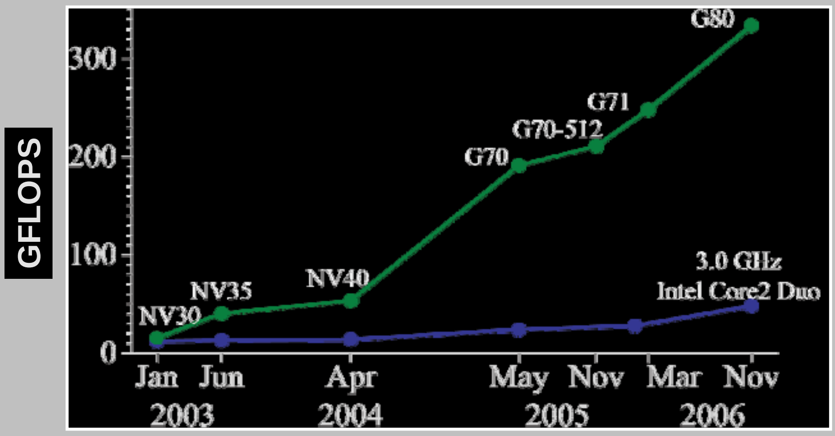
CUDA
(Compute Unified Device Architecture)

Mike Bailey

Oregon State University

History of GPU Performance vs. CPU Performance



Source: NVIDIA

G80 = GeForce 8800 GTX
G71 = GeForce 7900 GTX
G70 = GeForce 7800 GTX
NV40 = GeForce 6800 Ultra
NV35 = GeForce FX 5950 Ultra
NV30 = GeForce FX 5800

Why are GPUs Outpacing CPUs?

Due to the nature of graphics computations, GPU chips are customized to handle streaming data. This means that the data is already sequential, or cache-coherent, and thus the GPU chips do not need the significant amount of cache space that dominates CPU chips. The GPU die real estate can then be re-targeted to produce more processing power.

For example. while Intel and AMD are now shipping CPU chips with 4 cores, NVIDIA is shipping GPU chips with 128. Overall, in four years, GPUs have achieved a 17.5-fold increase in performance, a compound annual increase of 2.05X, which exceeds Moore's Law.

What is Cache Memory?

In computer science, a ***cache*** *is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (due to longer access time) or to compute, compared to the cost of reading the cache. In other words, a cache is a temporary storage area where frequently accessed data can be stored for rapid access. Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching or recomputing the original data, so that the average access time is shorter. Cache, therefore, helps expedite data access that the CPU would otherwise need to fetch from main memory.*

-- Wikipedia

How Can You Gain Access to that GPU Power?

1. Write a graphics display program (≥ 1985)

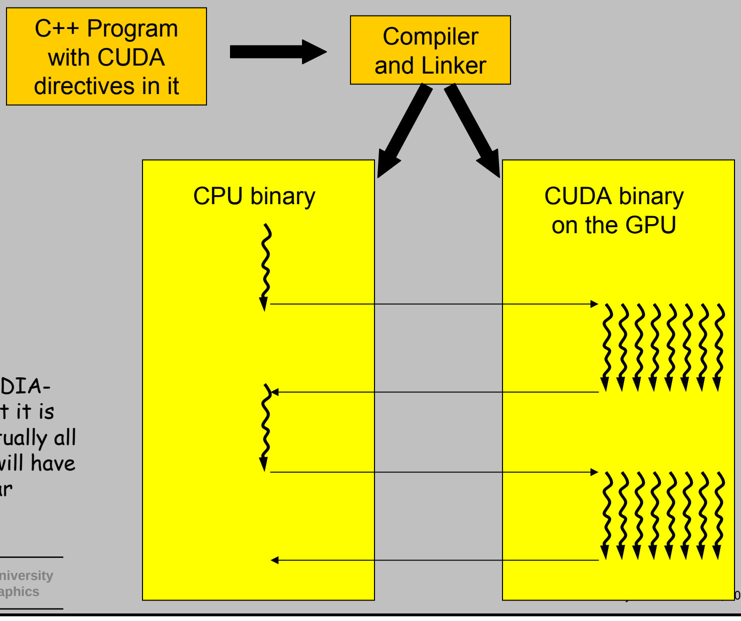
2. Write an application that looks like a graphics display program (≥ 2002)

3. Write in CUDA, which looks like C++ (≥ 2006)

CUDA Architecture

- The GPU has some number of MultiProcessors (MPs), depending on the model
- The NVIDIA 8800 comes in 2 models: either 12 or 16 MPs
- The NVIDIA 8600 has 4 MPs
- Each MP has 8 independent processors
- There are 16 KB of Shared Memory per MP, arranged in 16 banks
- There are 64 KB of Constant Memory

The CUDA Paradigm



CUDA is an NVIDIA-only product, but it is likely that eventually all graphics cards will have something similar

If GPUs have so Little Cache, how can they Execute General C++ Code Efficiently?

1. Multiple Multiprocessors
2. Threads – lots and lots of threads

• CUDA expects you to not just have a few threads, but to have *thousands* of them!

• All threads execute the same code (called the *kernel*), but operates on different data

• Each thread can determine which one it is

•Think of all the threads as living in a “pool”, waiting to be executed

• All processors start by grabbing a thread from the pool

• When a thread gets blocked somehow (a memory access, waiting for information from another thread, etc.), the processor quickly returns the thread to the pool and grabs another one to work on.

• This thread-swap happens within a single cycle

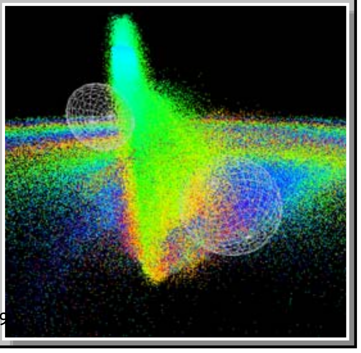
A full memory access requires 200 instruction cycles to complete

So, the Trick is to Break your Problem into Many, Many Small Pieces

Particle Systems are a great example.

1. Have one thread per *each particle*.
2. Put all of the initial parameters into an array in GPU memory.
3. Tell each thread what the current Time is.
4. Each thread then computes its particle's position, color, etc. and writes it into arrays in GPU memory.
5. The CPU program then initiates drawing of the information in those arrays.

Note: once setup, the data never leaves GPU memory



Ben Weiss, CS 519

Oregon State University
Computer Graphics

Organization: Threads are Arranged in Blocks

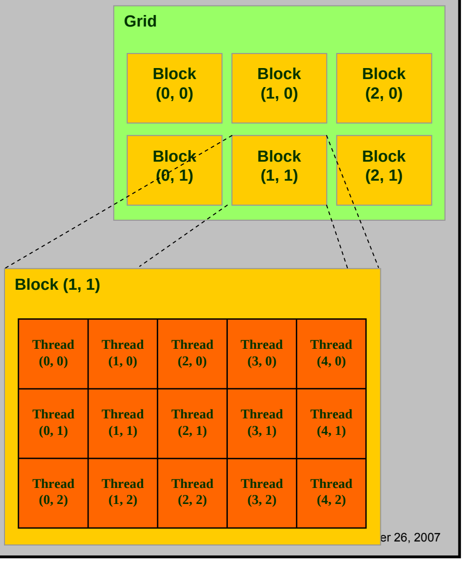
- A Thread Block has:
  - Size: 1 to 512 concurrent threads
  - Shape: 1D, 2D, or 3D (really just a convenience)
- Threads have *Thread ID* numbers within the Block
- The program uses these Thread IDs to select work and pull data from memory
- Threads share data and synchronize while doing their share of the work
- A *Thread Block* is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate

Oregon State University
Computer Graphics

mjb – November 26, 2007

Organization: Blocks are Arranged in Grids

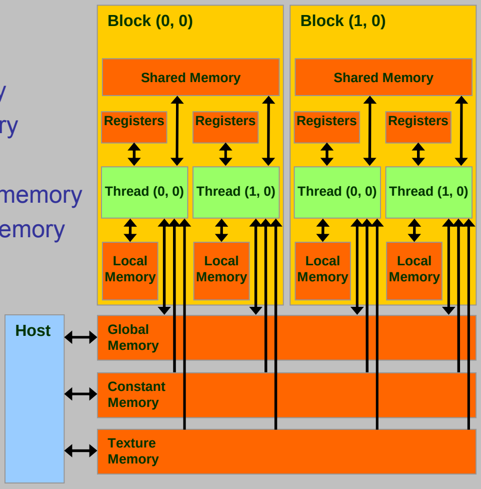
A CUDA program is organized as a *Grid of Thread Blocks*



Threads Can Access Various Types of Storage

- Each thread has access to:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read-only per-grid constant memory
  - Read-only per-grid texture memory

- The CPU can read and write global, constant, and texture memories



Rules

• You can have at most 512 Threads per Block

• Threads can share memory with the other Threads in the same Block

• Threads can synchronize with other Threads in the same Block

• Global, Constant, and Texture memory is accessible by all Threads in all Blocks

• Each Thread has registers and local memory

• Each Block can use at most 8,192 registers, divided equally among all Threads

• You can be executing up to 8 Blocks and 768 Threads simultaneously per MP

• A Block is run on only one MP (i.e., cannot switch to another MP)

• A Block can be run on any of the 8 processors of its MP

Types of CUDA Functions

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	GPU	GPU
`__global__ void KernelFunc()`	GPU	CPU
`__host__ float HostFunc()`	CPU	CPU

`__global__` defines a kernel function – it must return `void`

Different Types of CUDA Memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + CPU
Constant	Off-chip	Yes	Read	All threads + CPU
Texture	Off-chip	Yes	Read	All threads + CPU

Types of CUDA Variables

- Declarations
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
- Runtime API
  - Memory, symbol, execution management
- CUDA function launch

```
__device__ float filter[N];

__global__ void convolve (float *image)
{

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads();
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

A CUDA Thread can Query Where it Fits in its “Community”

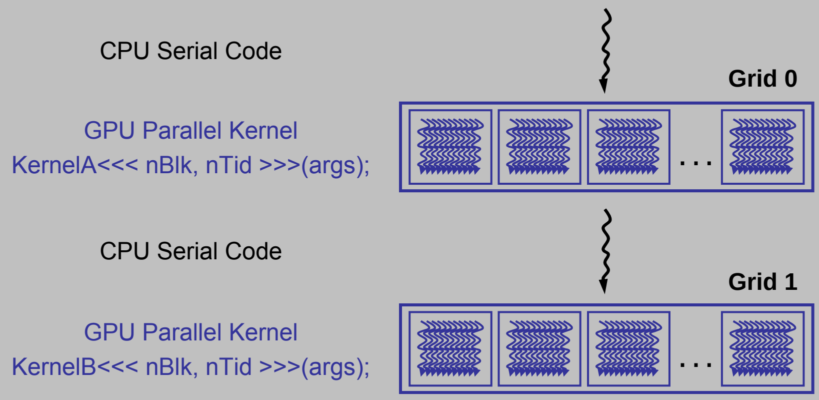
- `dim3 blockDim;`
  - Dimensions of the block in threads
- `dim3 threadIdx;`
  - Thread index within the block
- `dim3 gridDim;`
  - Dimensions of the grid in blocks (`gridDim.z` is not used)
- `dim3 blockIdx;`
  - Block index within the grid

Oregon State University
Computer Graphics

mjb – November 26, 2007

The CPU Invokes the CUDA Kernel using a Special Syntax

CUDAFunction<<< NumBlocks, NumThreadsPerBlock >>>(arg1, arg2, …)



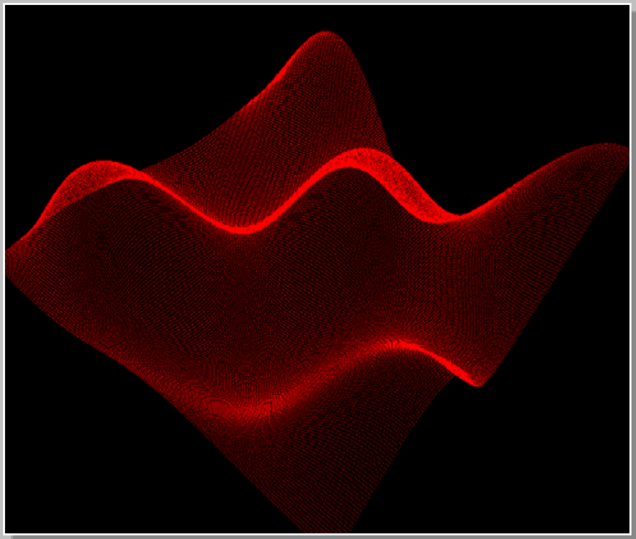
Oregon State University
Computer Graphics

mjb – November 26, 2007

Rules of Thumb

- OpenGL Buffer Objects can be mapped into CUDA space
- CUDA kernel is asynchronous
- Can call *cudaThreadSynchronize( )* from the application
- At least 16/12/4 Blocks must be run to fill the device
- The number of Blocks should be at least twice the number of MPs
- The number of Threads per Block should be a multiple of 64
- 192 or 256 are good numbers of Threads per Block

An Example: Recomputing Particle Positions



An Example: Static Image De-noising

Oregon State University
Computer Graphics

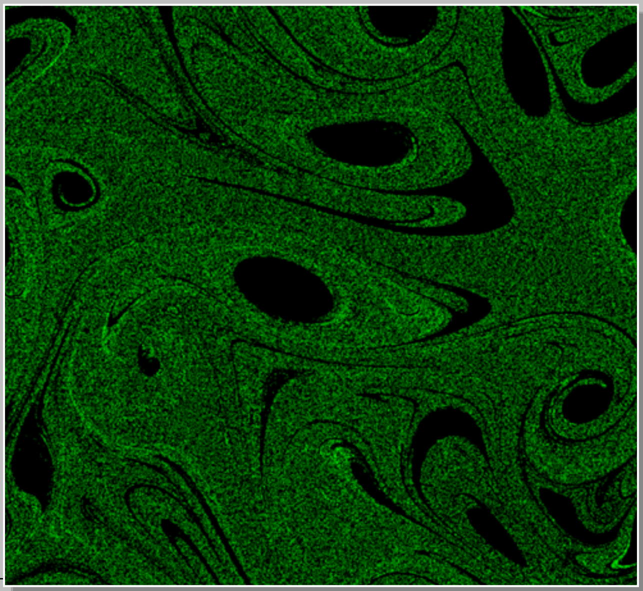
mjb – November 26, 2007

An Example: Dynamic Scene Blurring

Oregon State University
Computer Graphics

mjb – November 26, 2007

An Example: Realtime Fluids



Oregon State University
Computer Graphics

512x512 pixels, 76 FPS