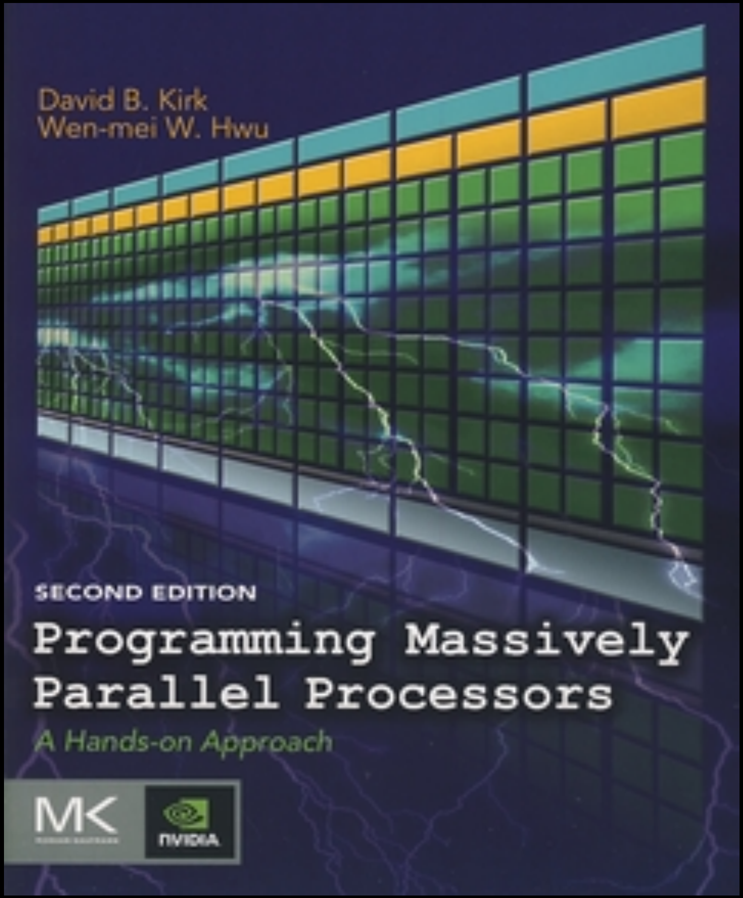
Introduction to Parallel Programming & CUDA

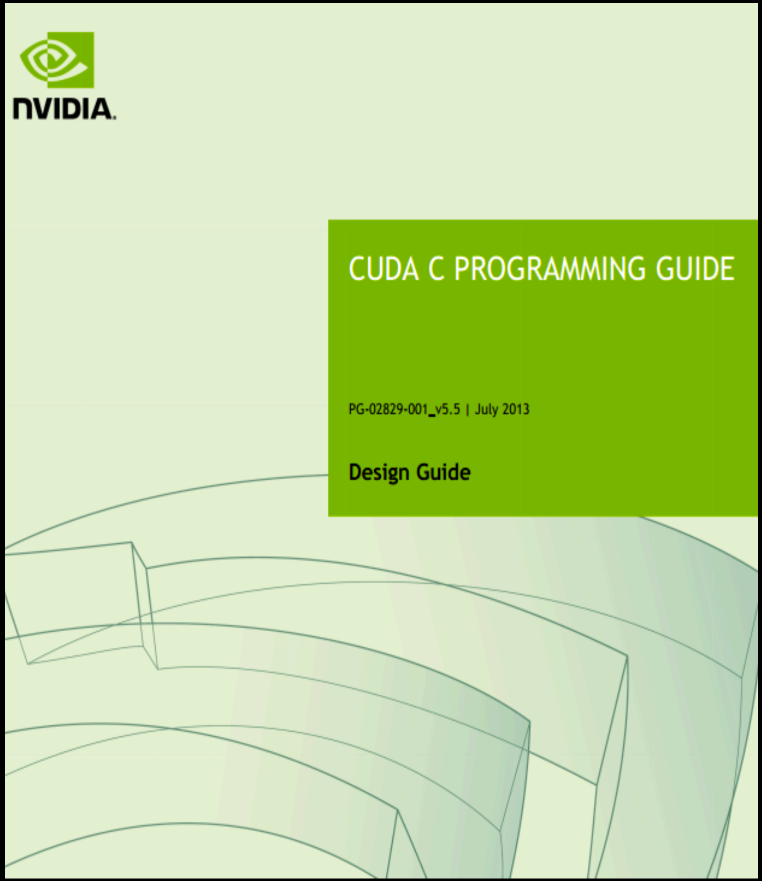
Abhijit Bendale

CS 4440/5440

03/04/2014

Textbook





Available on Nvidia's website

Goals

- Learn how to program massively parallel processors and achieve
  - High performance
  - Functionality and maintainability
  - Scalability across future generations
- Acquire technical knowledge required to achieve above goals
  - Principles and patterns of parallel programming
  - Processor architecture features and constraints
  - Programming API, tools and techniques

Moore’s Law (paraphrased)

“The number of transistors on an integrated circuit doubles every two years.”

– Gordon E. Moore

Moore's Law

Motivation

Share

- The most economic number of components in an IC will double every year
- Historically – CPUs get faster

→ Hardware reaching frequency limitations

- Now – CPUs get wider

slide by Matthew Bolitho

Parallel Computing

GPU?

Share

- Rather than expecting CPUs to get twice as fast, expect to have twice as many!

- Parallel processing for the masses
- Unfortunately: Parallel programming is hard.

→ Algorithms and Data Structures must be fundamentally redesigned

slide by Matthew Bolitho

Serial Performance Scaling is Over

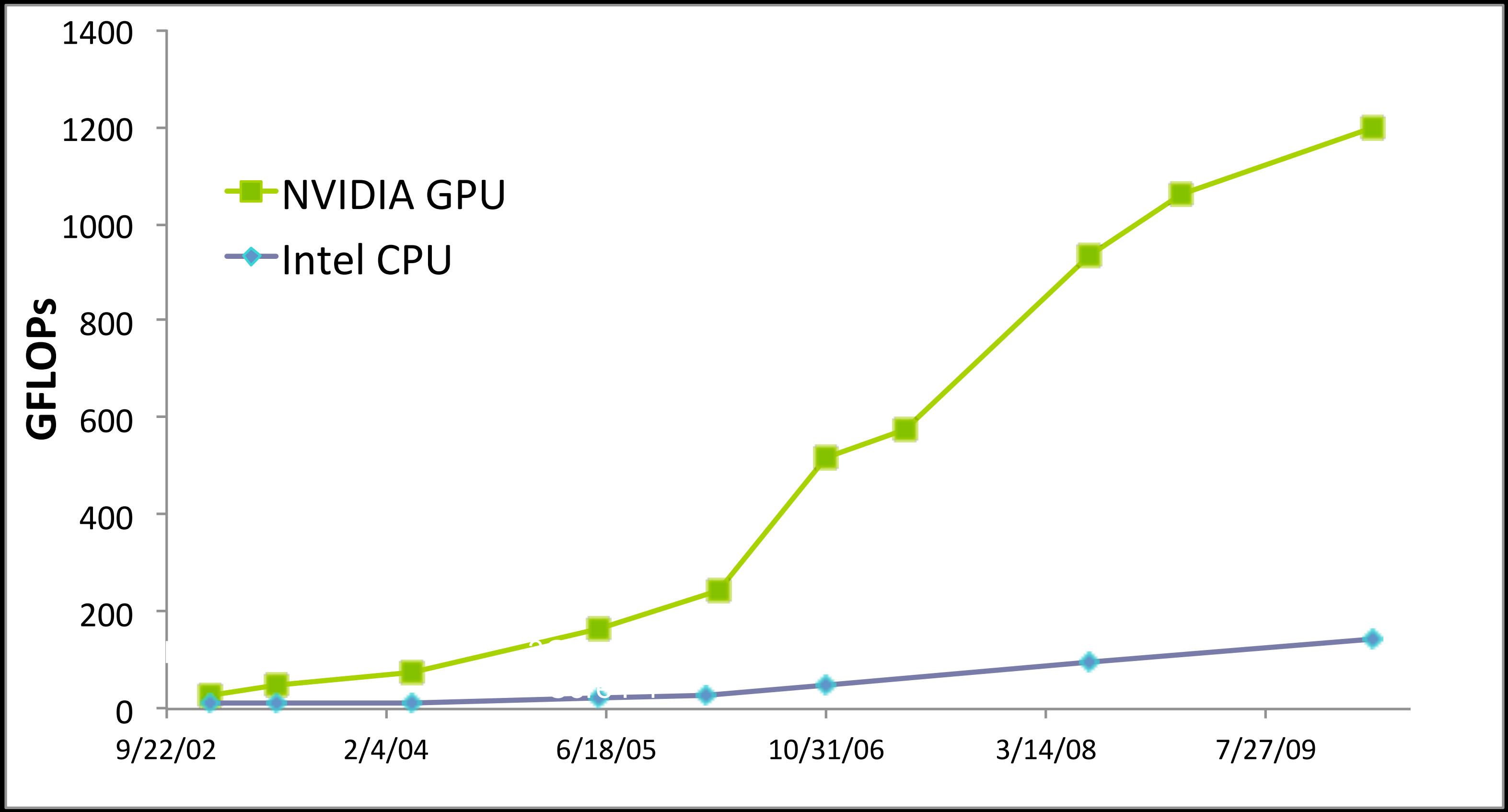
- Cannot continue to scale processor frequencies
  - no 10 GHz chips
- Cannot continue to increase power consumption
  - can’t melt chip
- Can continue to increase transistor density
  - as per Moore’s Law

How to Use Transistors?

- Instruction-level parallelism
  - out-of-order execution, speculation, …
  - vanishing opportunities in power-constrained world
- Data-level parallelism
  - vector units, SIMD execution, …
  - increasing … SSE, AVX, Cell SPE, Clearspeed, GPU
- Thread-level parallelism
  - increasing … multithreading, multicore, manycore
  - Intel Core2, AMD Phenom, Sun Niagara, STI Cell, NVIDIA Fermi, …

Why Massively Parallel Processing?

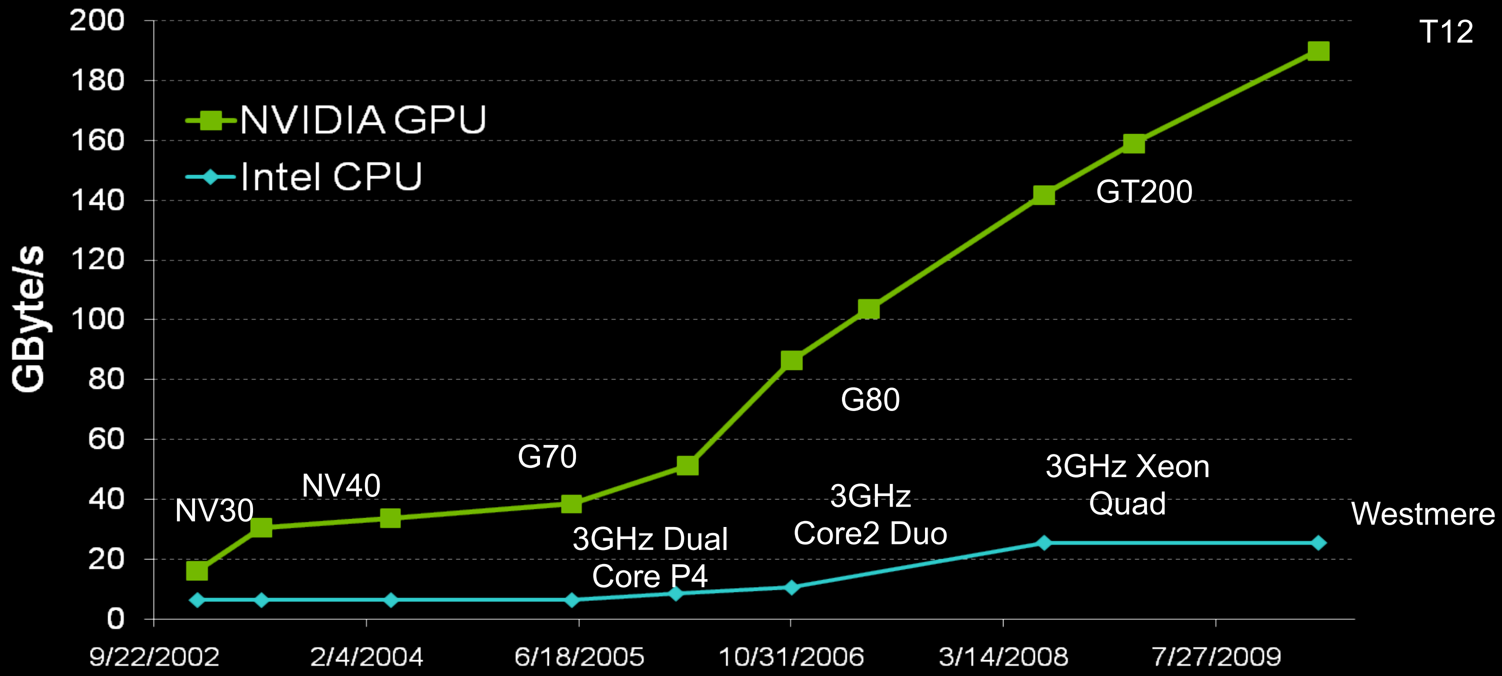
- A quiet revolution and potential build-up
  - Computation: TFLOPs vs. 100 GFLOPs



FLOPS: Floating point operation per second = cores x clock x FLOPs/cycle
e.g. 4 FLOPs/Cycle i.e 2.5 GHz processor has theoretical performance of 10 Billion FLOPS i.e 10 GFlops

Why Massively Parallel Processing?

- A quiet revolution and potential build-up
  - Bandwidth: ~10x



  - GPU in every PC – massive volume & potential impact

The "New" Moore's Law

- Computers no longer get faster, just wider

- You *must* re-think your algorithms to be parallel !

~~4 cores~~ ~~8 cores~~ 16 cores…

- Data-parallel computing is most scalable solution
  - Otherwise: refactor code for 2 cores
  - You will always have more data than cores – build the computation around the data

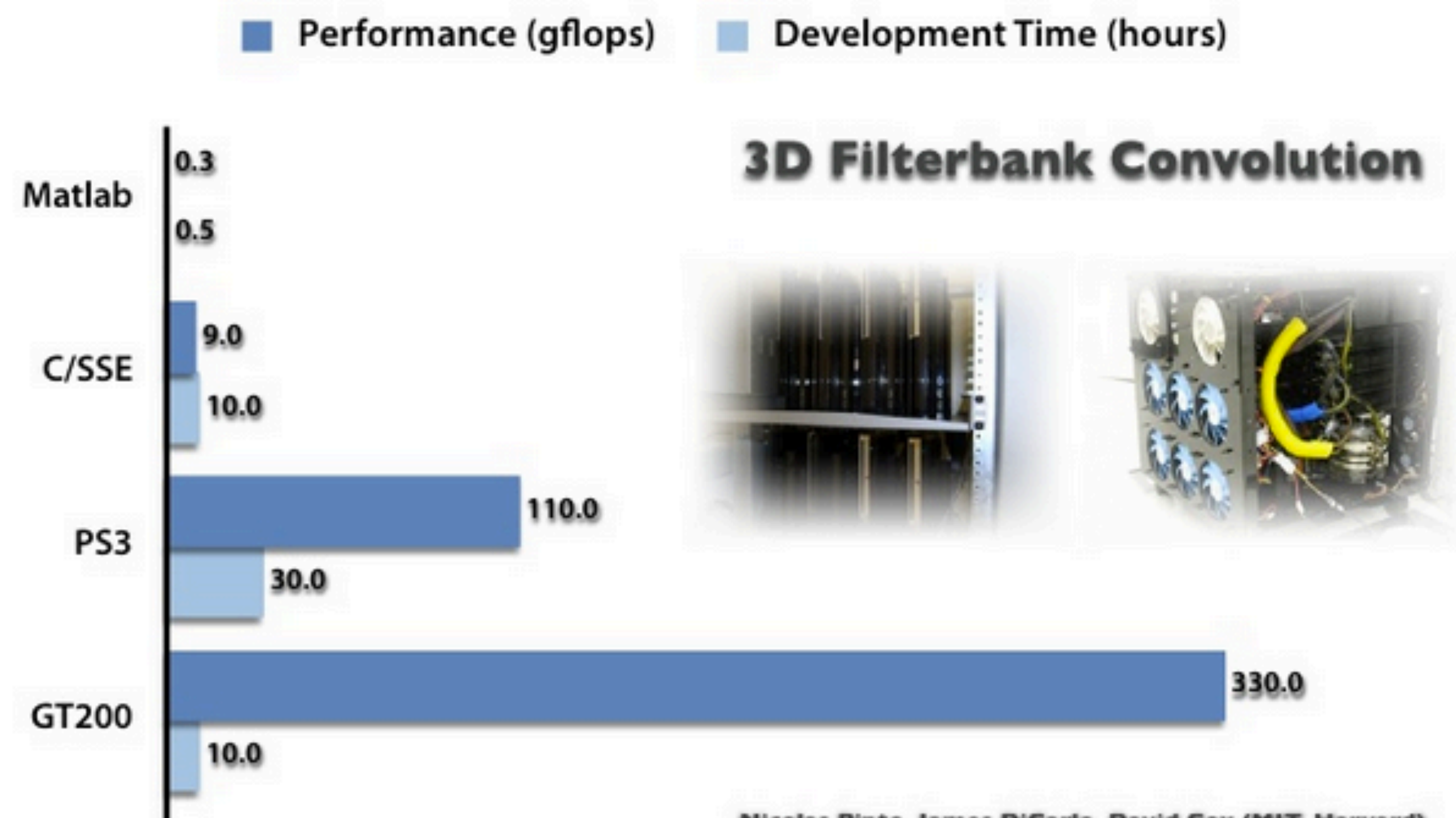
Enter the GPU

- Massive economies of scale
- Massively parallel

GPUs are **REALLY** fast

GPU?

Share

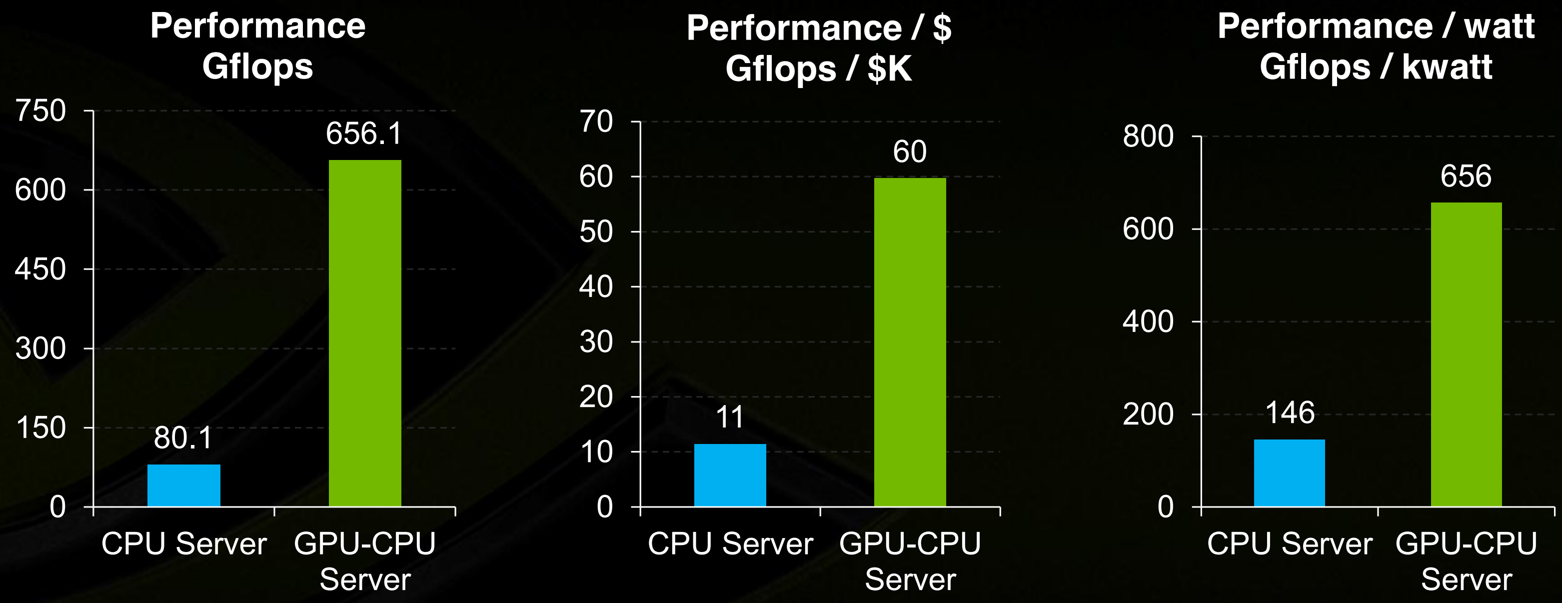


3D Filterbank Convolution

Nicolas Pinto, James DiCarlo, David Cox (MIT, Harvard)

GPUs are Fast!

8x Higher Linpack



CPU 1U Server: 2x Intel Xeon X5550 (Nehalem) 2.66 GHz, 48 GB memory, $7K, 0.55 kw

GPU-CPU 1U Server: 2x Tesla C2050 + 2x Intel Xeon X5550, 48 GB memory, $11K, 1.0 kw

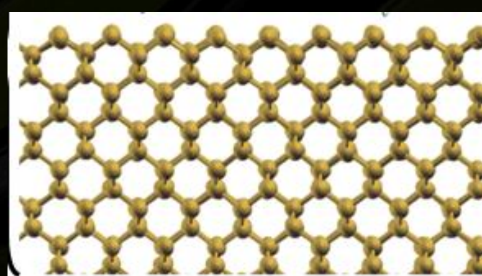
World's Fastest MD Simulation

Sustained Performance of 1.87 Petaflops/s

Institute of Process Engineering (IPE)

Chinese Academy of Sciences (CAS)

MD Simulation for Crystalline Silicon



Used all 7168 Tesla GPUs on Tianhe-1A GPU Supercomputer

World's Greenest Petaflop Supercomputer

Tsubame 2.0

Tokyo Institute of Technology

- 1.19 Petaflops
- 4,224 Tesla M2050 GPUs

Increasing Number of Professional CUDA Applications

Available Now → ········ Future ··········

Tools & Libraries	CUDA C/C++	Parallel Nsight Vis Studio IDE	NVIDIA Video Libraries	ParaTools VampirTrace	PGI Accelerators	EMPhotonics CULAPACK	Allinea DDT Debugger	TauCUDA Perf Tools	PGI CUDA-X86
	NVIDIA NPP Perf Primitives	PGI Fortran	Thrust C++ Template Lib	Bright Cluster Manager	CAPS HMPP	MAGMA	GPU Packages For R Stats Pkg	Platform LSF Cluster Mgr	GPU.net
	pyCUDA	R-Stream Reservoir Labs	PBSWorks	MOAB Adaptive Comp	Torque Adaptive Comp	TotalView Debugger	IMSL		
Oil & Gas	Headwave Suite	OpenGeo Solns OpenSEIS	GeoStar Seismic	Acceleware RTM Solver	StoneRidge RTM	Seismic City RTM	Tsunami RTM		Schlumberger Petrel
	ffA SVI Pro	Paradigm SKUA	VSG Open Inventor	Paradigm GeoDepth RTM	VSG Avizo	SVI Pro	SEA 3D Pro 2010	Schlumberger Omega	Paradigm VoxelGeo
Numerical Analytics	LabVIEW Libraries	AccelerEyes Jacket: MATLAB	MATLAB	Mathematica					
Finance	NAG RNG	Numerix CounterpartyRisk	SciComp SciFinance	Aquimin AlphaVision	Hanweck Volera Options Analysi	Murex MACS			
Other	Siemens 4D Ultrasound	Digisens CT	Schrodinger Core Hopping	Useful Prog Medical Imag	ASUCA Weather Model				
	Manifold GIS	MVTech Mach Vision	Dalsa Mach Vision	WRF Weather					

Available | Announced

NVIDIA Developer Ecosystem

NVIDIA

Numerical Packages
- MATLAB
- Mathematica
- NI LabView
- pyCUDA

Debuggers & Profilers
- cuda-gdb
- NV Visual Profiler
- Parallel Nsight
- Visual Studio
- Allinea
- TotalView

GPU Compilers
- C
- C++
- Fortran
- OpenCL
- DirectCompute
- Java
- Python

Parallelizing Compilers
- PGI Accelerator
- CAPS HMPP
- mCUDA
- OpenMP

Libraries
- BLAS
- FFT
- LAPACK
- NPP
- Video
- Imaging
- GPULib

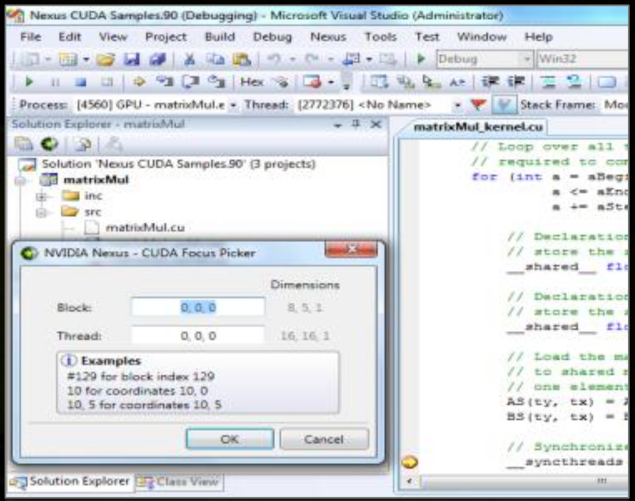
GPGPU Consultants & Training

acceleware, GASS, STONE RIDGE TECHNOLOGY, ANEO, GPU Tech, TECH-X, CAPS, HPC, EM Photonics, SCALABLE GRAPHICS, WIPRO

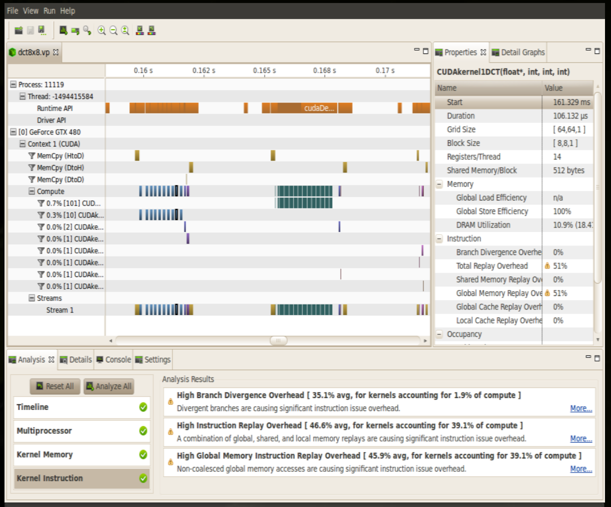
OEM Solution Providers

DELL, hp, IBM, CRAY, ASUS, SUPERMICRO, sgi, FUJITSU, Bull, APPRO, lenovo, NEC

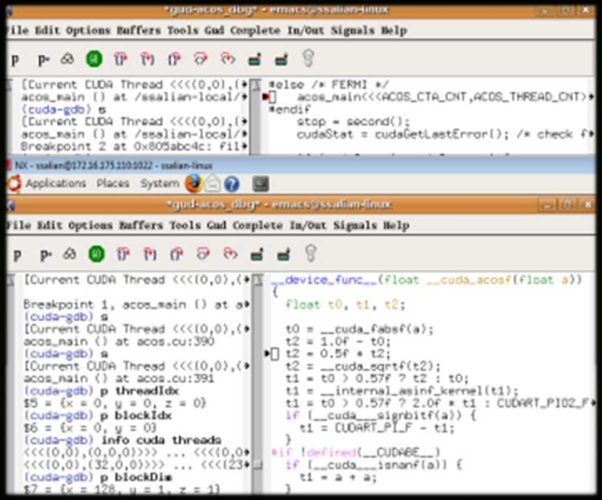
Parallel Nsight
Visual Studio



Visual Profiler
Windows/Linux/Mac



cuda-gdb
Linux/Mac



Why so fast?

GPU?

Share

- Designed for math-intensive, parallel problems:

- More transistors dedicated to ALU than flow control and data cache

slide by Matthew Bolitho

Is it free?

GPU?

Share

- What are the consequences?
- Program must be more predictable:
  - Data access coherency
  - Program flow

slide by Matthew Bolitho

CPU vs. GPU

- CPU
  - Really fast caches (great for data reuse)
  - Fine branching granularity
  - Lots of different processes/threads
  - High performance on a single thread of execution
- GPU
  - Lots of math units
  - Fast access to onboard memory
  - Run a program on each fragment/vertex
  - High throughput on parallel tasks

- **CPUs are great for *task* parallelism**
- **GPUs are great for *data* parallelism**

slide by Mike Houston

Task vs. Data parallelism

- **Task parallel**
  - Independent processes with little communication
  - Easy to use
    - "Free" on modern operating systems with SMP
- **Data parallel**
  - Lots of data on which the same computation is being executed
  - No dependencies between data elements in each step in the computation
  - Can saturate many ALUs
  - But often requires redesign of traditional algorithms

slide by Mike Houston

The Importance of Data Parallelism for GPUs

- GPUs are designed for highly parallel tasks like rendering
- GPUs process *independent* vertices and fragments
  - Temporary registers are zeroed
  - No shared or static data
  - No read-modify-write buffers
  - In short, no communication between vertices or fragments
- Data-parallel processing
  - GPU architectures are ALU-heavy
    - Multiple vertex & pixel pipelines
    - Lots of compute power
  - GPU memory systems are designed to ***stream*** data
    - Linear access patterns can be prefetched
    - Hide memory latency

slide by Mike Houston

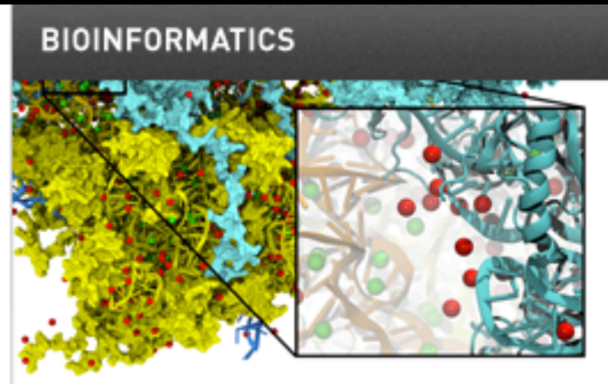
Never believe anything unless you have seen it on Mythbusters

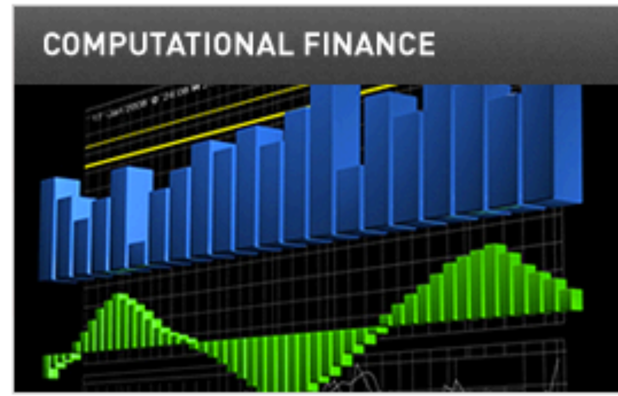
Where are GPUs used?

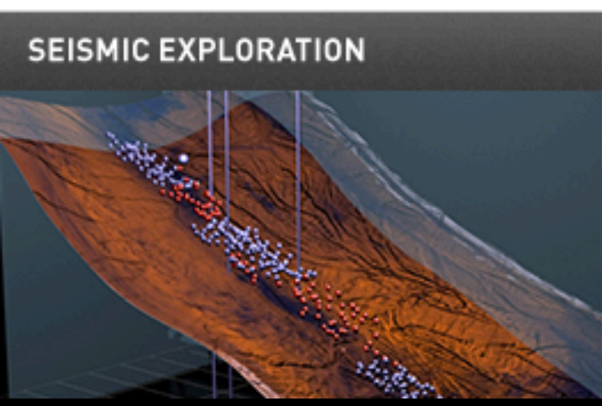
GTX 760M	1080p, High	1080p, High	1080p, High	1080p, High	1080p, High
GTX 780M	1080p, Ultra	1080p, Ultra	1080p, Ultra	1080p, Ultra	1080p, Ultra

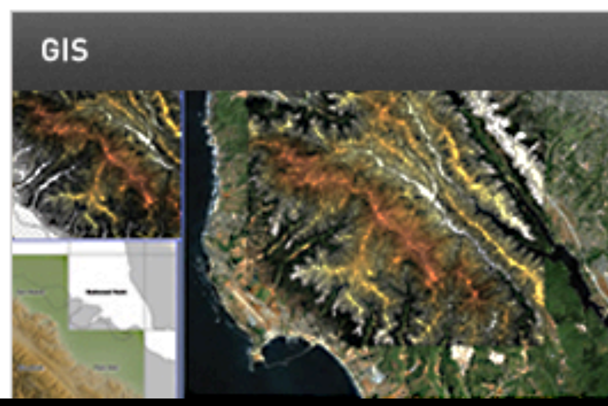
Computer Games industry is the biggest force behind development of GPU Technology

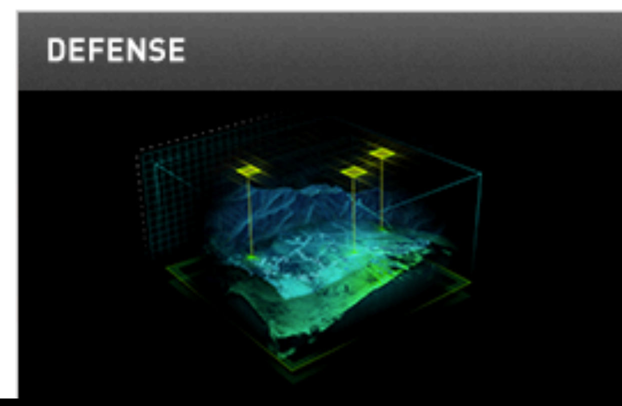
Where are GPUs used?







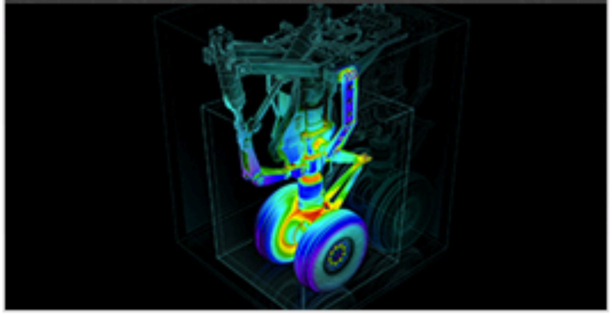




Where are GPUs used?

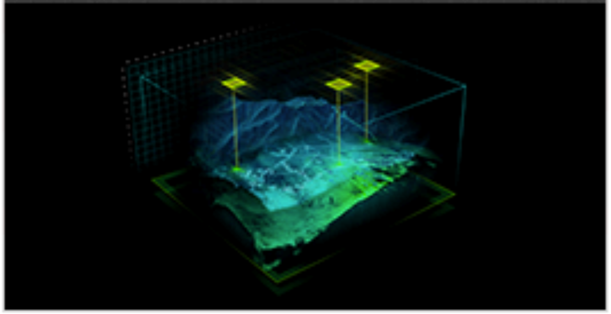
COMPUTATIONAL FLUID DYNAMICS

COMPUTATIONAL STRUCTURAL MECHANICS

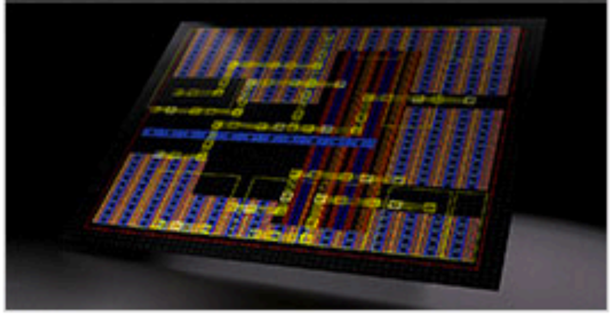


DATA SCIENCE

DEFENSE

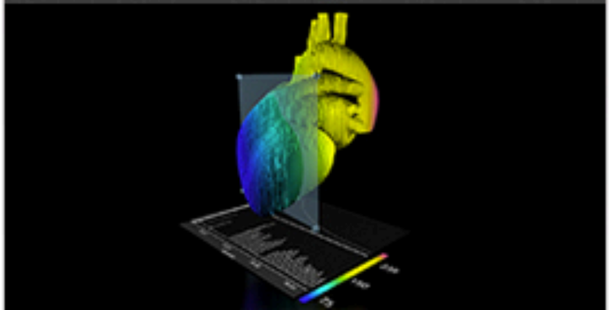


ELECTRONIC DESIGN AUTOMATION

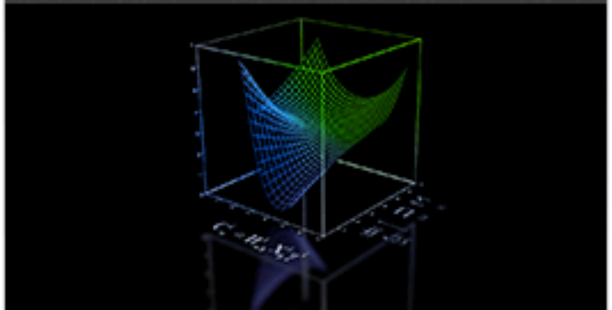


IMAGING & COMPUTER VISION

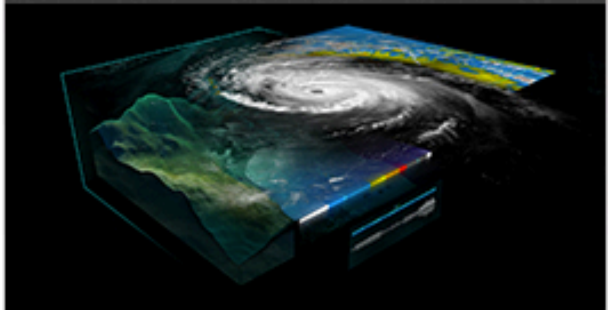
MEDICAL IMAGING

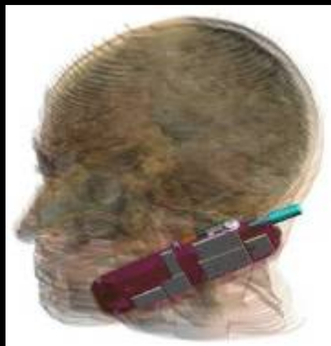


NUMERICAL ANALYTICS

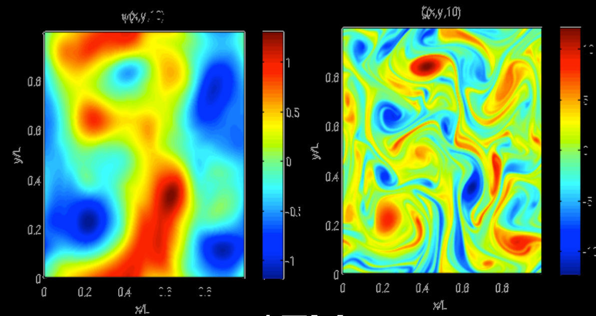


WEATHER AND CLIMATE

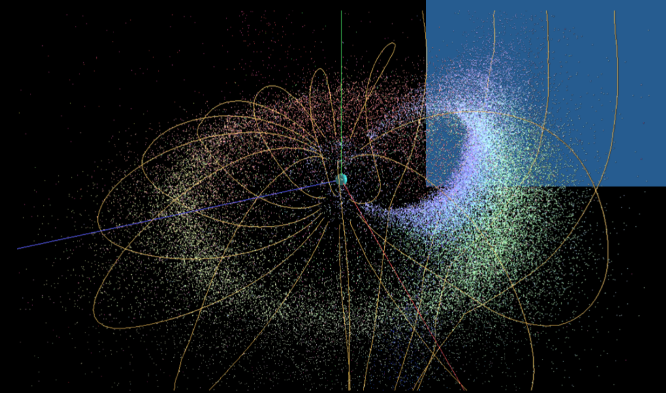




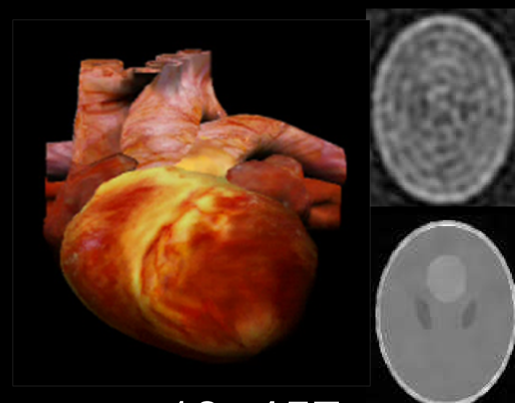
45X



17X

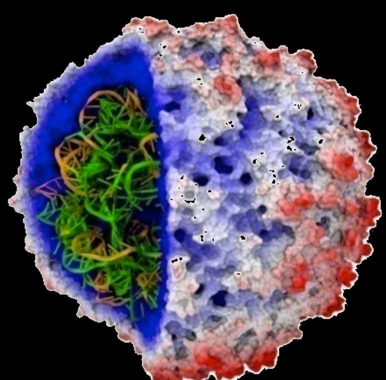


100X

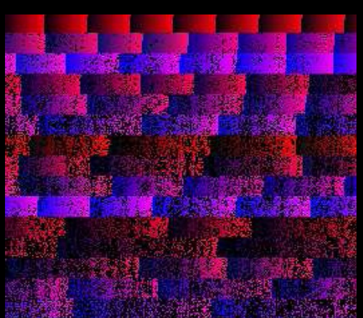


13–457x

Motivation



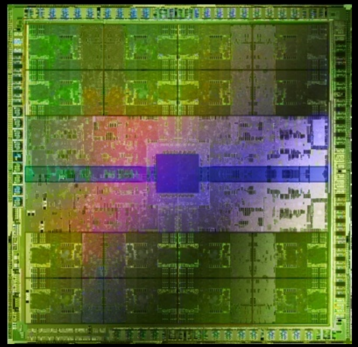
110-240X

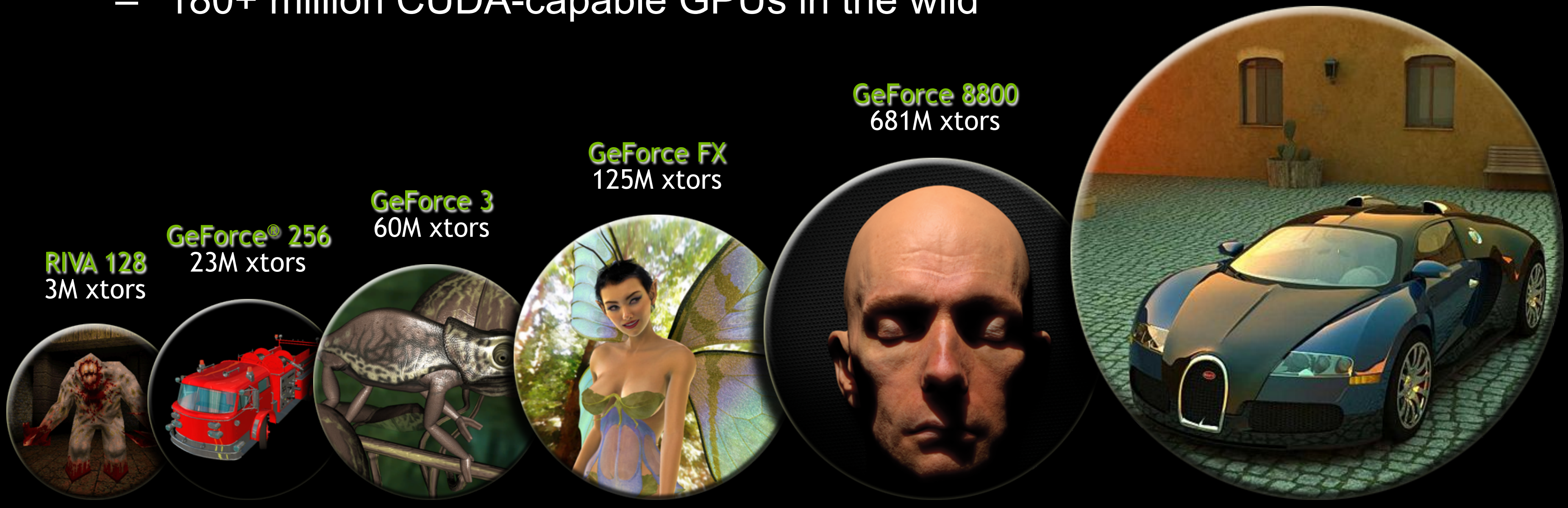


35X

GPU Evolution and History

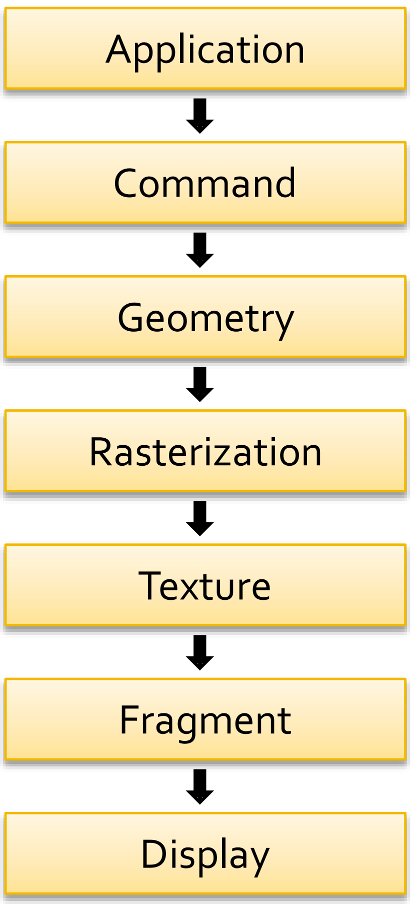
- High throughput computation
  - GeForce GTX 280: 933 GFLOP/s
- High bandwidth memory
  - GeForce GTX 280: 140 GB/s
- High availability to all
  - 180+ million CUDA-capable GPUs in the wild





Graphics Pipeline

History



The traditional model for 3-D Rendering

Input

- Vertices and Primitives
- Transformations
- Lighting Parameters, etc…

Output

- 2D Image for display

slide by Matthew Bolitho

Challenge

History

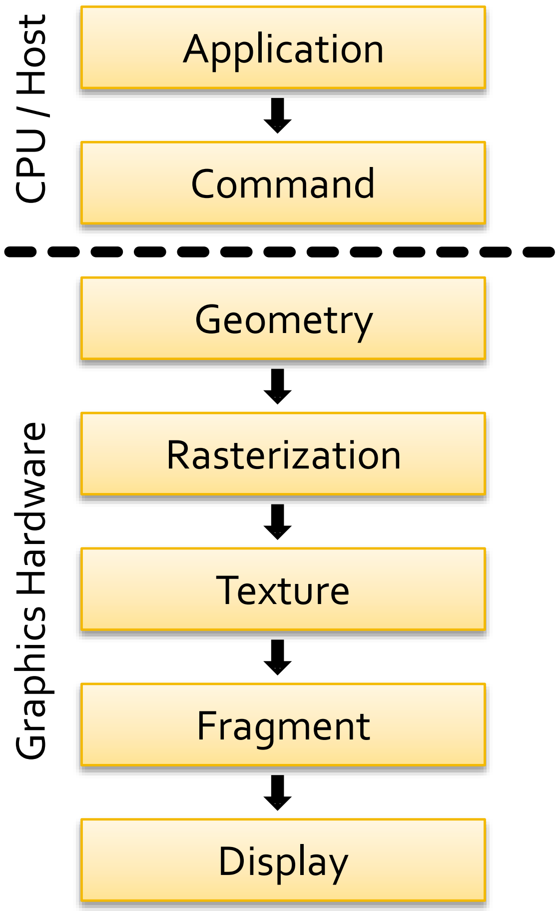
- Render **interactive**, realistic computer generated scenes
  - Each frame is complex
  - Need 60 frames per second
- CPU's were too slow!

→ **Dedicated hardware**

slide by Matthew Bolitho

Graphics Pipeline

History



- To improve performance, move some work to dedicated hardware
- Hardware could process each vertex and each fragment independently
  → **Highly Parallel**

slide by Matthew Bolitho

Evolution

History

- The Graphics Pipeline was "*fixed-function*"

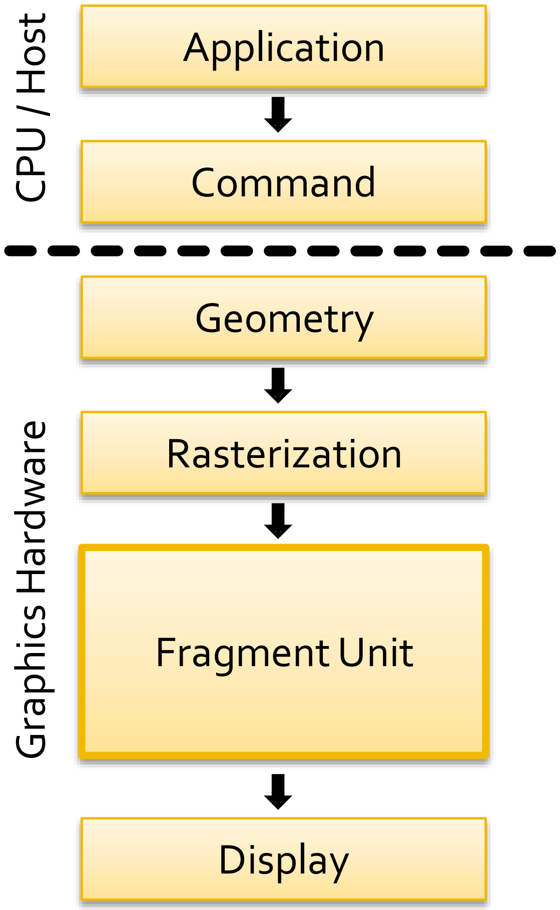
  → Hardware was hardwired to perform the operations in the pipeline

- Eventually, pipeline became more programmable

slide by Matthew Bolitho

Programability (2000)

History



- Texture and Fragment stages became more programmable, combined into “Fragment Unit”

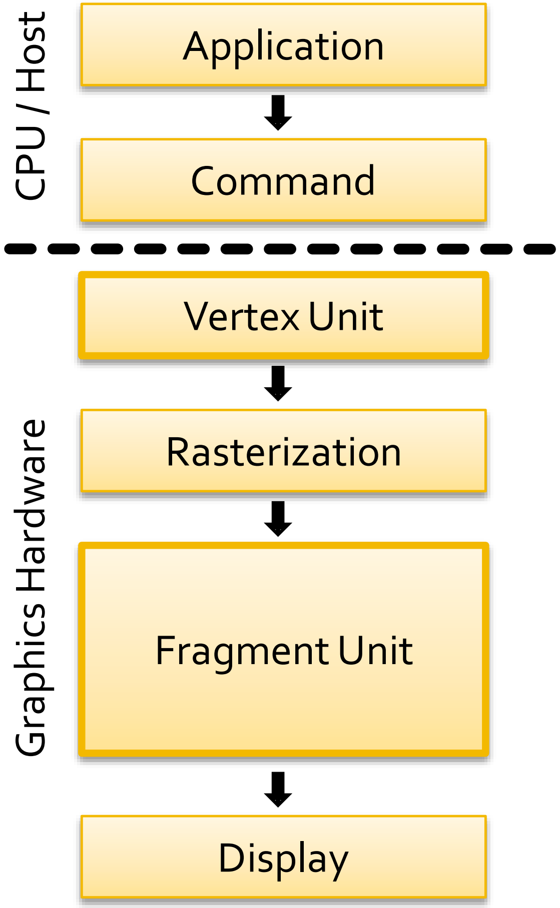
- Programmable via assembly language
- Memory reads via texture lookups
- “Dependant” texture lookups

- Limited Program size
- No real branching (thus looping)

slide by Matthew Bolitho

Programability (2001)

History



- Geometry stage became programmable, called "Vertex Unit"

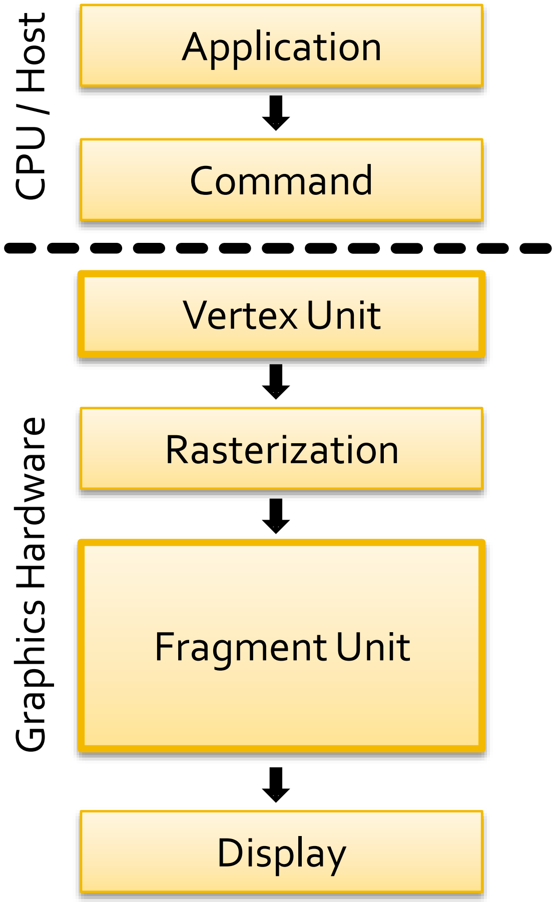
- Programmable via assembly language
- No memory reads!

- Limited Program size
- No real branching (thus looping)

slide by Matthew Bolitho

Programability (2003)

History



- Things improved over time:

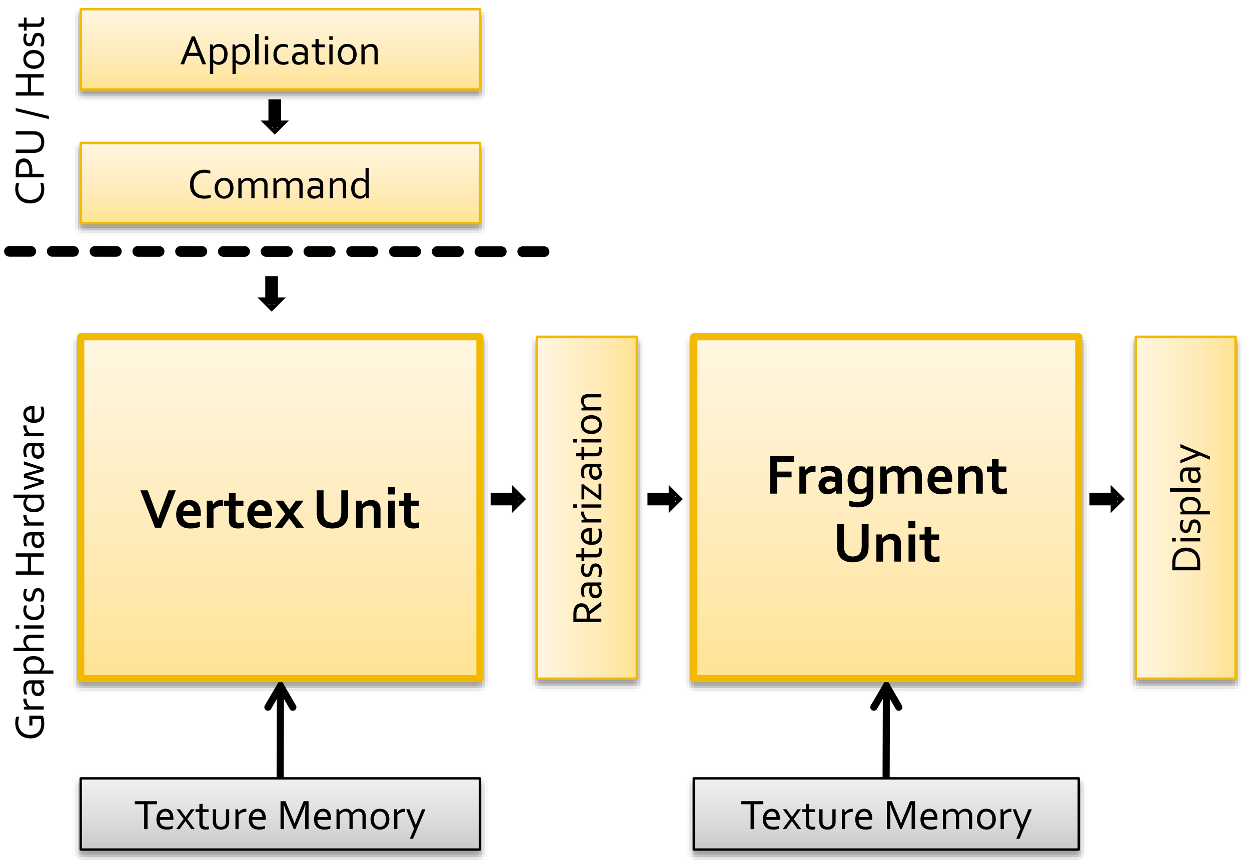
- Vertex unit can do memory reads
- Maximum Program size increased
- Branching support
- Higher level languages (e.g. HLSL, Cg)

- Neither the Vertex or Fragment units could write to memory. Can only write to frame buffer
- No integer math
- No bitwise operators

slide by Matthew Bolitho

History

Graphics Pipeline (2003)



slide by Matthew Bolitho

Graphics Pipeline (2003)

History

- In 2003 GPU's became mostly programmable,
- "Multi-pass" algorithms allowed writes to memory:
  - In pass 1 write to framebuffer
  - Rebind the framebuffer as a texture
  - Read it in pass 2, etc.
- But were inefficient

slide by Matthew Bolitho

GPGPU

History

- Despite limitations, GPGPU community grew
  (GPGPU = General Purpose Computation on the GPU)

GPGPU Program:

- Don't use Vertex Unit
- Place data in textures
- Draw a flat quad (off-screen)
- Write multi-pass algorithm using Fragment Unit to perform custom processing

slide by Matthew Bolitho

GPGPU Limitations

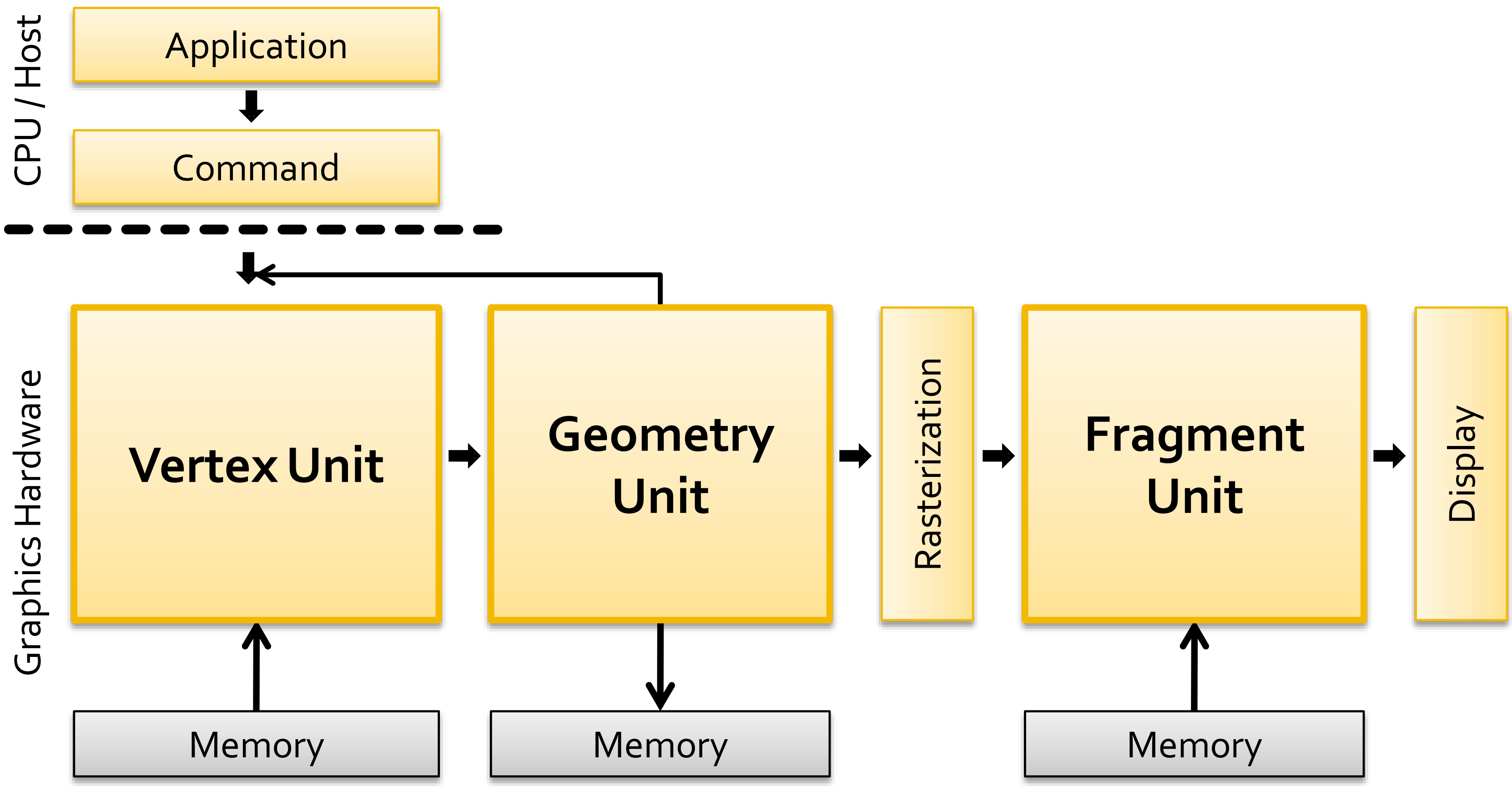
History

- Under-utilized hardware
  - Only utilized Fragment Unit
  - Often memory bandwidth limited
- Gather-based algorithms only (no scatter)
- Used the Graphics API

slide by Matthew Bolitho

Graphics Pipeline (2007)

History



slide by Matthew Bolitho

Graphics Pipeline (2007)

History

- Geometry Unit operates on a primitive, can write back to memory
- Changes to underlying hardware:
  - Ability to write to memory
  - “Unified” processing units

slide by Matthew Bolitho

Graphics in a Nutshell

- Make great images
  - intricate shapes
  - complex optical effects
  - seamless motion
- Make them fast
  - invent clever techniques
  - use every trick imaginable
  - build monster hardware

Eugene d'Eon, David Luebke, Eric Enderton
In *Proc. EGSR 2007* and *GPU Gems 3*

Lessons from Graphics Pipeline

- Throughput is paramount
  - must paint every pixel within frame time
  - scalability
- Create, run, & retire lots of threads very rapidly
  - measured 14.8 Gthread/s on `increment()` kernel
- Use multithreading to hide latency
  - 1 stalled thread is OK if 100 are ready to run

Why is this different from a CPU?

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches
  - share control logic across many threads

CUDA Overview

Problem: GPGPU

- OLD: *GPGPU* – trick the GPU into general-purpose computing by casting problem as graphics
  - Turn data into images (“texture maps”)
  - Turn algorithms into image synthesis (“rendering passes”)
- Promising results, but:
  - Tough learning curve, particularly for non-graphics experts
  - Potentially high overhead of graphics API
  - Highly constrained memory layout & access model
  - Need for many passes drives up bandwidth consumption

What Is CUDA?

Overview

- ***CUDA***: Compute Unified Device Architecture
- Created by NVIDIA

- A way to perform computation on the GPU

- Specification for:
  - A computer architecture
  - A language
  - An application interface (API)

slide by Matthew Bolitho

Prerequisites

- You (probably) need experience with C or C++
- You don't need GPU experience
- You don't need parallel programming experience
- You don't need graphics experience

Overview

CUDA Advantages over Legacy GPGPU

- Random access to memory
  - Thread can access any memory location
- Unlimited access to memory
  - Thread can read/write as many locations as needed
- User-managed cache (per block)
  - Threads can cooperatively load data into SMEM
  - Any thread can then access any SMEM location
- Low learning curve
  - Just a few extensions to C
  - No knowledge of graphics is required
- No graphics API overhead

Some Design Goals

Overview

- Scale to 100's of cores, 1000's of parallel threads
- Let programmers focus on parallel algorithms
  - Not on the mechanics of a parallel programming language
- Enable heterogeneous systems (i.e. CPU + GPU)
  - CPU and GPU are separate devices with separate DRAMs

nVISION 08 THE WORLD OF VISUAL COMPUTING

NVIDIA

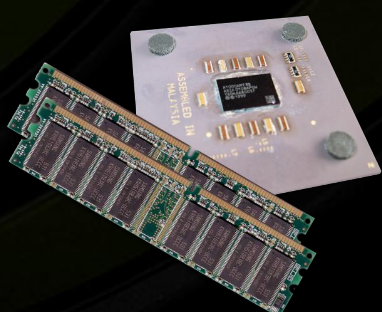
CUDA Installation

Overview

- CUDA installation consists of
  - Driver
  - CUDA Toolkit (compiler, libraries)
  - CUDA SDK (example codes)

Heterogeneous Computing

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



Host

Device

Heterogeneous Computing

device code

host code

```
#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;
}
```

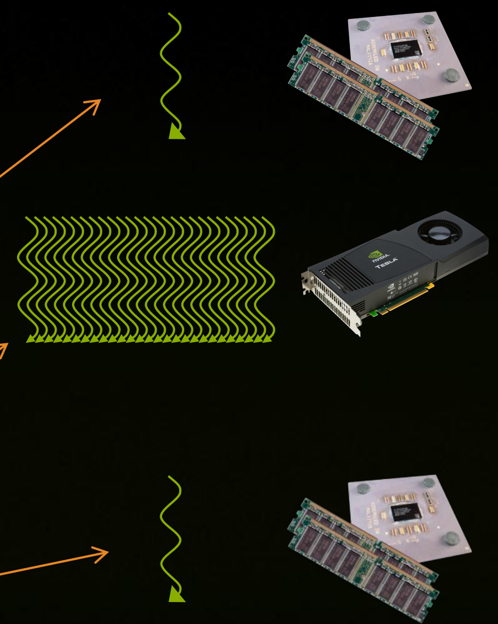
parallel function

serial function

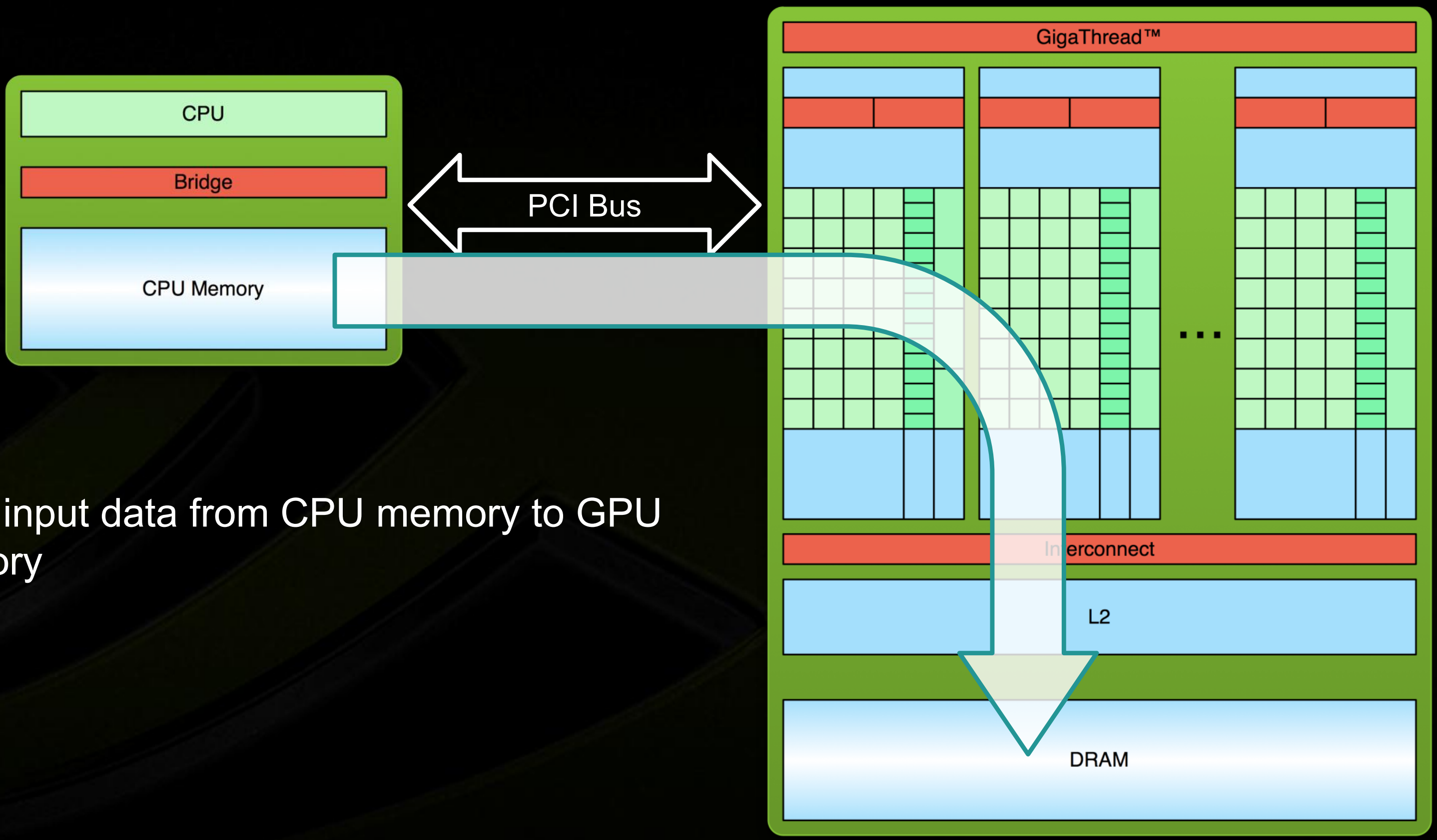
serial code

parallel code

serial code

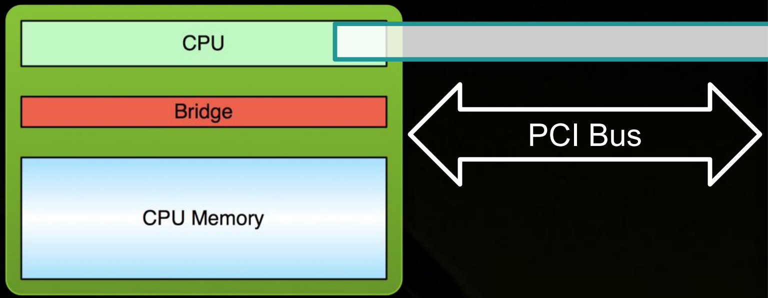


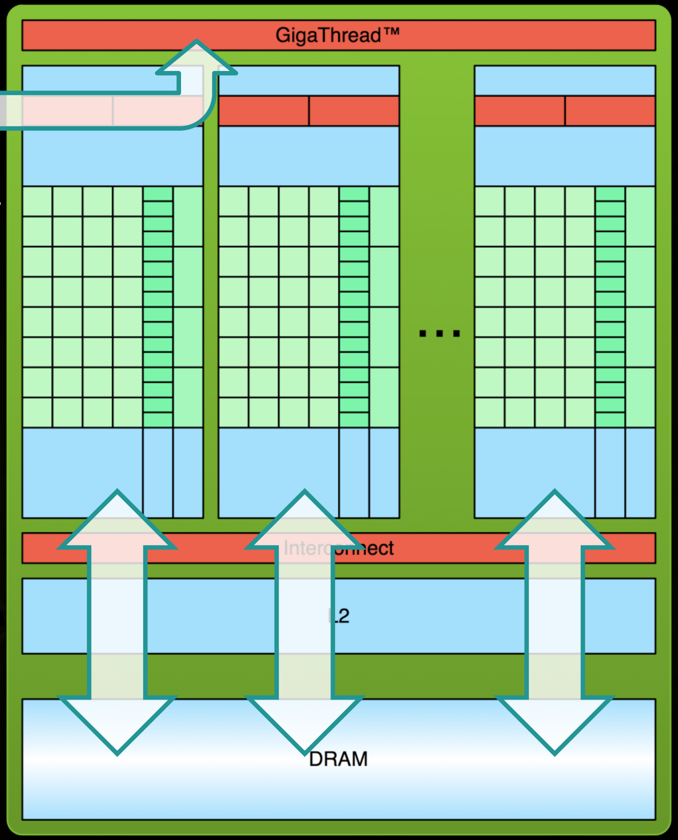
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory

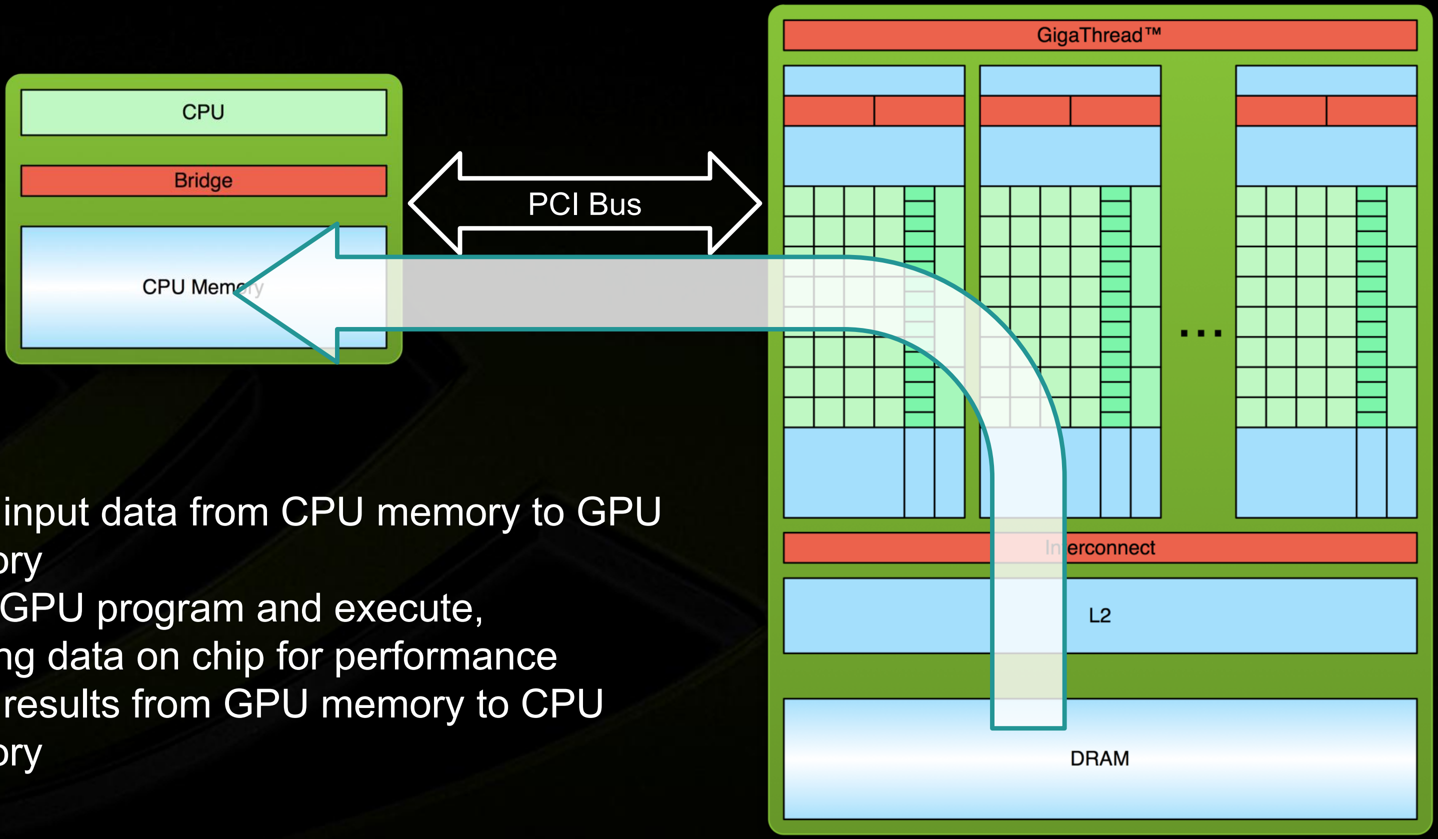
Simple Processing Flow





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

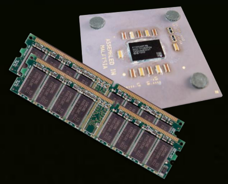
Simple Processing Flow



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

Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code



- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

CUDA Software Development

Overview

CUDA Optimized Libraries:
math.h, FFT, BLAS, …

Integrated CPU + GPU
C Source Code

NVIDIA C Compiler

NVIDIA Assembly
for Computing (PTX)

CPU Host Code

CUDA
Driver

Profiler

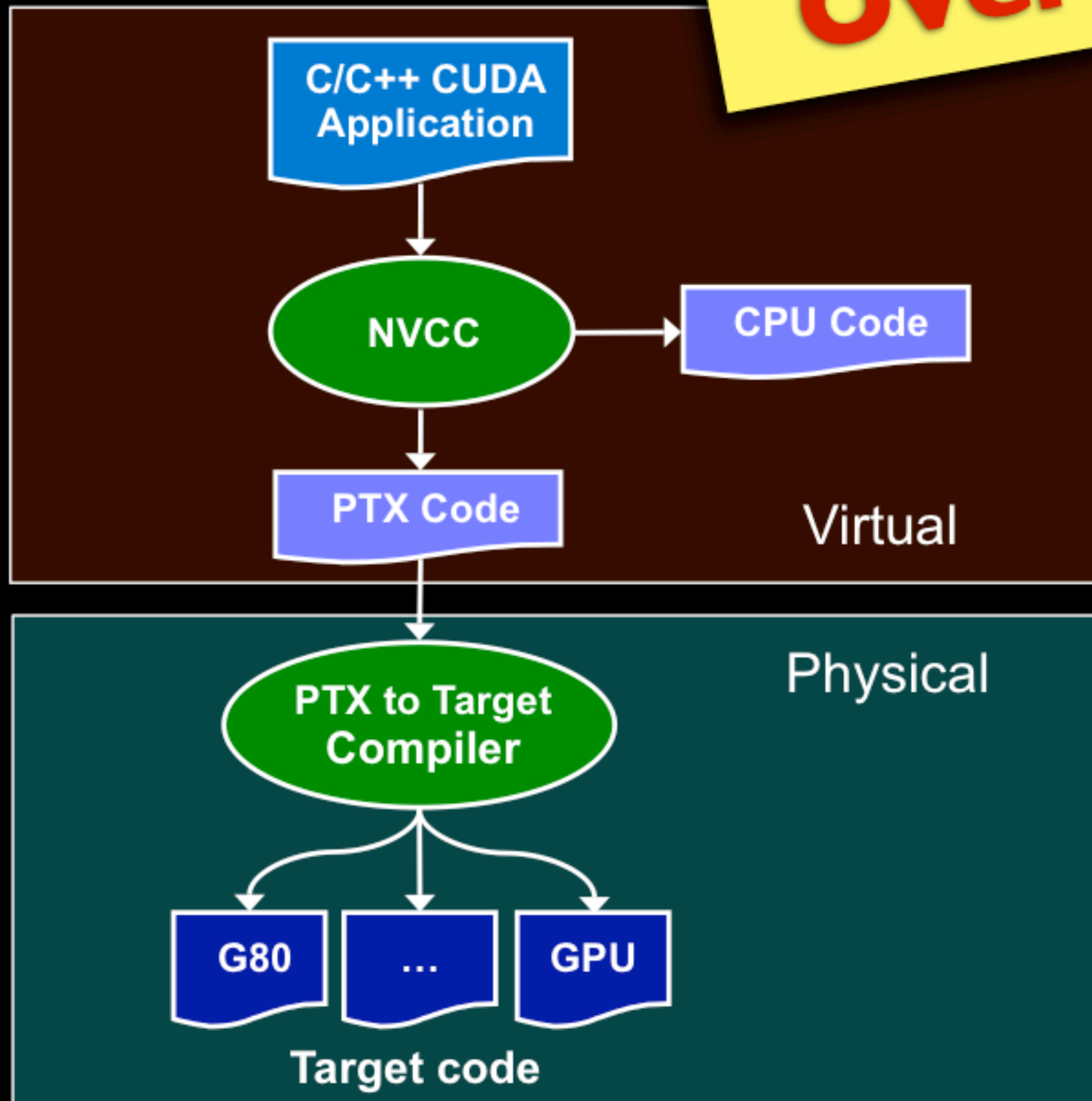
Standard C Compiler

GPU

CPU

Compiling CUDA Code

Overview

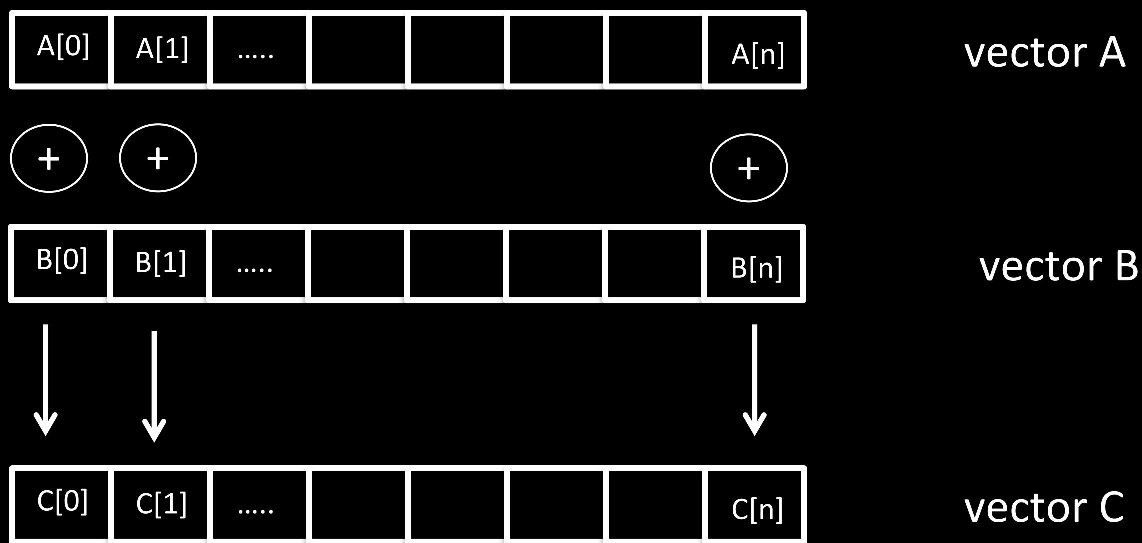


Key Parallel Abstractions in CUDA

Basics

- Trillions of lightweight threads
  - Simple decomposition model
- Hierarchy of concurrent threads
  - Simple execution model
- Lightweight synchronization of primitives
  - Simple synchronization model
- Shared memory model for thread cooperation
  - Simple communication model

Example:
Vector Addition Using CUDA



Vector addition is inherently a (data) parallel operation

Example: `vector_addition`

```
// compute vector sum h_C = h_A+ h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (int i = 0; i < N; i++)
      C[i] = h_A[i]+ h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, h_C
     // I/O to head h_A, h_B, N
    vecAdd(h_a, h_b, h_c, N);
}
```

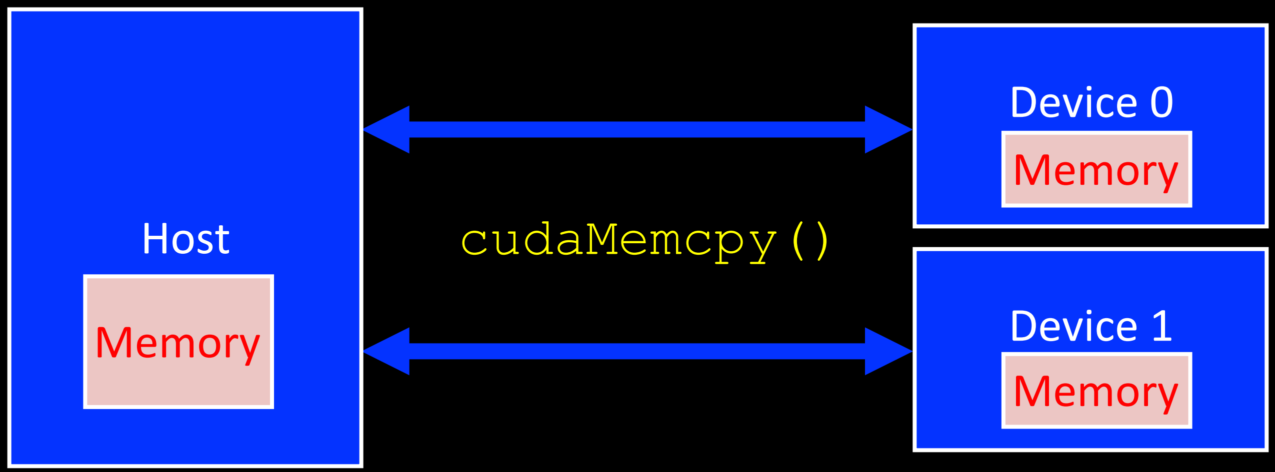
Now move the work to Device

```
#include <cuda.h>

void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
   int size = n * sizeof(float);
   float *A_d, *B_d, *C_d;

   1. // Allocate device memory for A, B, C
      // copy A and B to device memory
   2. // Kernel Launch code – to have the device
      // perform the actual vector addition
  3. // copy C from the device memory
     // Free the device vectors
}
```

Memory Model

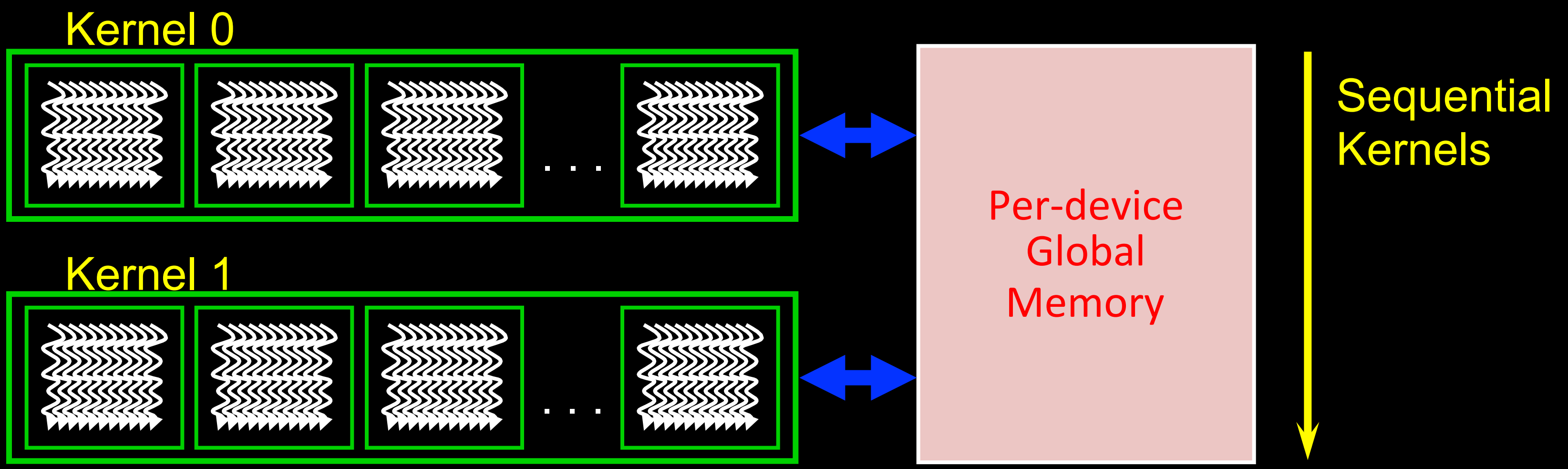
```
float *h_A, *h_B, *h_C
malloc(); // assign memory
free();   // free memory
```

```
float *d_A, *d_B, *d_C
cudaMalloc()
cudaFree()
```



We need to assign memory in the device (GPU) for the Variables that we wish to use in the device

Memory Model



All the blocks within the device have access to global memory of the device

Key Functions in CUDA

```
cudaMalloc()
      - Allocates object in the device global memory
      - Two parameters
         - Address of a pointer to the allocated object
         - Size of the allocated object in bytes

cudaFree()
      - Frees object from device global memory
      - Pointer to the object to  be freed

cudaMemcpy()
      - memory data transfer
      - requires four parameters
          - Pointer to destination
          - Pointer to source
          - Number of bytes to be copied
          - Type/Direction of transfer

Eg: cudaMemcpy(d_a, A, size, cudaMemcpyHostToDevice)
```

A more complete version of vecAdd()

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
   int size = n * sizeof(float);
   float *A_d, *B_d, *C_d;

   // Allocate Memory on Device
   cudaMalloc((void**) &d_A, size);
   cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void**) &d_B, size);
   cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
   cudaMalloc((void**) &d_C, size);

   // Kernel invocation code – shown in next slide

   // Copy result back to variable on Host
   cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

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

Launching the kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n) // Disable unused threads
        C[i] = A[i] + B[i];
}

void vecAdd(float* h_A, float* h_B, float* h_C, int n){
   // memory assignment statements omitted

   vecAddKernel <<< ceil(n/256.0), 256 >>> (d_A, d_B, d_C, n);
}
```

Number of blocks/grid

Number of threads/block

CUDA Keywords

```
__global__
```

	Executed on	Only callable from
__device__ float DeviceFunc()	device	device
__global__ float KernelFunc()	device	host
__host__ float HostFunc()	host	host

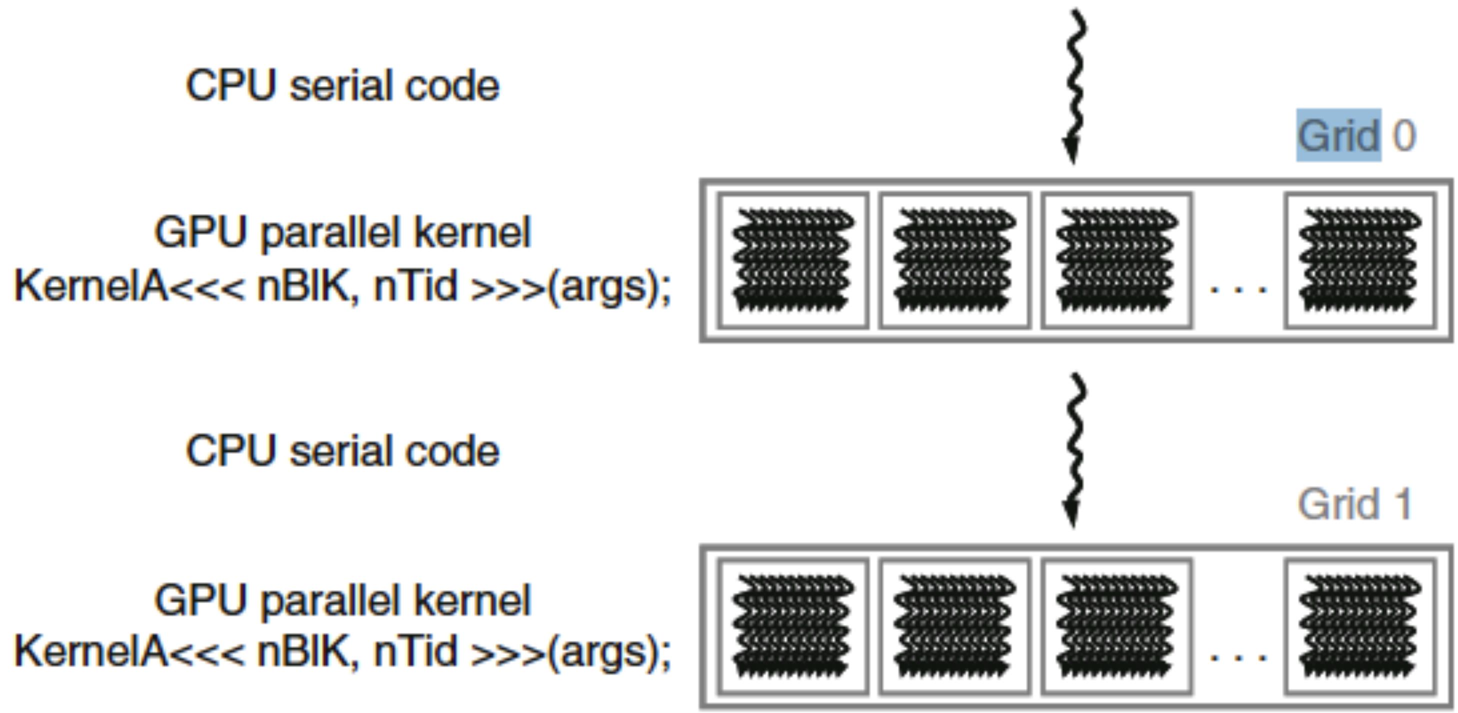
Executing Code on the GPU

Basics

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

Grid, Block, Thread, Kernel..

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```



All threads that are generated by a kernel during an invocation are collectively called a grid

FIGURE 3.2

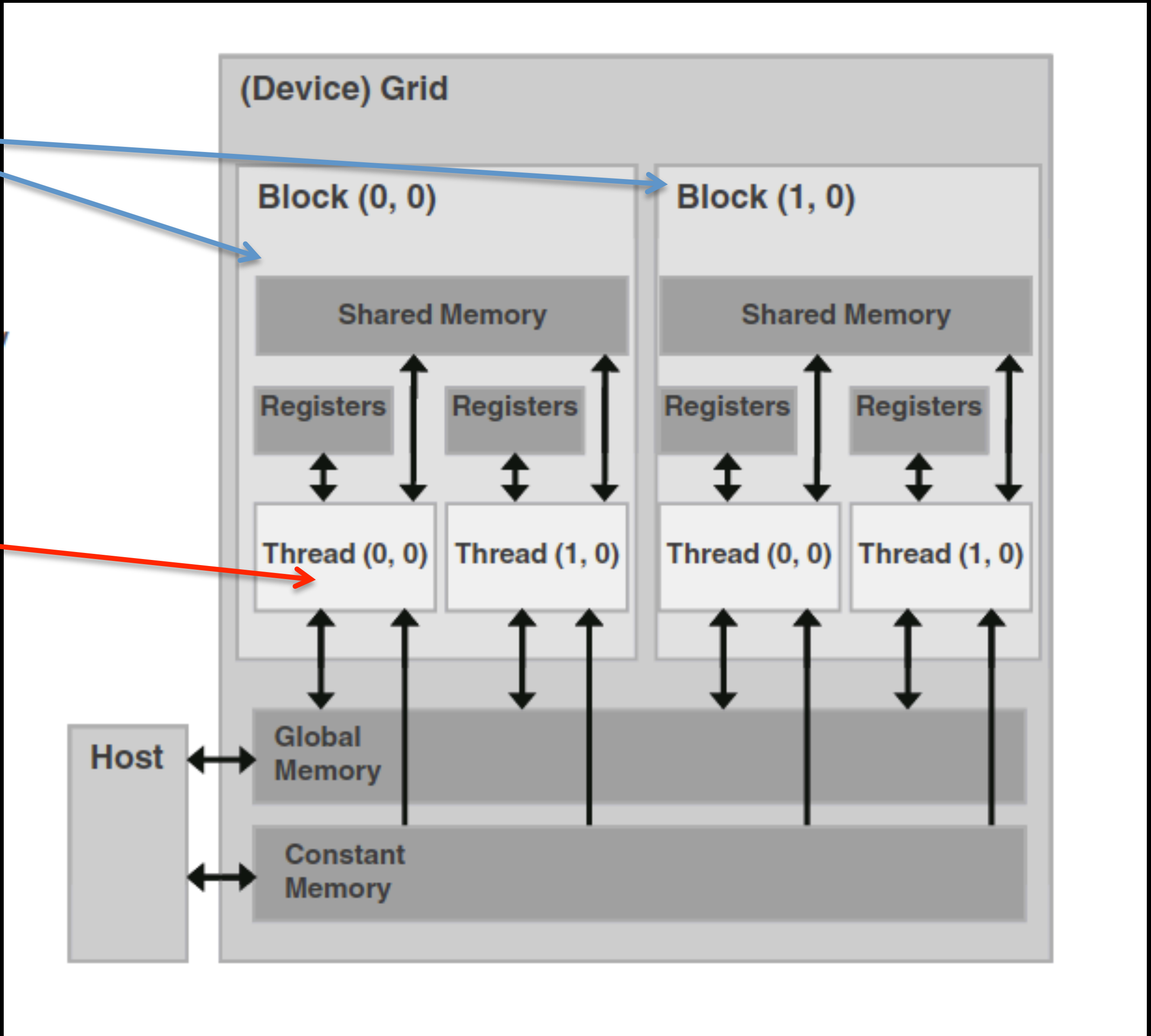
Execution of a CUDA program.

Grid, Block, Thread, Kernel..

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```

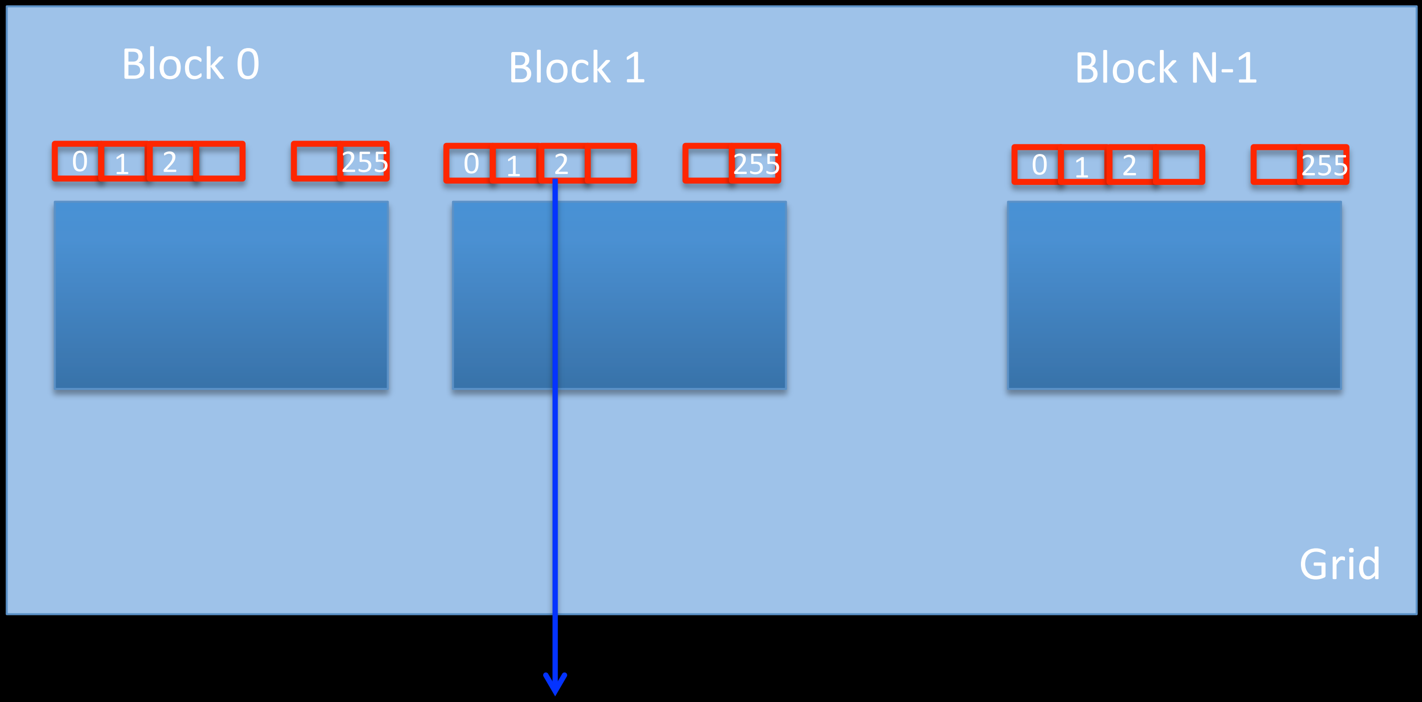
A grid consists
Of multiple blocks.
Each block has finite
Size (usually in
Increments of 32, since
32 threads form a
warp).

Each block can execute
many threads



Grid, Block, Thread, Kernel..

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```

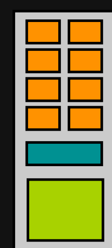
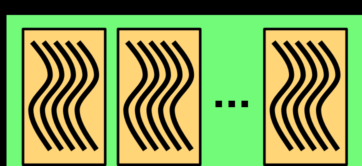
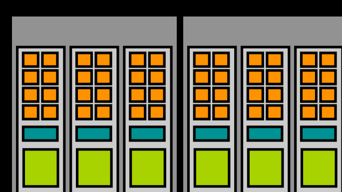


Block = 1, Block Dimension = 256, Thread id = 2

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
 258  =       2      + 256 * 1
```

Execution Model

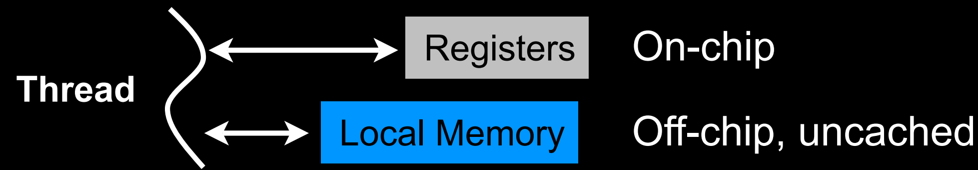
Basics

Software	Hardware	
 Thread	 Thread Processor	Threads are executed by thread processors
 Thread Block	 Multiprocessor	Thread blocks are executed on multiprocessors Thread blocks do not migrate Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)
 Grid	 Device	A kernel is launched as a grid of thread blocks Only one kernel can execute on a device at one time

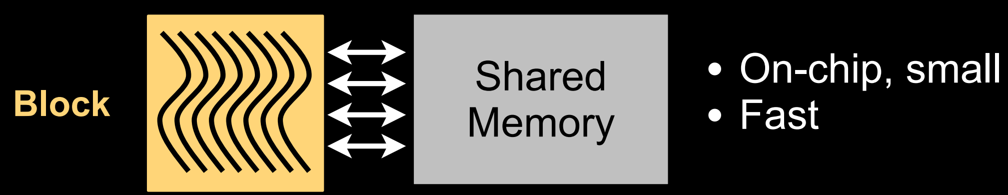
Kernel Memory Access

Basics

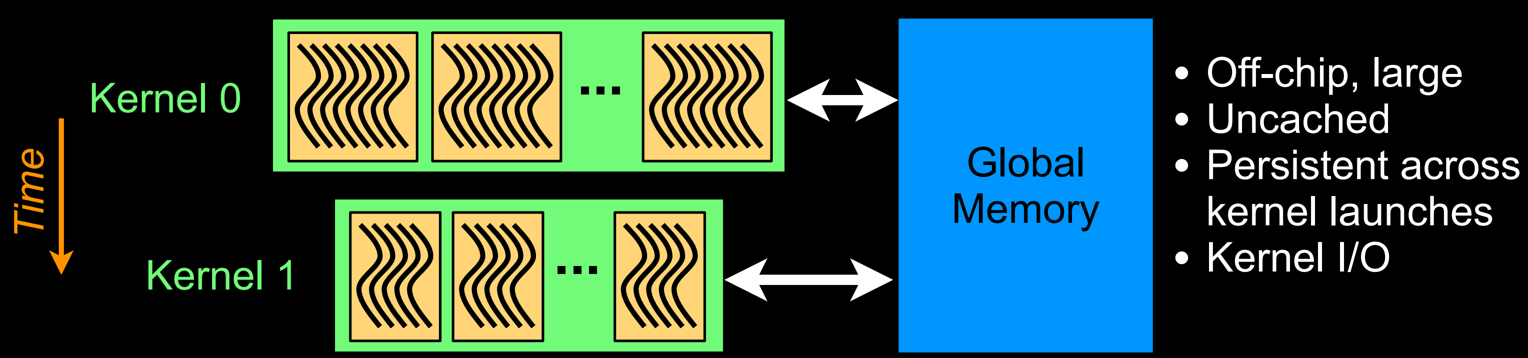
- Per-thread



- Per-block



- Per-device



Launching Kernels

Basics

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(…)
```

- Execution Configuration (“<<< >>>”)
  - dG - dimension and size of grid in blocks
    - Two-dimensional: x and y
    - Blocks launched in the grid: dG.x * dG.y
  - dB - dimension and size of blocks in threads:
    - Three-dimensional: x, y, and z
    - Threads per block: dB.x * dB.y * dB.z
  - Unspecified dim3 fields initialize to 1

A more complete version of vecAdd()

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

Device (GPU) code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    cudaMalloc((void**) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**) &d_C, size);

    vecAddKernel <<< ceil(n/256.0), 256 >>> (d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

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

Launching the kernel code

Host (CPU) code

Error Handling in CUDA

```
cudaMalloc((void**) &d_A, size);
```

In practice, this code should be surrounded by tests for error conditions and printout error messages.

```
cudaError_t err = cudaMalloc((void**) &d_A, size);
if(err != cudaSuccess){
  printf("%s in %s at line %d \n", cudaGetErrorString
  (err), __FILE__, __LINE__);

  exit(EXIT_FAILURE)
}
```

C for CUDA

- Philosophy: provide minimal set of extensions necessary to expose power
- Function qualifiers:

```
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

- Variable qualifiers:

```
__constant__ float my_constant_array[32];
__shared__   float my_shared_array[32];
```

- Execution configuration:

```
dim3 grid_dim(100, 50);  // 5000 thread blocks
dim3 block_dim(4, 8, 8); // 256 threads per block
my_kernel <<< grid_dim, block_dim >>> (...); // Launch kernel
```

- Built-in variables and functions valid in device code:

```
dim3 gridDim;   // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index
dim3 threadIdx; // Thread index
void __syncthreads(); // Thread synchronization
```

Kernel Variations and Output

```
__global__ void kernel( int *a )
{
   int idx = blockIdx.x*blockDim.x + threadIdx.x;
   a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
   int idx = blockIdx.x*blockDim.x + threadIdx.x;
   a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel( int *a )
{
   int idx = blockIdx.x*blockDim.x + threadIdx.x;
   a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Code executed on GPU

- C/C++ with some restrictions:
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - No recursion
  - No dynamic polymorphism
- Must be declared with a qualifier:
  - __global__ : launched by CPU,
    cannot be called from GPU must return void
  - __device__ : called from other GPU functions,
    cannot be called by the CPU
  - __host__ : can be called by CPU
  - __host__ and __device__ qualifiers can be combined
    - sample use: overloading operators

Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C functions
- Pointers are just addresses
  - Can't tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa

GPU Memory Allocation / Release

- Host (CPU) manages device (GPU) memory:
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)

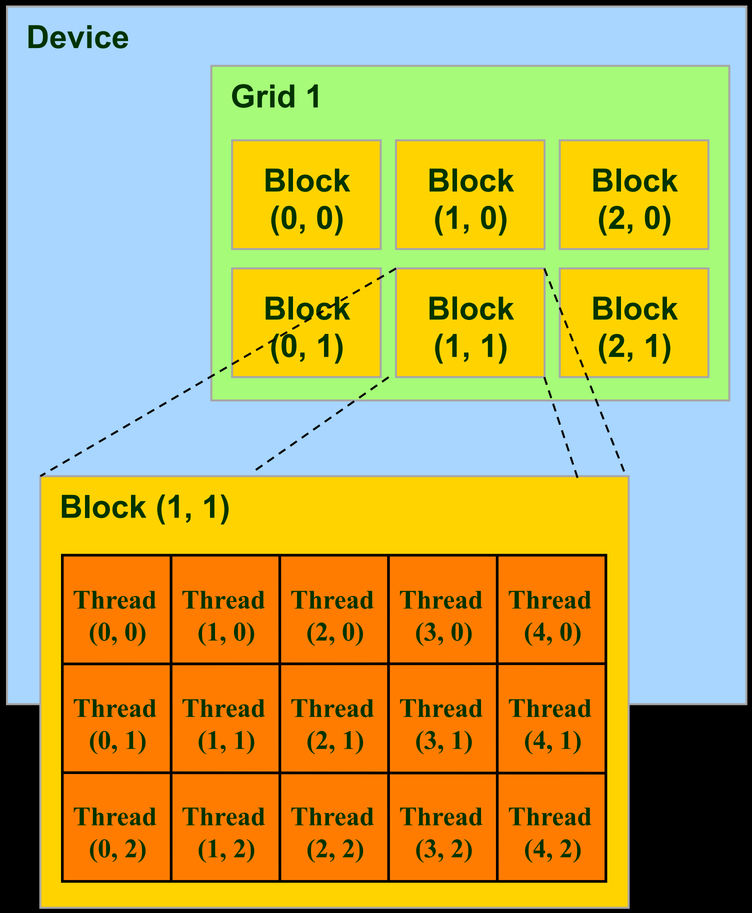
```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes,
  enum cudaMemcpyKind direction);
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- Non-blocking copies are also available

IDs and Dimensions

- Threads:
  - 3D IDs, unique within a block
- Blocks:
  - 2D IDs, unique within a grid
- Dimensions set at launch
  - Can be unique for each grid
- Built-in variables:
  - threadIdx, blockIdx
  - blockDim, gridDim



Kernel with 2D Indexing

```
__global__ void kernel( int *a, int dimx, int dimy )
{
   int ix   = blockIdx.x*blockDim.x + threadIdx.x;
   int iy   = blockIdx.y*blockDim.y + threadIdx.y;
   int idx = iy*dimx + ix;

   a[idx]  = a[idx]+1;
}
```

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- Independence requirement gives scalability

Host Synchronization

Basics

- 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

Host Synchronization Example

Basics

```
// copy data from host to device
cudaMemcpy(a_d, a_h, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
inc_gpu<<<ceil(N/(float)blocksize), blocksize>>>(a_d, N);

// run independent CPU code
run_cpu_stuff();

// copy data from device back to host
cudaMemcpy(a_h, a_d, numBytes, cudaMemcpyDeviceToHost);
```

nVISION 08
THE WORLD OF VISUAL COMPUTING

NVIDIA

Device Runtime Component: Synchronization Function

Basics

- `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

Host Runtime Component: Device Management

Basics

- Device **enumeration**
  - `cudaGetDeviceCount(), cudaGetDeviceProperties()`
- Device **selection**
  - `cudaChooseDevice(), cudaSetDevice()`

```
>  ~/NVIDIA_CUDA_SDK/bin/linux/release/deviceQuery
There is 1 device supporting CUDA

Device 0: "Quadro FX 5600"
  Major revision number:                         1
  Minor revision number:                         0
  Total amount of global memory:                 1609891840 bytes
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1350000 kilohertz
```

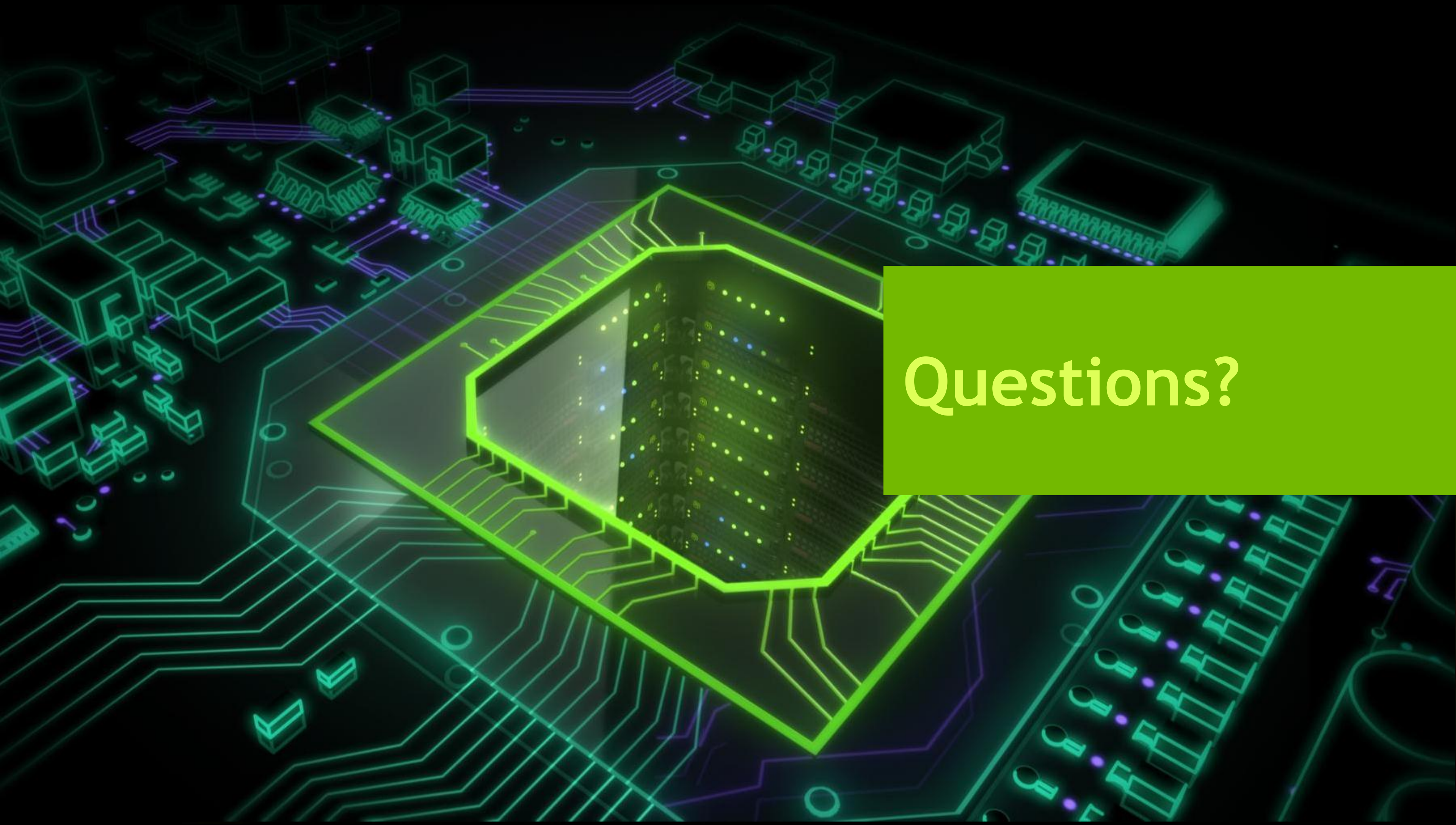
Host Runtime Component: Memory Management

Basics

- Two kinds of memory:
  - Linear memory: accessed through 32-bit pointers
  - CUDA arrays:
    - opaque layouts with dimensionality
    - readable only through texture objects
- Memory allocation
  - `cudaMalloc()`, `cudaFree()`, `cudaMallocPitch()`, `cudaMallocArray()`, `cudaFreeArray()`
- Memory copy
  - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToArray()`, `cudaMemcpyFromArray()`, etc. `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
- Memory addressing
  - `cudaGetSymbolAddress()`

Final Thoughts

- Parallel hardware is here to stay
- GPUs are massively parallel manycore processors
  - easily available and fully programmable
- Parallelism & scalability are crucial for success
- This presents many important research challenges
  - not to speak of the educational challenges



Questions?