http://theinf2.informatik.uni-jena.de/ theinf2_multimedia/Website_downloads/NVIDIA_Fermi_Perf_Jena_ 2011.pdf