

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/221032946>

NVIDIA CUDA software and GPU parallel computing architecture

Conference Paper · January 2007

DOI: 10.1145/1296907.1296909 · Source: DBLP

CITATIONS

184

READS

1,244

1 author:



David B. Kirk

NVIDIA

112 PUBLICATIONS 6,623 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Digital Inheritance [View project](#)



NVIDIA®

**NVIDIA CUDA Software and GPU
Parallel Computing Architecture**

David B. Kirk, Chief Scientist

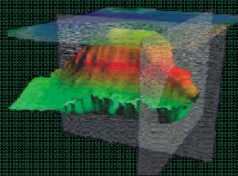
Outline



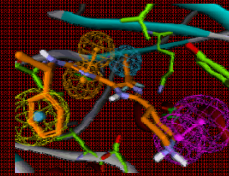
- Applications of GPU Computing
- CUDA Programming Model Overview
- Programming in CUDA – The Basics
- How to Get Started!

- Exercises / Examples Interleaved with Presentation Materials
 - Homework for later 😊

Future Science and Engineering Breakthroughs Hinge on Computing



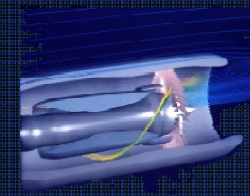
**Computational
Geoscience**



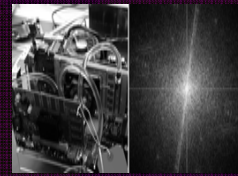
**Computational
Chemistry**



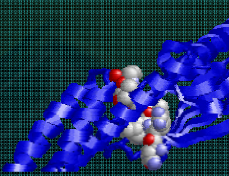
**Computational
Medicine**



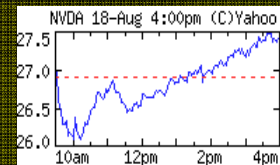
**Computational
Modeling**



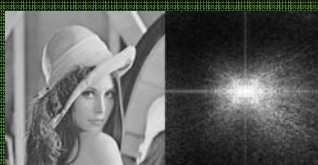
**Computational
Physics**



**Computational
Biology**



**Computational
Finance**



**Image
Processing**

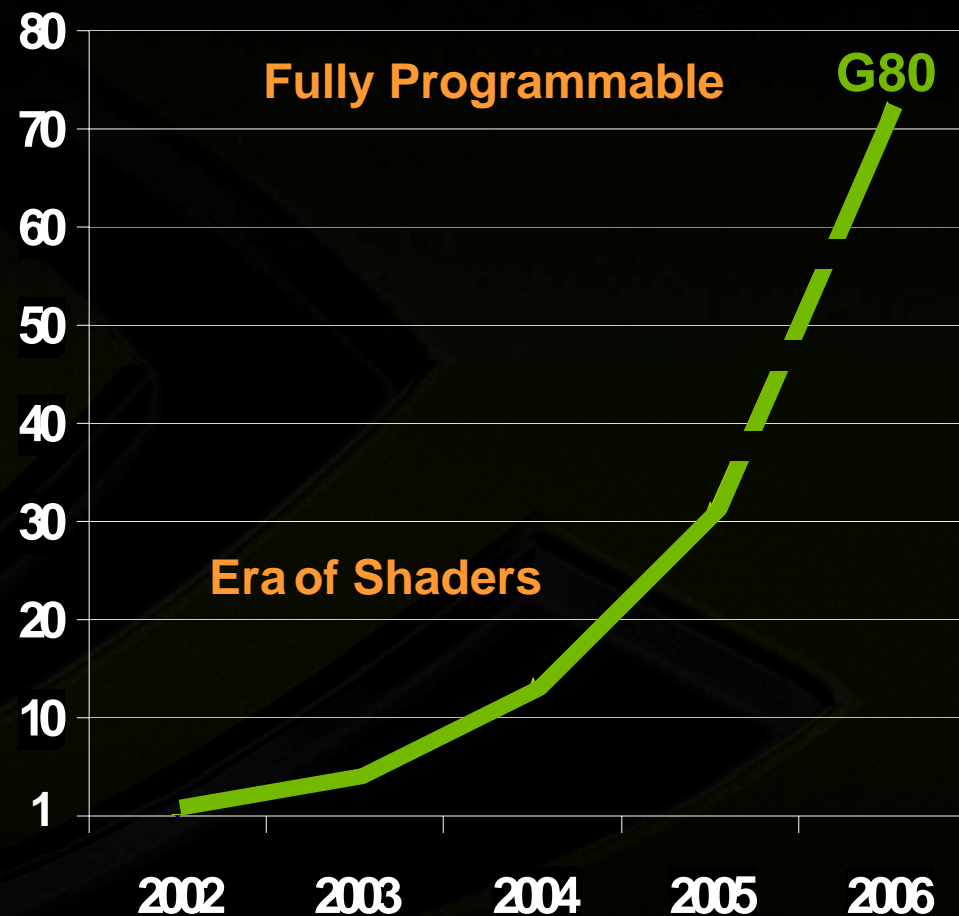
Faster is not “just Faster”



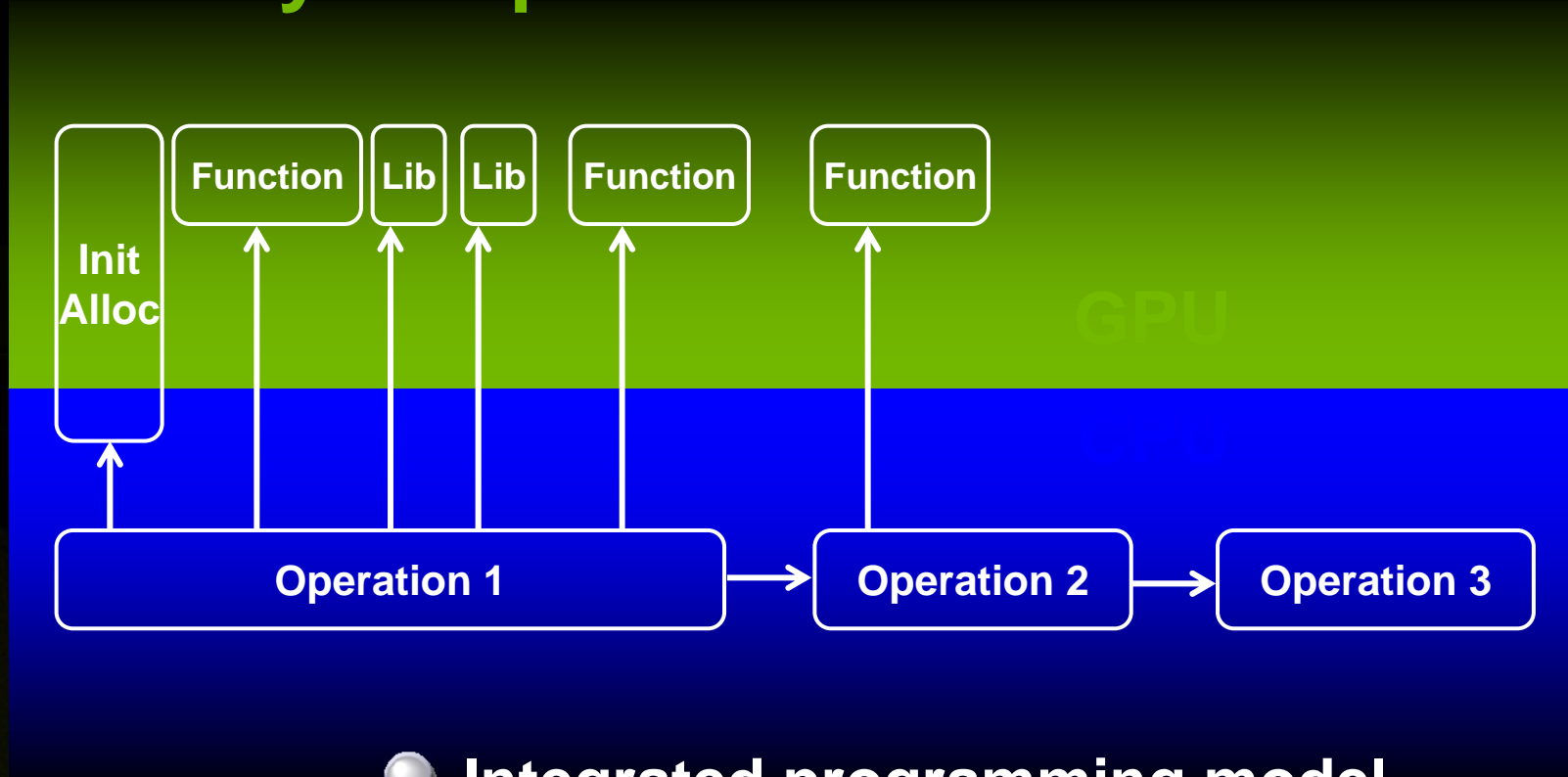
- **2-3X faster is “just faster”**
 - Do a little more, wait a little less
 - Doesn't change how you work
- **5-10x faster is “significant”**
 - Worth upgrading
 - Worth re-writing (parts of) the application
- **100x+ faster is “fundamentally different”**
 - Worth considering a new platform
 - Worth re-architecting the application
 - Makes new applications possible
 - Drives “time to discovery” and creates fundamental changes in Science

The GPU is a New Computation Engine

Relative
Floating Point
Performance



Closely Coupled CPU-GPU

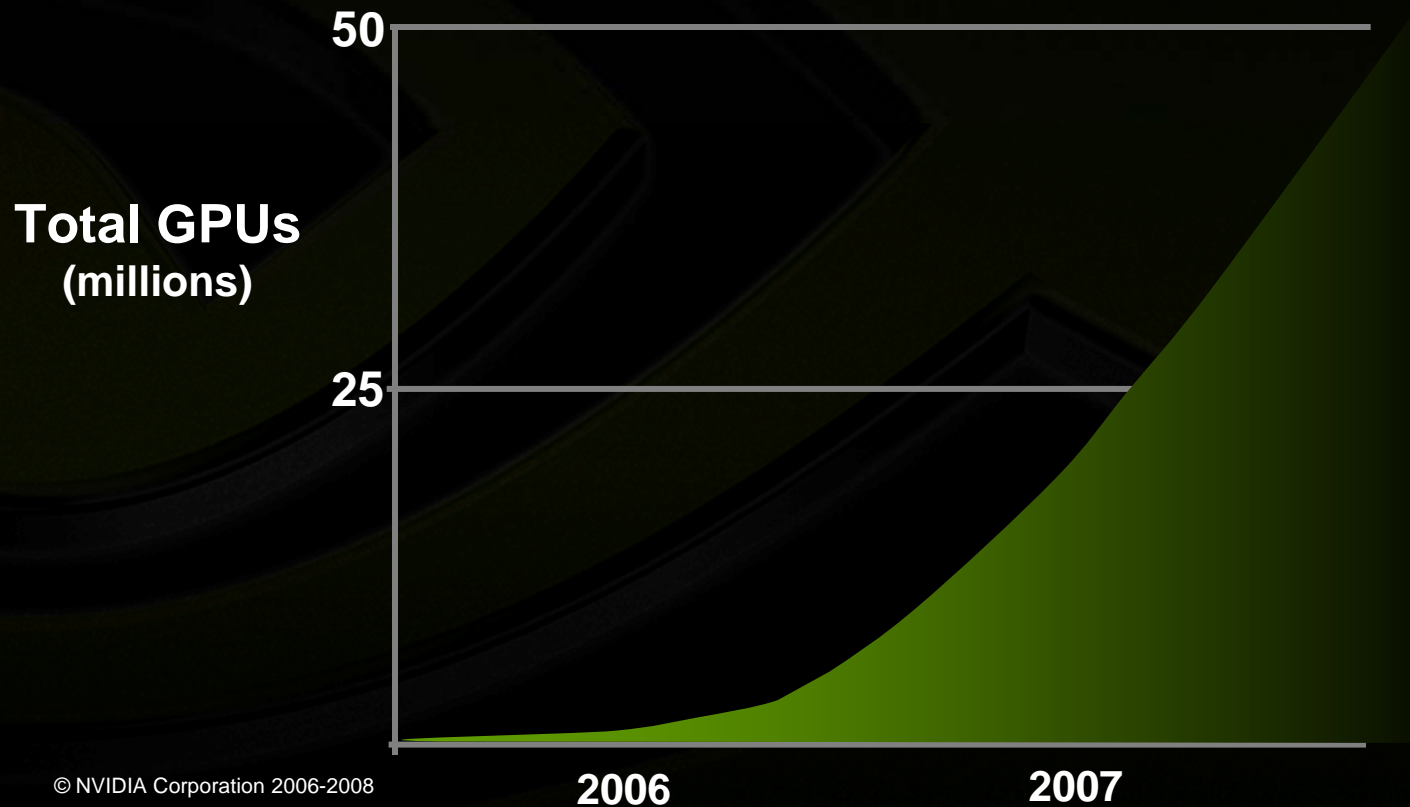


- Integrated programming model
- High speed data transfer – up to 3.2 GB/s
- Asynchronous operation
- Large GPU memory systems

Millions of CUDA-enabled GPUs



- Dedicated computing
- C on the GPU
- Servers through Notebook PCs



GeForce®

Entertainment



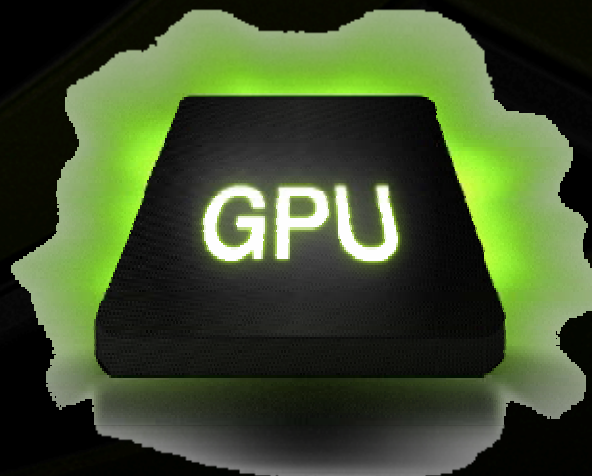
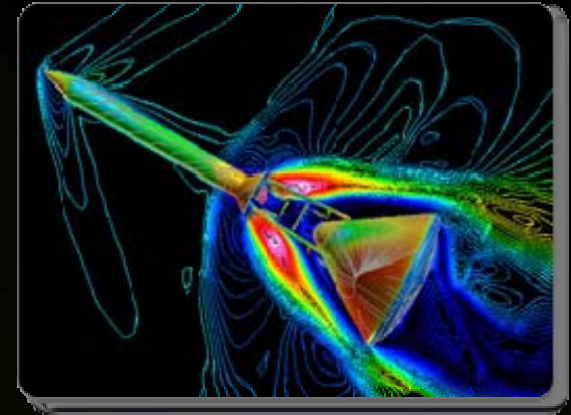
Quadro®

Design & Creation



Tesla™

High Performance Computing



VMD/NAMD Molecular Dynamics



- 240X speedup
- Computational biology

THEORETICAL and COMPUTATIONAL BIOPHYSICS GROUP
SIMULATIONS FOR MOLECULAR MODELING AND BIOPHYSICS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

NAMD

Scalable Molecular Dynamics

NAMD, recipient of a 2002 Gordon Bell Award, is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. Used on **Charm++ parallel objects**, NAMD **scales** to hundreds of processors on high-end parallel platforms and tens of processors on commodity **clusters** using gigabit ethernet. NAMD uses the popular molecular graphics program **VMD** for simulation setup and trajectory analysis, but is also file-compatible with AMBER, CHARMM, and X-PLOR. NAMD is distributed **free of charge** with source code. You can **build NAMD** yourself or download **binaries** for a wide variety of platforms. Our **tutorials** show you how to use NAMD and VMD for biomolecular modeling.

News: High performance computing in biology: Multimillion atom simulations of nanoscale systems. K.Y. Sanbonmatsu and C.-S. Tung. *Journal of Structural Biology*, 157:470-400, 2007.

News: Supercomputer Simulations May Pinpoint Causes of Parkinson's, Alzheimer's Diseases (SDSC article referring to NAMD simulations on Blue Gene/L, reported in Tsigelely et al., *FEBS Journal*, 274:1862-1877, 2007.)

Single search:

Parallel GPUs with Multithreading: 705 GFLOPS /w 3 GPUs

- One host thread is created for each CUDA GPU
- Threads are spawned and attach to their GPU based on their host thread ID
 - First CUDA call binds that thread's CUDA context to that GPU for life
 - Handling error conditions within child threads is dependent on the thread library and, makes dealing with any CUDA errors somewhat tricky, left as an exercise to the reader... ☺
- Map slices are computed cyclically by the GPUs
- Want to avoid false sharing on the host memory system
 - map slices are usually much bigger than the host memory page size, so this is usually not a problem for this application
- Performance of 3 GPUs is stunning!
- Power: 3 GPU test box consumes 700 watts running flat out

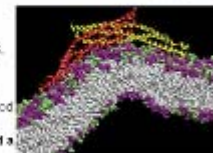
antipod
AMD
load VMD
parallel
ramming
memory

Spotlight: Step Up to the BAR Domain (Apr 2007)

PSC News Release: University of Utah chemist Gregory Voth and grad student Phil Blood are using PSC's Cray XT3 to tackle a basic question of embryology—the **blebbing** process by which cells absorb material from outside the cell by bonding their membrane to form a "vesicle" and engulf it. All animal cells depend on endocytosis, which involves various steps, but begins with curvature of the membrane.

BAR domains are a family of basket-shaped proteins shown to bend cellular membranes as if curves. Experiments suggest that BAR domains mold their concave surface to a section of membrane and induce a corresponding curvature. Voth and Blood undertook molecular dynamics simulations to look more closely. **With the XT3 they've been able to run efficiently, using software called NAMD, with as many as 1,024 processors.** "The XT3 has been amazing," says Blood. "We haven't found a hard limit on scaling up the number of processors."

They used TeraGrid systems at SDSC, NCSA and University of Chicago/Argonne to construct a model and to explore how long a stretch of membrane they needed for curvature to occur. Their final simulations used the XT3 to include the protein with a 50-nanometer length of membrane—probably the longest patch of

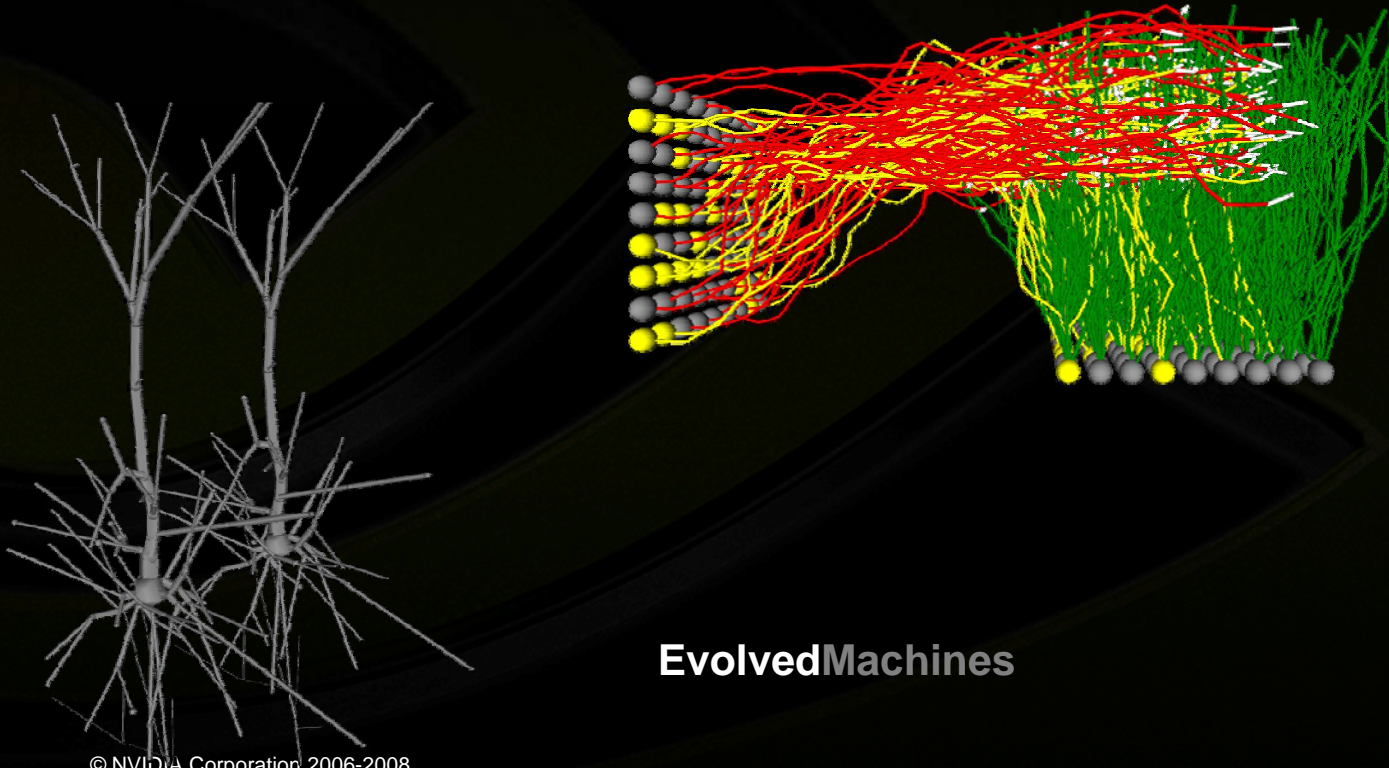


<http://www.ks.uiuc.edu/Research/vmd/projects/ece498/lecture/>

Evolved**Machines**

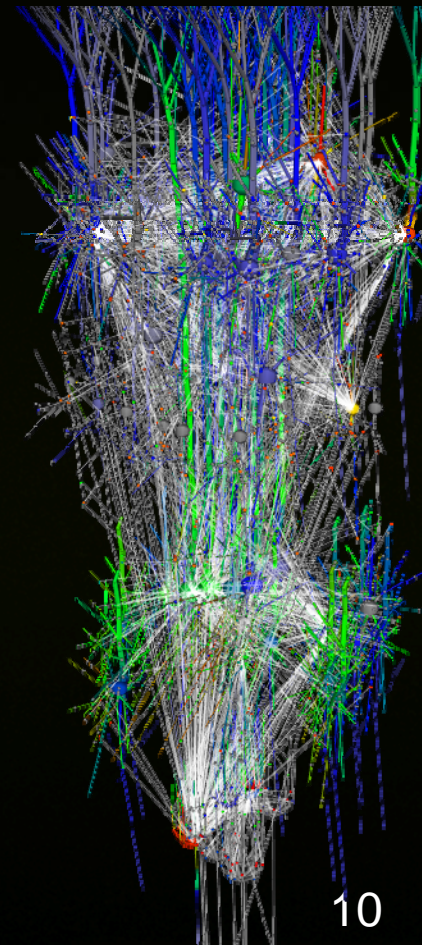


- Simulate the brain circuit
- Sensory computing: vision, olfactory
- 130X Speed up



Evolved**Machines**

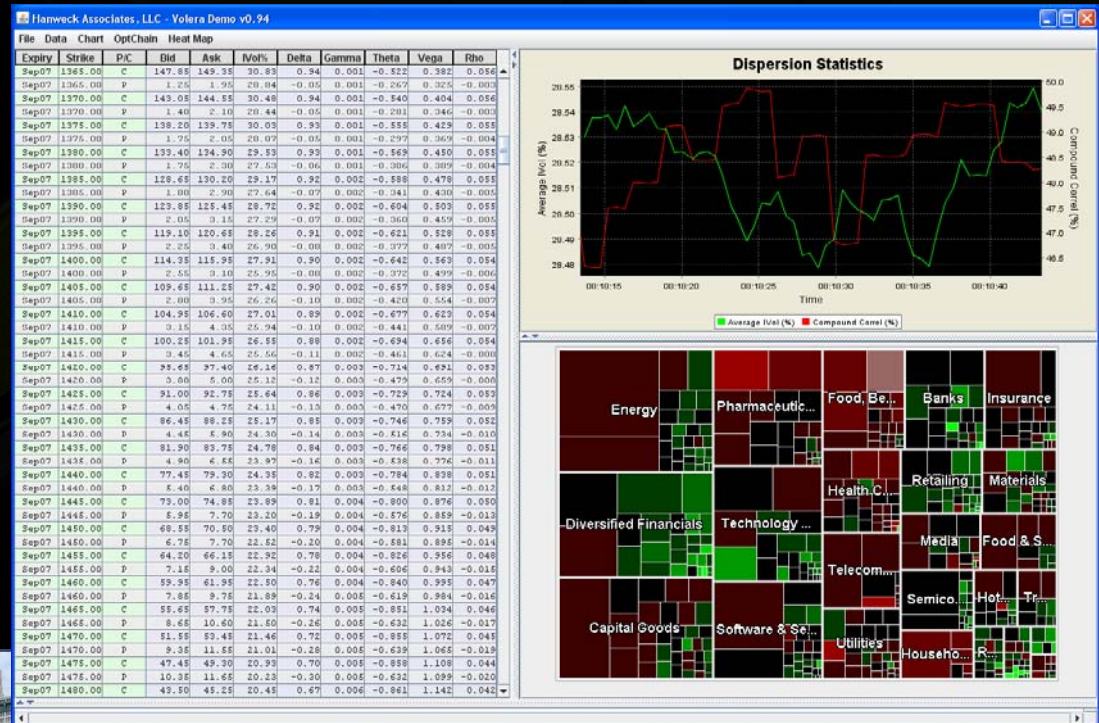
© NVIDIA Corporation 2006-2008



Hanweck Associates



- VOLERA, real-time options implied volatility engine
- Accuracy results with SINGLE PRECISION
- Evaluate all U.S. listed equity options in <1 second



www.hanweckassoc.com

LIBOR APPLICATION:

Mike Giles and Su Xiaoke

Oxford University Computing Laboratory

- LIBOR Model with portfolio of swaptions
- 80 initial forward rates and 40 timesteps to maturity
- 80 Deltas computed with adjoint approach

	No Greeks		Greeks	
Intel Xeon	18.1s	-	26.9s	-
ClearSpeed Advance 2 CSX600	2.9s	6x	6.4s	4x
NVIDIA 8800 GTX	0.045s	400x	0.18s	149x

"The performance of the CUDA code on the 8800 GTX is exceptional"
-Mike Giles

Source codes and papers available at:

<http://web.comlab.ox.ac.uk/oucl/work/mike.giles/hpc>

Manifold 8 GIS Application



From the Manifold 8 feature list:

... applications fitting CUDA capabilities that might have taken **tens of seconds or even minutes** can be accomplished in **hundredths of seconds**. ... **CUDA will clearly emerge to be the future of almost all GIS computing**

From the user manual:

"NVIDIA CUDA ... could well be the most revolutionary thing to happen in computing since the invention of the microprocessor



nbody Astrophysics



Astrophysics research

1 GF on standard PC

300+ GF on GeForce 8800GTX

Faster than GRAPE-6Af custom simulation computer

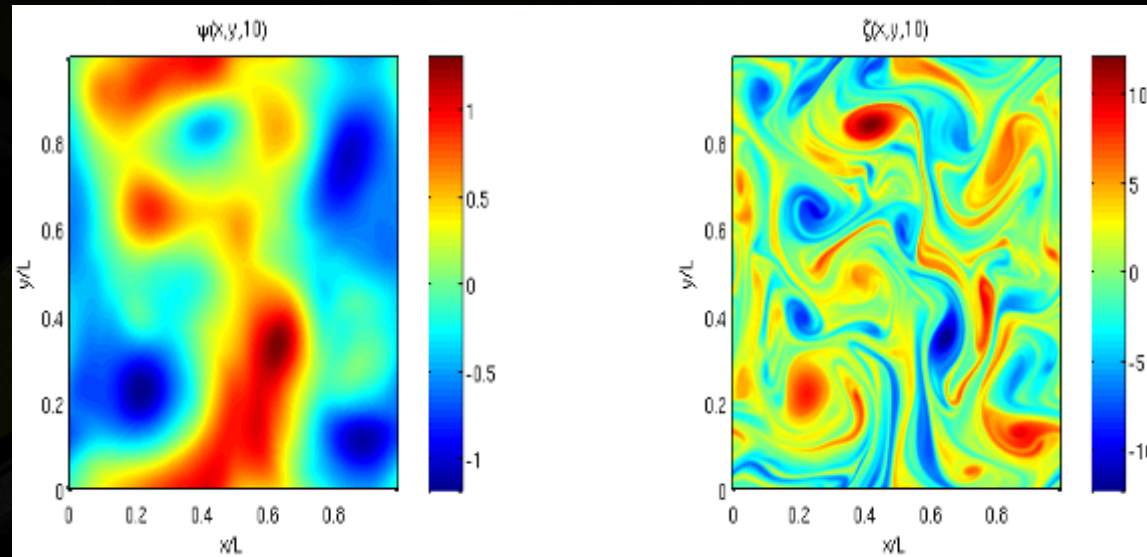
<http://progrape.jp/cs/>

Matlab: Language of Science



17X with MATLAB CPU+GPU

http://developer.nvidia.com/object/matlab_cuda.html



Pseudo-spectral simulation of 2D Isotropic turbulence

http://www.amath.washington.edu/courses/571-winter-2006/matlab/FS_2Dturb.m



NVIDIA®

CUDA Programming Model Overview

GPU Computing



- **GPU is a massively parallel processor**
 - **NVIDIA G80: 128 processors**
 - **Support thousands of active threads (12,288 on G80)**
- **GPU Computing requires a programming model that can efficiently express that kind of parallelism**
 - **Most importantly, data parallelism**
- **CUDA implements such a programming model**

CUDA Kernels and Threads



- Parallel portions of an application are executed on the device as **kernels**
 - One **kernel** is executed at a time
 - Many threads execute each **kernel**
- Differences between CUDA and CPU threads
 - CUDA threads are extremely lightweight
 - Very little creation overhead
 - Instant switching
 - CUDA uses 1000s of threads to achieve efficiency
 - Multi-core CPUs can use only a few

Definitions:

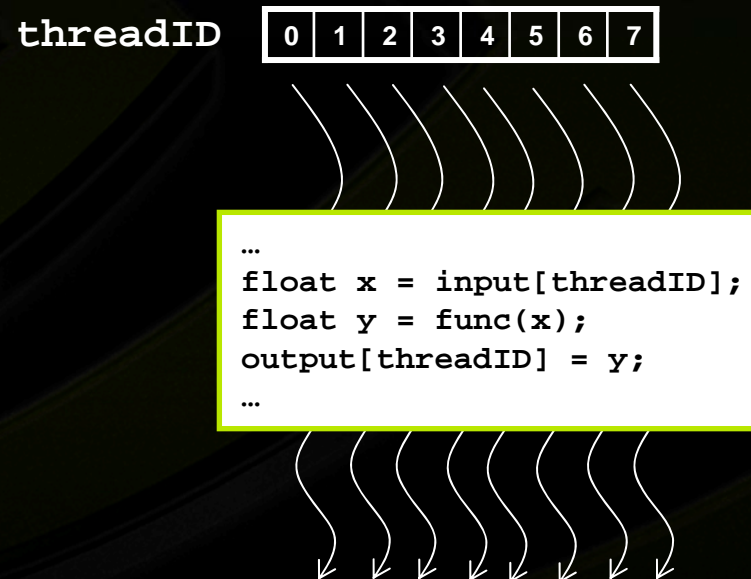
Device = GPU; *Host* = CPU

Kernel = function that runs on the device

Arrays of Parallel Threads



- A CUDA kernel is executed by an array of threads
 - All threads run the same code
 - Each thread has an ID that it uses to compute memory addresses and make control decisions



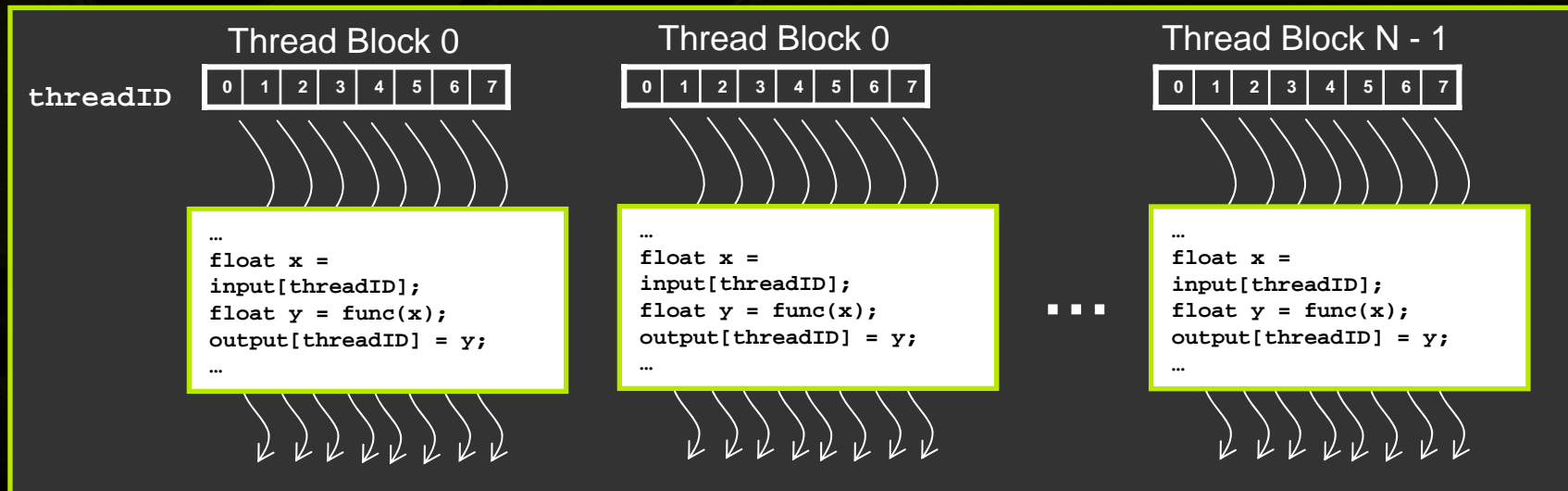
Thread Cooperation



- **The Missing Piece: threads may need to cooperate**
- **Thread cooperation is valuable**
 - **Share results to save computation**
 - **Synchronization**
 - **Share memory accesses**
 - **Drastic bandwidth reduction**
- **Thread cooperation is a powerful feature of CUDA**

Thread Blocks: Scalable Cooperation

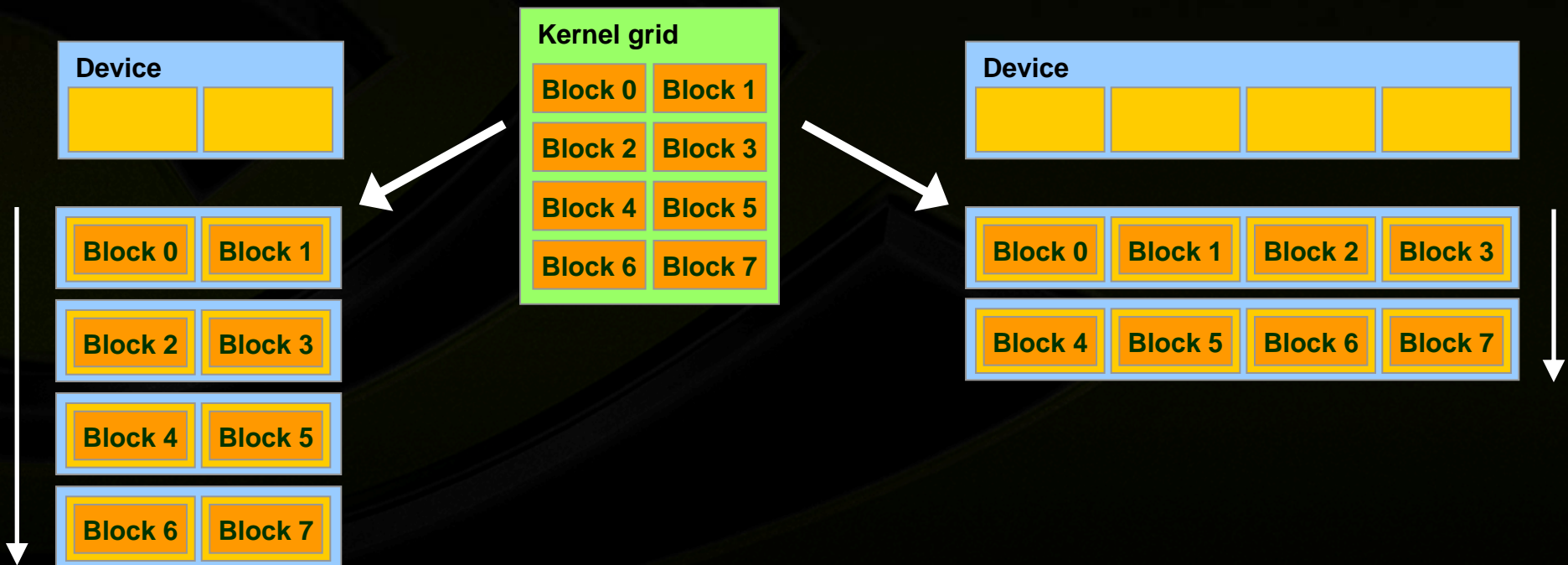
- Divide monolithic thread array into multiple blocks
 - Threads within a block cooperate via **shared memory**
 - Threads in different blocks cannot cooperate
- Enables programs to **transparently scale** to any number of processors!



Transparent Scalability



- Hardware is free to schedule thread blocks on any processor at any time
 - A kernel scales across any number of parallel multiprocessors

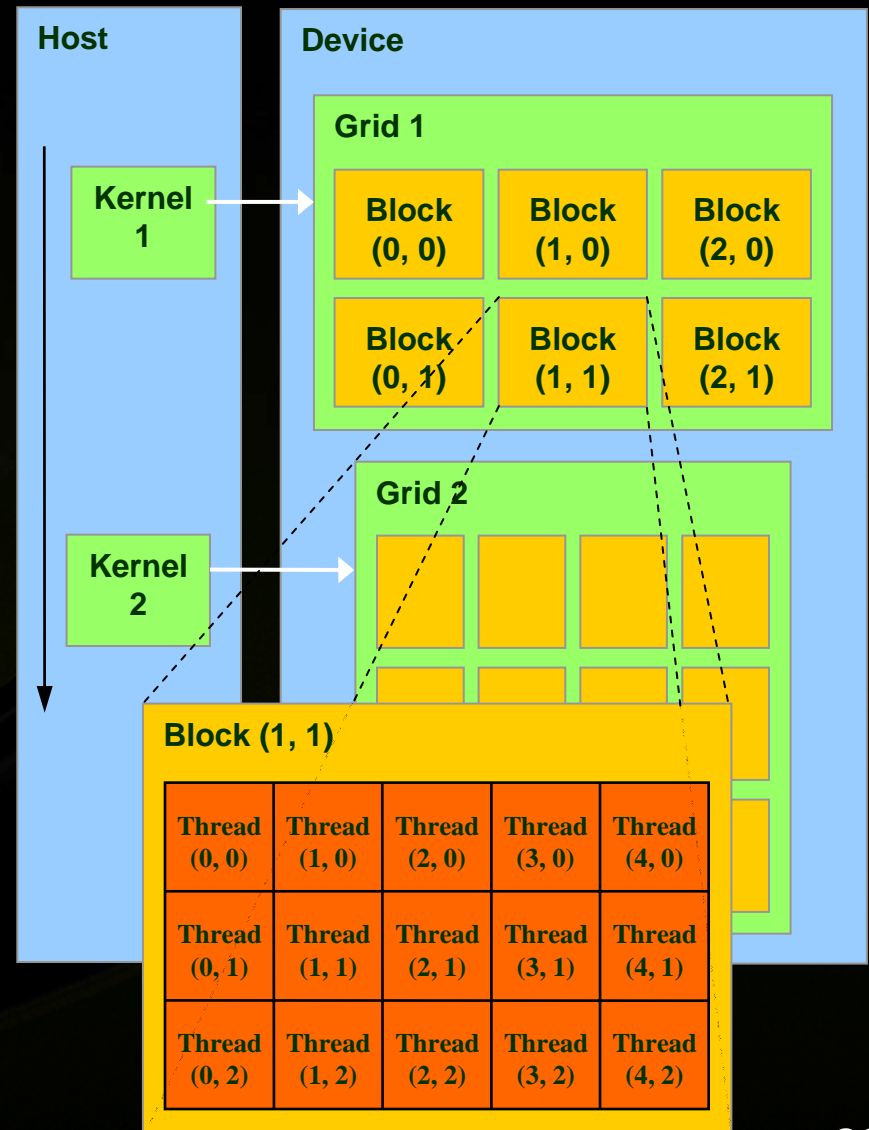


CUDA Programming Model



A kernel is executed by a **grid** of **thread blocks**

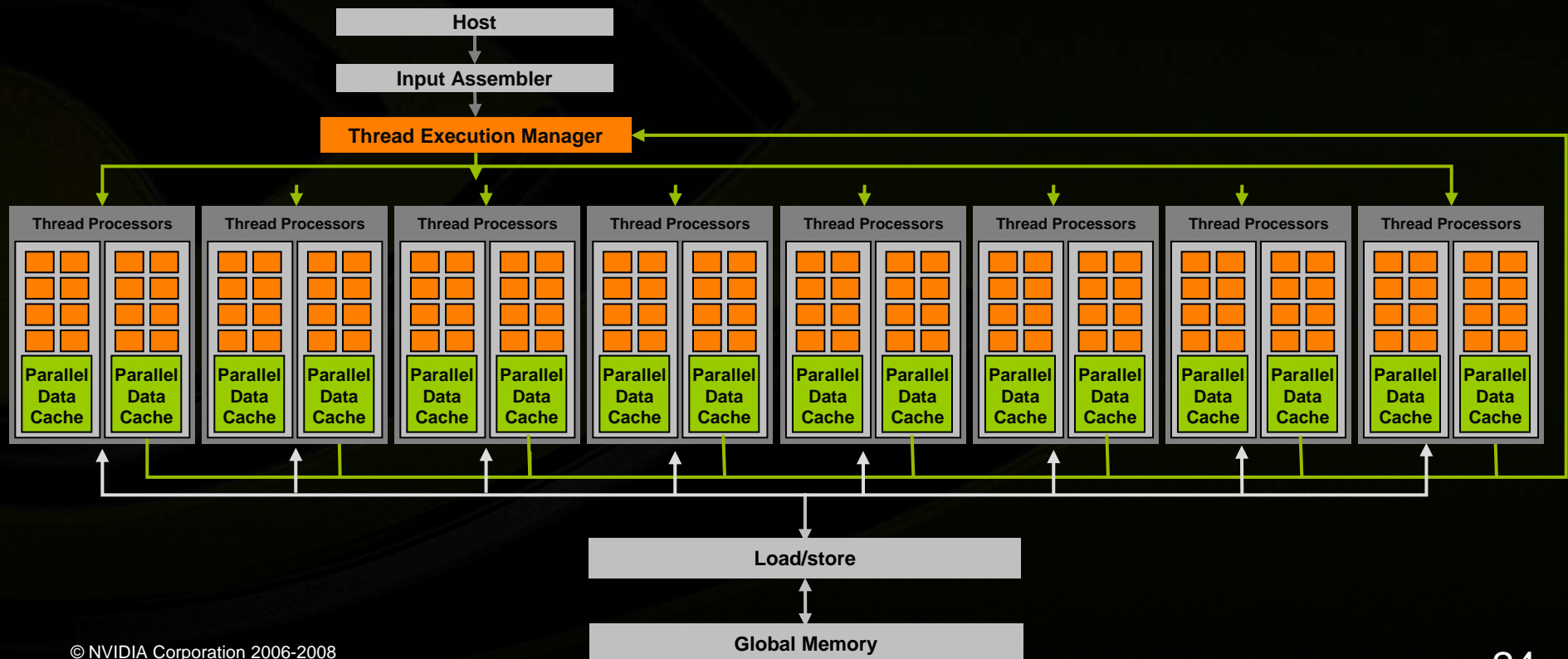
- A **thread block** is a batch of threads that can cooperate with each other by:
 - Sharing data through shared memory
 - Synchronizing their execution
- Threads from different blocks cannot cooperate



G80 Device



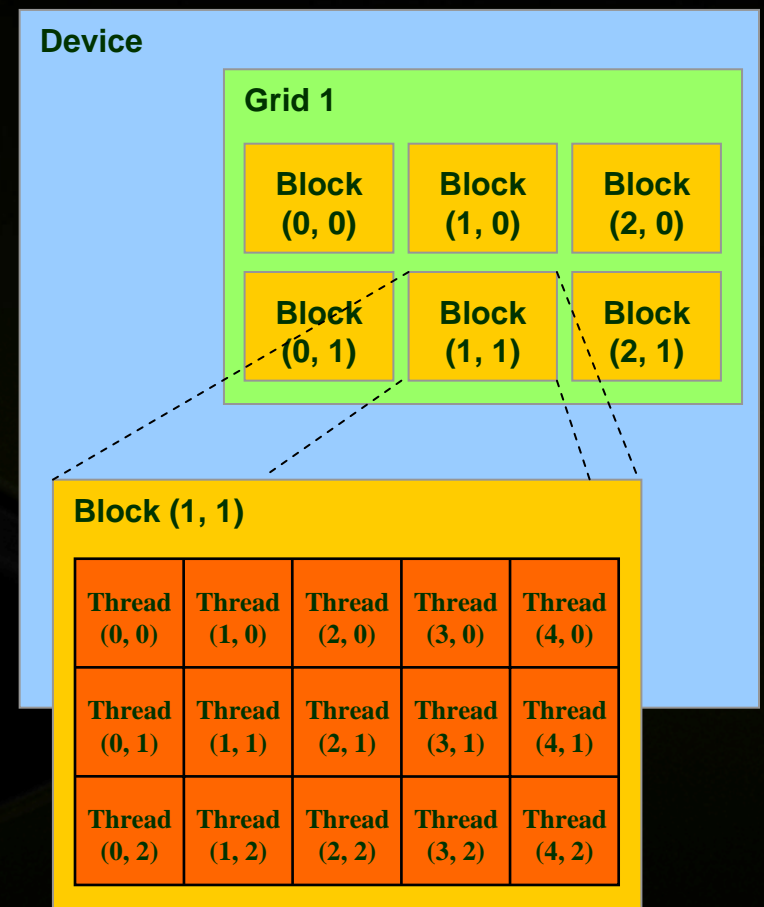
- Processors ■ execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors grouped into 16 Multiprocessors (SMs)
- Parallel Data Cache (Shared Memory) enables thread cooperation



Thread and Block IDs



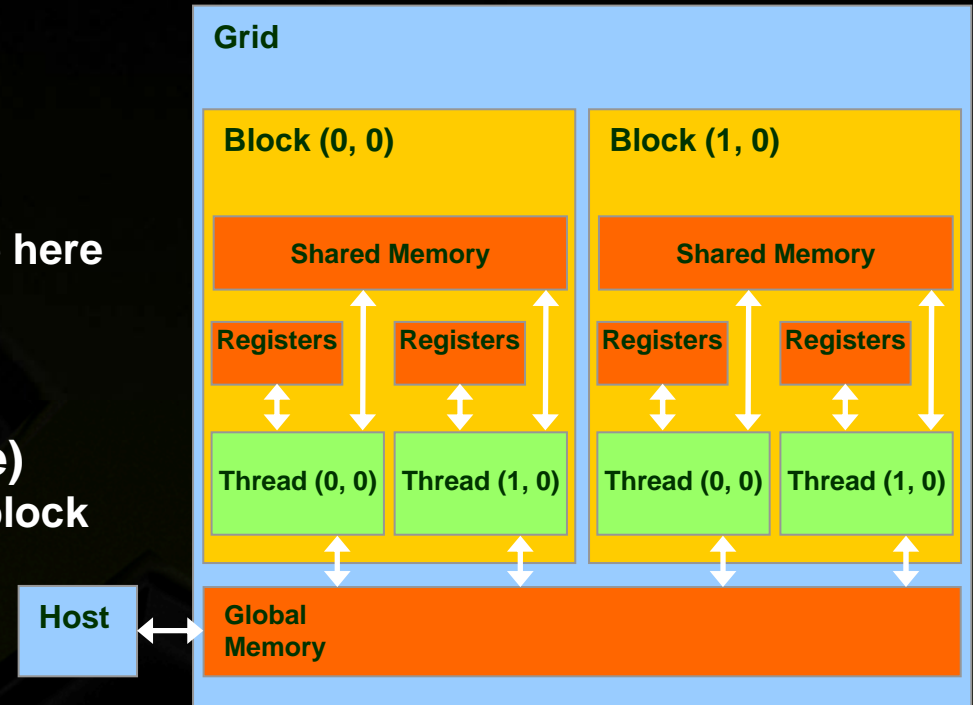
- Threads and blocks have IDs
 - Each thread can decide what data to work on
- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multi-dimensional data
 - Image processing
 - Solving PDEs on volumes



Kernel Memory Access



- **Registers**
- **Global Memory (external DRAM)**
 - Kernel input and output data reside here
 - Off-chip, large
 - Uncached
- **Shared Memory (Parallel Data Cache)**
 - Shared among threads in a single block
 - On-chip, small
 - As fast as registers



- **The host can read & write global memory but not shared memory**

Execution Model



- **Kernels are launched in grids**
 - One kernel executes at a time
- **A block executes on one Streaming Multiprocessor (SM)**
 - Does not migrate
- **Several blocks can reside concurrently on one SM**
 - **Control limitations (of G8X/G9X GPUs):**
 - At most **8** concurrent blocks per SM
 - At most **768** concurrent threads per SM
 - **Number is further limited by SM resources**
 - **Register file** is partitioned among all resident threads
 - **Shared memory** is partitioned among all resident thread blocks

CUDA Advantages over Legacy GPGPU

(Legacy GPGPU is programming GPU through graphics APIs)

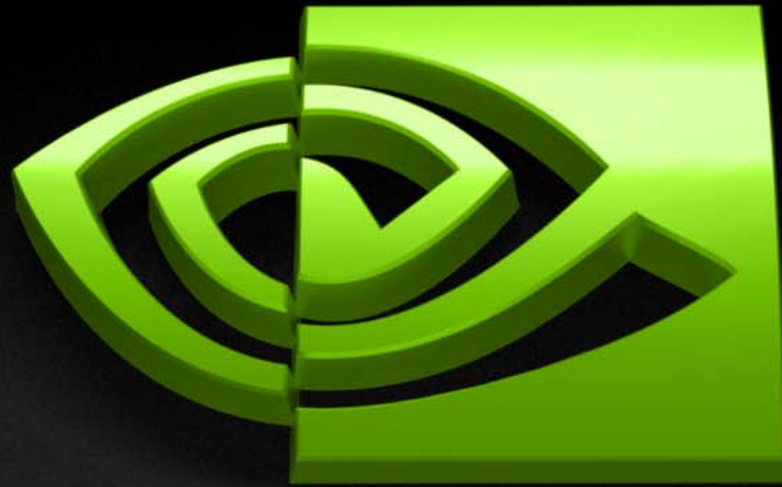
- **Random access byte-addressable memory**
 - Thread can access any memory location
- **Unlimited access to memory**
 - Thread can read/write as many locations as needed
- **Shared memory (per block) and thread synchronization**
 - Threads can cooperatively load data into shared memory
 - Any thread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

CUDA Model Summary



- **Thousands of lightweight concurrent threads**
 - No switching overhead
 - Hide instruction and memory latency
- **Shared memory**
 - User-managed L1 cache
 - Thread communication / cooperation within blocks
- **Random access to global memory**
 - Any thread can read/write any location(s)
- **Current generation hardware:**
 - Up to 128 streaming processors

Memory	Location	Cached	Access	Scope (“Who?”)
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host



NVIDIA®

Programming CUDA

The Basics

Outline of CUDA Basics



● Basics to set up and execute GPU code:

- GPU memory management
- GPU kernel launches
- Some specifics of GPU code

● Basics of some additional features:

- Vector types
- Managing multiple GPUs, multiple CPU threads
- Checking CUDA errors
- CUDA event API
- Compilation path

● NOTE: only the basic features are covered

- See the Programming Guide for many more API functions

Managing Memory



- **Host (CPU) code manages device (GPU) memory:**
 - Allocate / free
 - Copy data
 - Applies to *global* and *constant* device memory (DRAM)
- ***Shared* memory (on-chip) is statically allocated**
- **Host manages texture data:**
 - Stored on GPU
 - Takes advantage of texture caching / filtering / clamping
- **Host manages pinned (non-pageable) CPU memory:**
 - Allocate / free

GPU Memory Allocation / Release



- `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);**
 - **direction** specifies locations (host or device) of **src** and **dst**
 - Blocks CPU thread: returns after the copy is complete
 - Doesn't start copying until previous CUDA calls complete
- **cudaMemcpyAsync(..., cudaStream_t streamId)**
 - Host memory must be pinned (allocate with **cudaMallocHost**)
 - Returns immediately
 - doesn't start copying until previous CUDA calls in stream **streamId** or 0 complete
- **enum cudaMemcpyKind**
 - **cudaMemcpyHostToDevice**
 - **cudaMemcpyDeviceToHost**
 - **cudaMemcpyDeviceToDevice**

Exercise 1



- **We're going to dive right into programming CUDA**
- **In exercise 1 you will learn to use `cudaMalloc` and `cudaMemcpy`**

Executing Code on the GPU



- **C function with some restrictions**

- Can only access GPU memory
- No variable number of arguments (“varargs”)
- No static variables

- **Must be declared with a qualifier**

- **__global__** : invoked from within host (CPU) code, cannot be called from device (GPU) code must return void
- **__device__** : called from other GPU functions, cannot be called from host (CPU) code
- **__host__** : can only be executed by CPU, called from host
- **__host__** and **__device__** qualifiers can be combined
 - sample use: overloading operators
 - Compiler will generate both CPU and GPU code

Launching kernels on GPU



● Modified C function call syntax:

```
kernel<<<dim3 grid, dim3 block, int smem, int stream>>>(...)
```

● Execution Configuration (“<<< >>>”):

- grid dimensions: **x** and **y**
- thread-block dimensions: **x**, **y**, and **z**
- shared memory: number of bytes per block for extern smem variables declared without size
 - optional, 0 by default
- stream ID
 - optional, 0 by default

```
dim3 grid(16, 16);
```

```
dim3 block(16,16);
```

```
kernel<<<grid, block, 0, 0>>>(...);
```

```
kernel<<<32, 512>>>(...);
```

CUDA Built-in Device Variables



- All **__global__** and **__device__** functions have access to these automatically defined variables

- **dim3 gridDim;**

- Dimensions of the grid in blocks (gridDim.z unused)

- **dim3 blockDim;**

- Dimensions of the block in threads

- **dim3 blockIdx;**

- Block index within the grid

- **dim3 threadIdx;**

- Thread index within the block

Minimal Kernels



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

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

Common Pattern!

Minimal Kernel for 2D data



```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * w + ix;

    d_a[idx] = value;
}

...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```

Exercise 2: your first CUDA kernel



- In this exercise you will write and execute a simple CUDA kernel

Host Synchronization



- **All kernel launches are asynchronous**
 - control returns to CPU immediately
 - kernel executes after all previous CUDA calls have completed
- **cudaMemcpy is synchronous**
 - control returns to CPU after copy completes
 - copy starts after all previous CUDA calls have completed
- **cudaThreadSynchronize()**
 - blocks until all previous CUDA calls complete
- **Async API provides:**
 - GPU CUDA-call streams
 - non-blocking **cudaMemcpyAsync**

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}
```

```
void main()
{
    .....
    increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}
```

```
void main()
{
    .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```


Example: Increment Array Elements

Increment N-element vector a by scalar b



Let's assume $N=16$, **blockDim**=4 \rightarrow 4 blocks



blockIdx.x=0
blockDim.x=4
threadIdx.x=0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=0,1,2,3
idx=4,5,6,7

blockIdx.x=2
blockDim.x=4
threadIdx.x=0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15

int idx = blockDim.x * blockIdx.x + threadIdx.x;
will map from local index **threadIdx** to global index

NB: **blockDim** should be ≥ 32 in real code, this is just an example

Example: Host Code

```
// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numBytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<< N/blockSize, blockSize>>>(d_A, b);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```

Variable Qualifiers (GPU code)



● **__device__**

- stored in device memory (large, high latency, no cache)
- Allocated with **cudaMalloc** (**__device__** qualifier implied)
- accessible by all threads
- lifetime: application

● **__constant__**

- same as **__device__**, but cached and read-only by GPU
- written by CPU via **cudaMemcpyToSymbol(...)** call
- lifetime: application

● **__shared__**

- stored in on-chip shared memory (very low latency)
- accessible by all threads in the same thread block
- lifetime: kernel launch

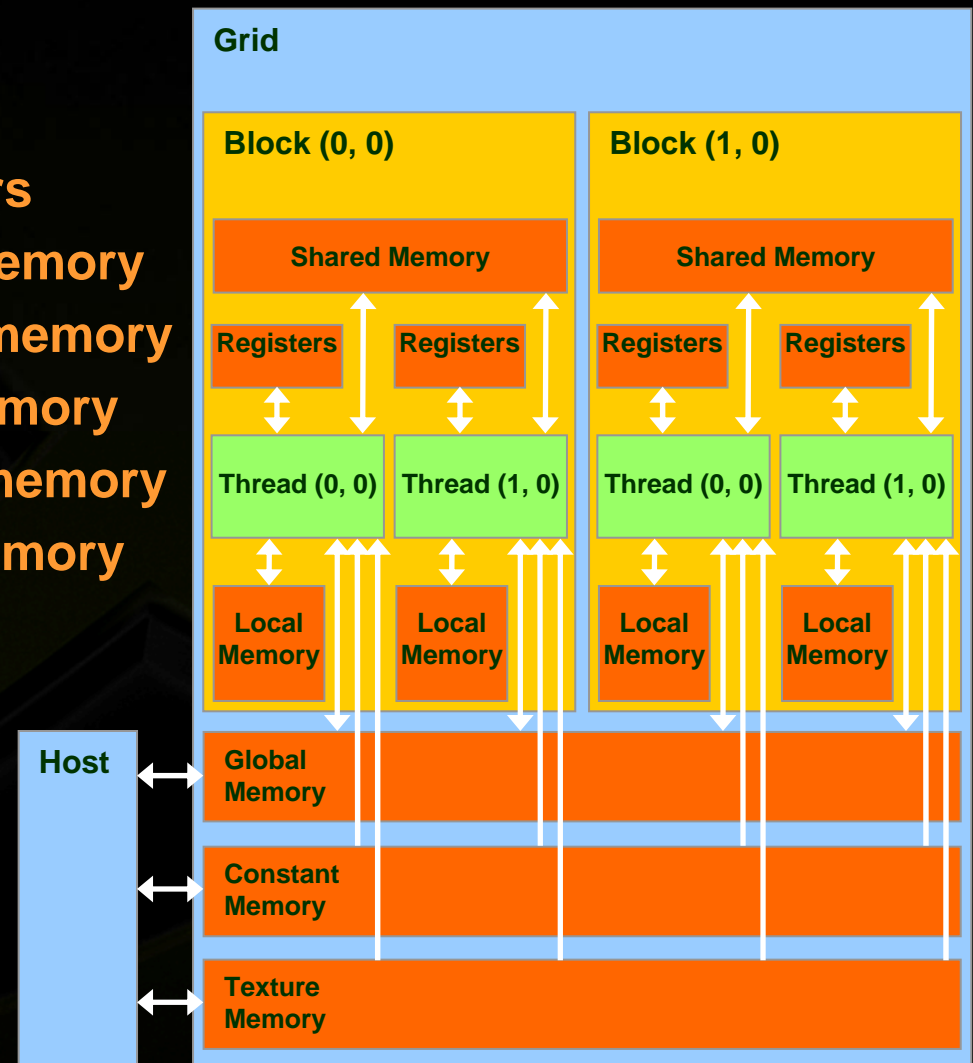
● **Unqualified variables:**

- scalars and built-in vector types are stored in registers
- arrays of more than 4 elements stored in device memory

CUDA Memory Spaces



- Each thread can:
 - Read/write per-thread **registers**
 - Read/write per-thread **local memory**
 - Read/write per-block **shared memory**
 - Read/write per-grid **global memory**
 - Read only per-grid **constant memory**
 - Read only per-grid **texture memory**
- The host can read/write **global, constant, and texture memory (stored in DRAM)**



CUDA Memory Spaces



- **Global and Shared Memory** introduced before
 - Most important, commonly used
- **Local, Constant, and Texture** for convenience/performance
 - **Local**: automatic array variables allocated there by compiler
 - **Constant**: useful for uniformly-accessed read-only data
 - **Cached** (see programming guide)
 - **Texture**: useful for spatially coherent random-access read-only data
 - **Cached** (see programming guide)
 - **Provides address clamping and wrapping**

Memory	Location	Cached	Access	Scope ("Who?")
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Built-in Vector Types



- Can be used in GPU and CPU code

- `[u]char[1..4]`, `[u]short[1..4]`, `[u]int[1..4]`,
`[u]long[1..4]`, `float[1..4]`

- Structures accessed with `x`, `y`, `z`, `w` fields:

```
uint4 param;  
int y = param.y;
```

- `dim3`

- Based on `uint3`
- Used to specify dimensions
- Default value (1,1,1)

Thread Synchronization Function



- **void __syncthreads();**
- **Synchronizes all threads in a block**
 - Generates barrier synchronization instruction
 - No thread can pass this barrier until all threads in the block reach it
 - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- **Allowed in conditional code only if the conditional is uniform across the entire thread block**

GPU Atomic Integer Operations



- **Atomic operations on integers in global memory:**
 - Associative operations on signed/unsigned ints
 - add, sub, min, max, ...
 - and, or, xor
 - Increment, decrement
 - Exchange, compare and swap
- **Requires hardware with compute capability 1.1**

Device Management



- **CPU can query and select GPU devices**

- `cudaGetDeviceCount(int *count)`
- `cudaSetDevice(int device)`
- `cudaGetDevice(int *current_device)`
- `cudaGetDeviceProperties(cudaDeviceProp* prop, int device)`
- `cudaChooseDevice(int *device, cudaDeviceProp* prop)`

- **Multi-GPU setup:**

- device 0 is used by default
- one CPU thread can control only one GPU
 - multiple CPU threads can control the same GPU
 - calls are serialized by the driver

Multiple CPU Threads and CUDA



- **CUDA resources allocated by a CPU thread can be consumed only by CUDA calls from the same CPU thread**
- **Violation Example:**
 - CPU **thread 2** allocates GPU memory, stores address in ***p***
 - **thread 3** issues a CUDA call that accesses memory via ***p***

CUDA Error Reporting to CPU



- **All CUDA calls return error code:**
 - except for kernel launches
 - `cudaError_t` type
- **`cudaError_t cudaGetLastError(void)`**
 - returns the code for the last error (no error has a code)
- **`char* cudaGetErrorString(cudaError_t code)`**
 - returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
```

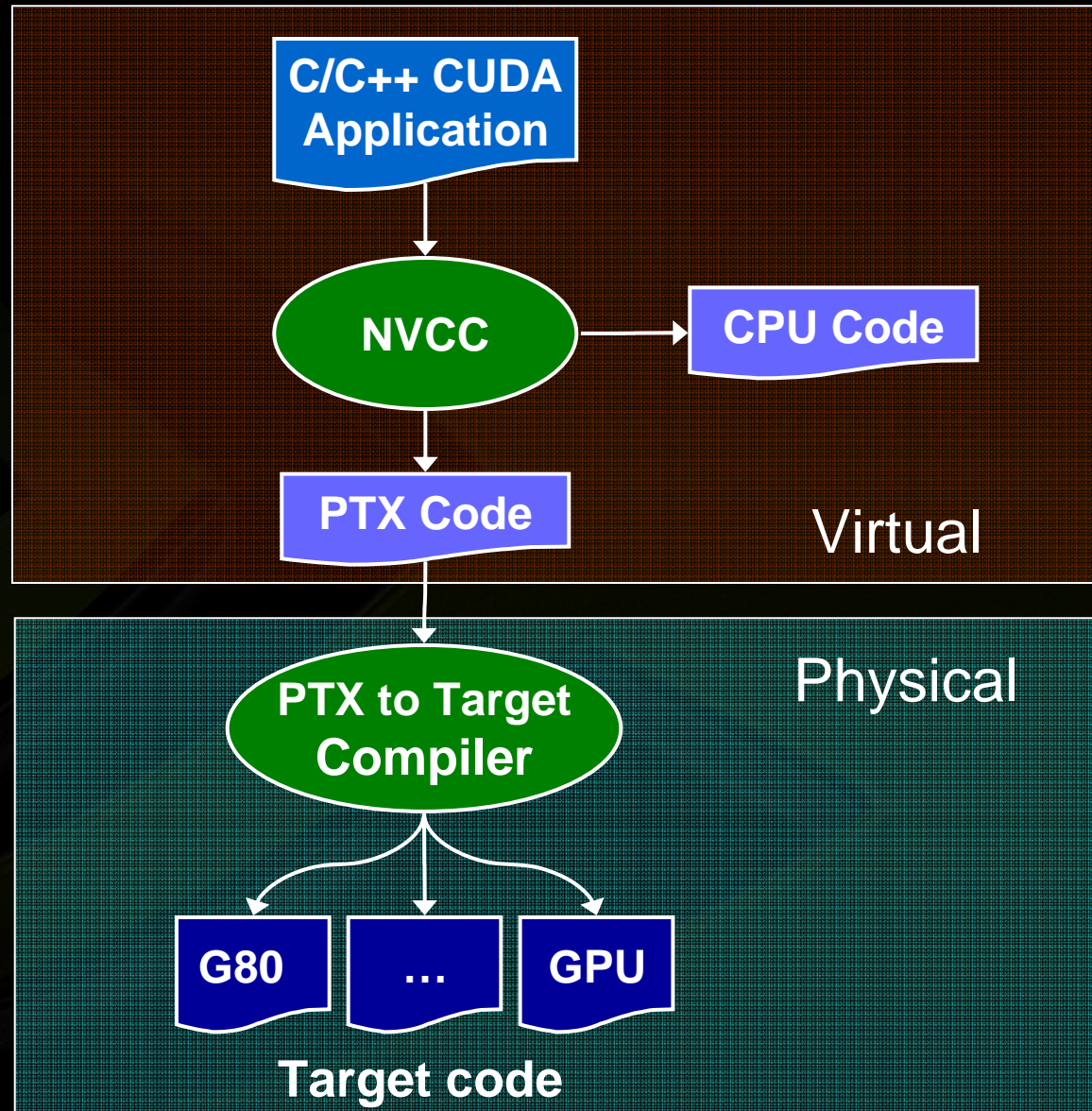
CUDA Event API



- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
 - measure elapsed time for CUDA calls (clock cycle precision)
 - query the status of an asynchronous CUDA call
 - block CPU until CUDA calls prior to the event are completed
 - **asyncAPI** sample in CUDA SDK

```
cudaEvent_t start, stop;  
cudaEventCreate(&start);          cudaEventCreate(&stop);  
cudaEventRecord(start, 0);  
kernel<<<grid, block>>>(...);  
cudaEventRecord(stop, 0);  
cudaEventSynchronize(stop);  
float et;  
cudaEventElapsedTime(&et, start, stop);  
cudaEventDestroy(start);          cudaEventDestroy(stop);
```

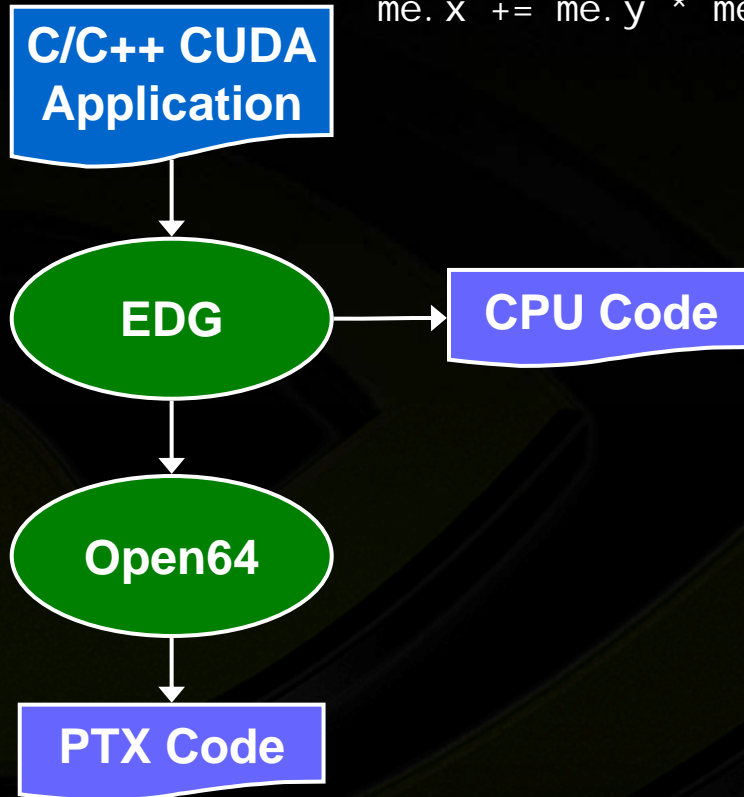

Compiling CUDA



NVCC & PTX Virtual Machine



```
float4 me = gx[gtid];  
me.x += me.y * me.z;
```



- **EDG**
 - Separate GPU vs. CPU code
- **Open64**
 - Generates GPU PTX assembly
- **Parallel Thread eXecution (PTX)**
 - Virtual Machine and ISA
 - Programming model
 - Execution resources and state

```
ld.global.v4.f32 {$f1, $f3, $f5, $f7}, [$r9+0];  
mad.f32 $f1, $f5, $f3, $f1;
```

Compilation



- Any source file containing CUDA language extensions must be compiled with **nvcc**
- NVCC is a **compiler driver**
 - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
 - Either C code (CPU Code)
 - That must then be compiled with the rest of the application using another tool
 - Or PTX object code directly
- **An executable with CUDA code requires:**
 - The CUDA core library (**cuda**)
 - The CUDA runtime library (**cudart**)
 - if runtime API is used
 - loads **cuda** library

Exercise 3: Reverse a Small Array



- **Given an input array, reverse it**
- **In this part, you will reverse a small array**
 - **the Size of a single thread block**

Exercise 4: Reverse a Large Array



- **Given a large input array, reverse it**
- **This requires launching many thread blocks**



NVIDIA®

Getting Started

Get CUDA



- **CUDA Zone: <http://nvidia.com/cuda>**
 - **Programming Guide and other Documentation**
 - **Toolkits and SDKs for:**
 - **Windows**
 - **Linux**
 - **MacOS**
 - **Libraries**
 - **Plugins**
 - **Forums**
 - **Code Samples**

Come visit the class!

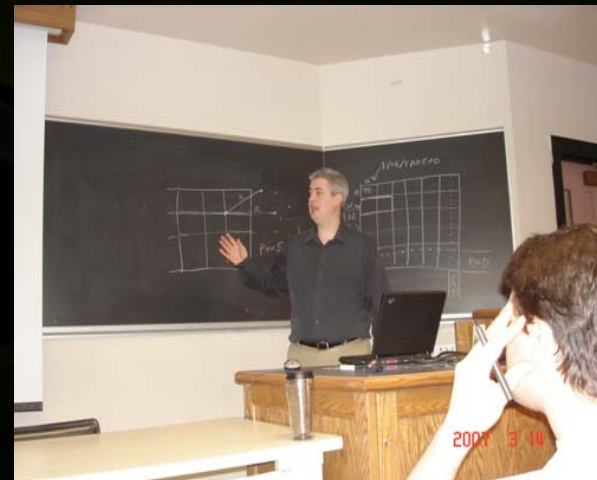


- **UIUC ECE498AL – Programming Massively Parallel Processors**
(<http://courses.ece.uiuc.edu/ece498/al/>)

- **David Kirk (NVIDIA) and Wenmei Hwu (UIUC) co-instructors**

- **CUDA programming, GPU computing, lab exercises, and projects**

- **Lecture slides and voice recordings**



Questions?