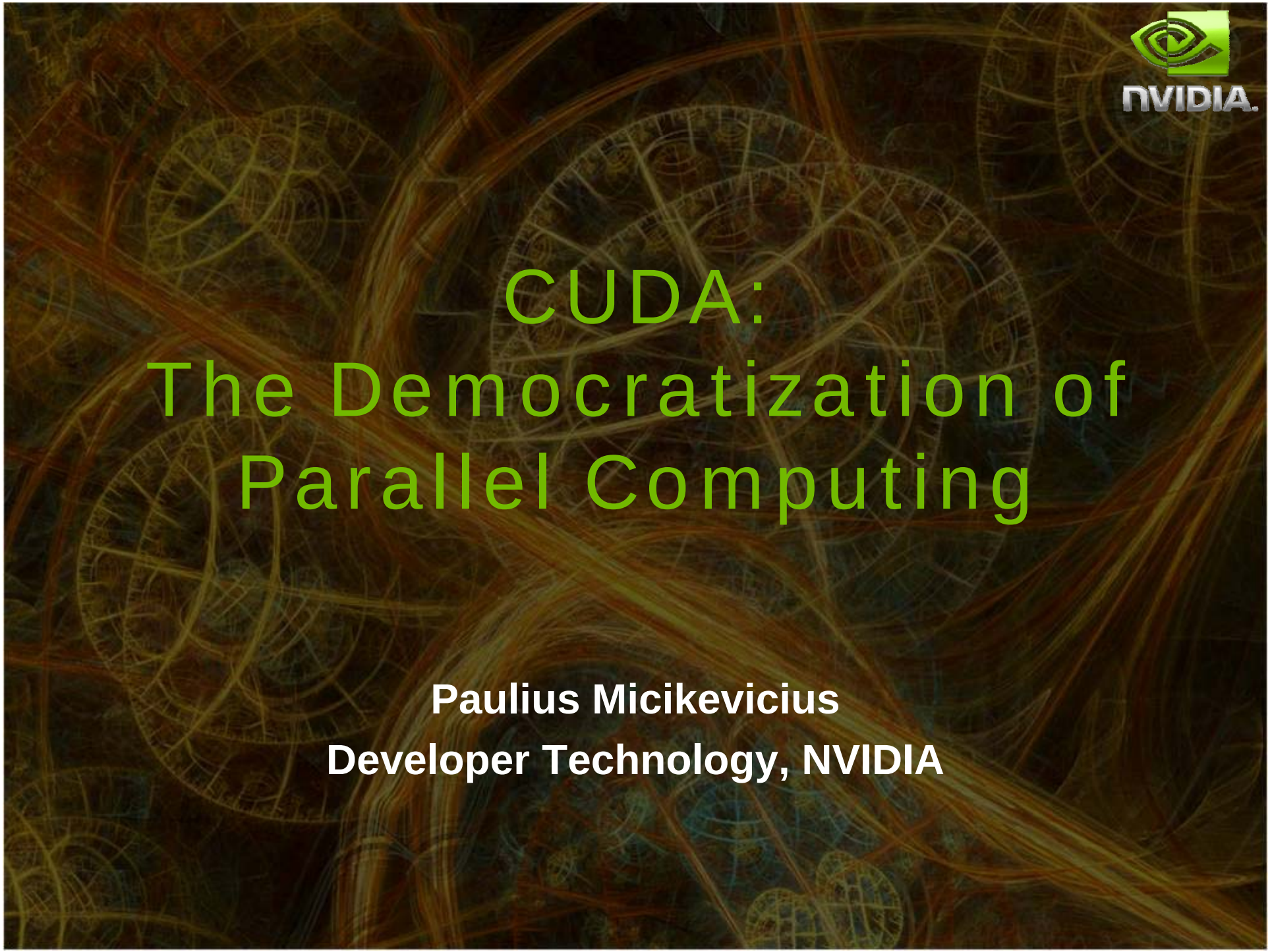


CUDA: The Democratization of Parallel Computing

Paulius Micikevicius
Developer Technology, NVIDIA

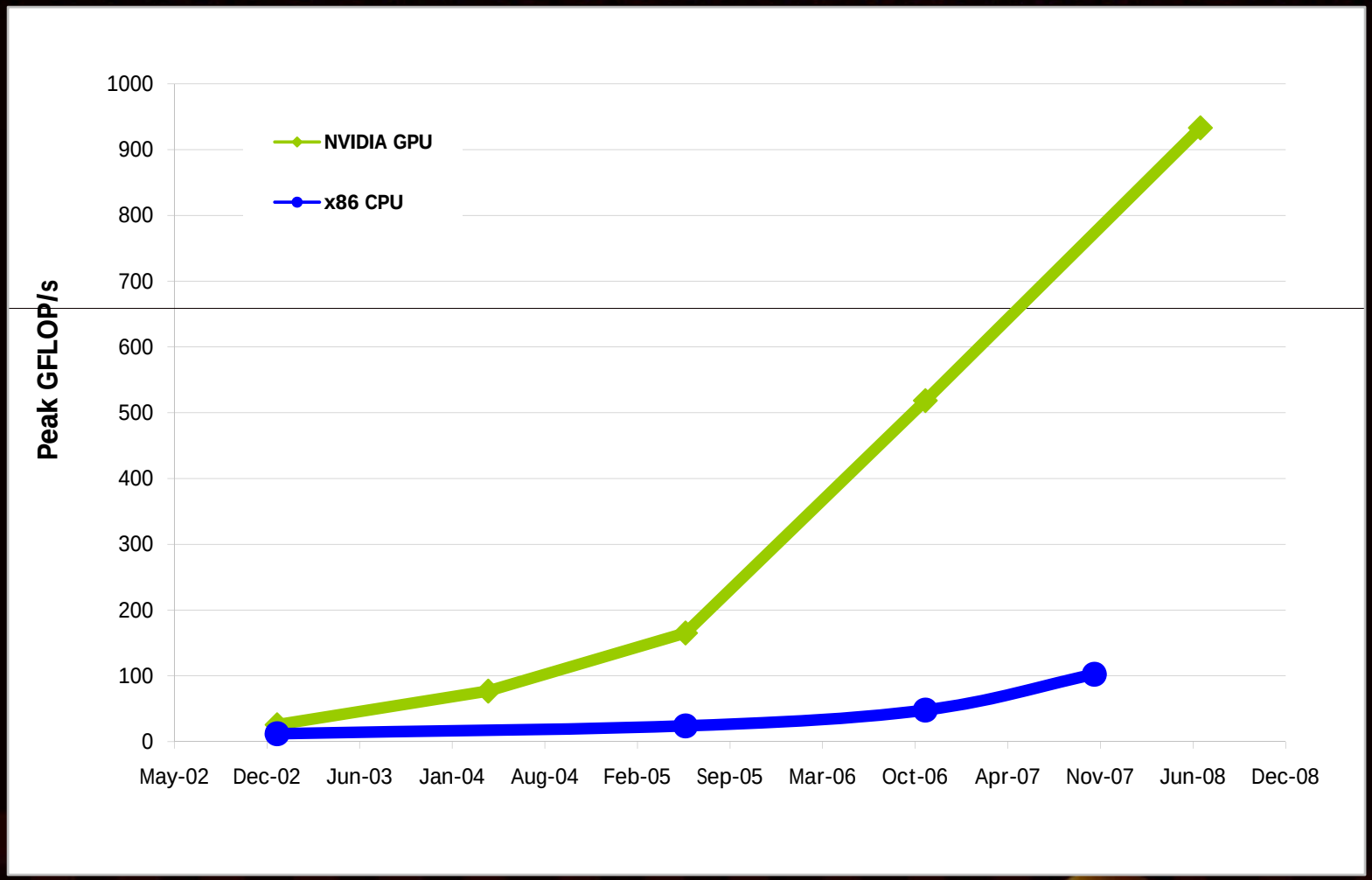
Outline

- Why GPUs?
- CUDA
  - Programming and memory models
- NVIDIA GPU Architecture
- CUDA resources
  - Tools, Libraries
- Sample compute apps

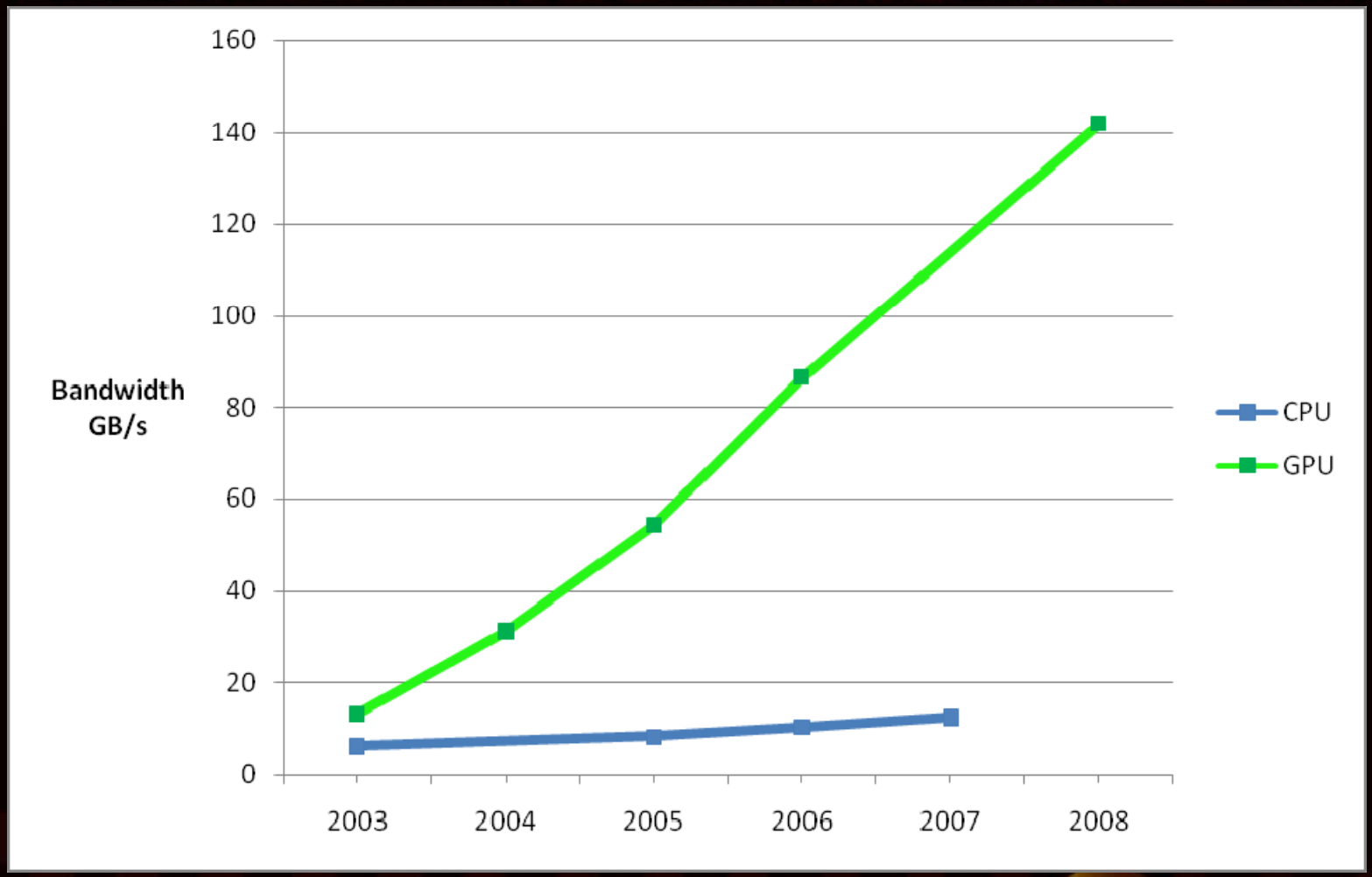
Why GPUs?

- GPUs have evolved into highly parallel machines
  - already provide 10s to 100s of cores
  - fully programmable in C
    - no graphics knowledge needed
  - lots of compute power and bandwidth
- GPUs are widely available
  - over 80M CUDA-capable GPUs shipped
  - laptops to 1U servers

Single Precision GFlops/s



Memory Bandwidth



What is CUDA?

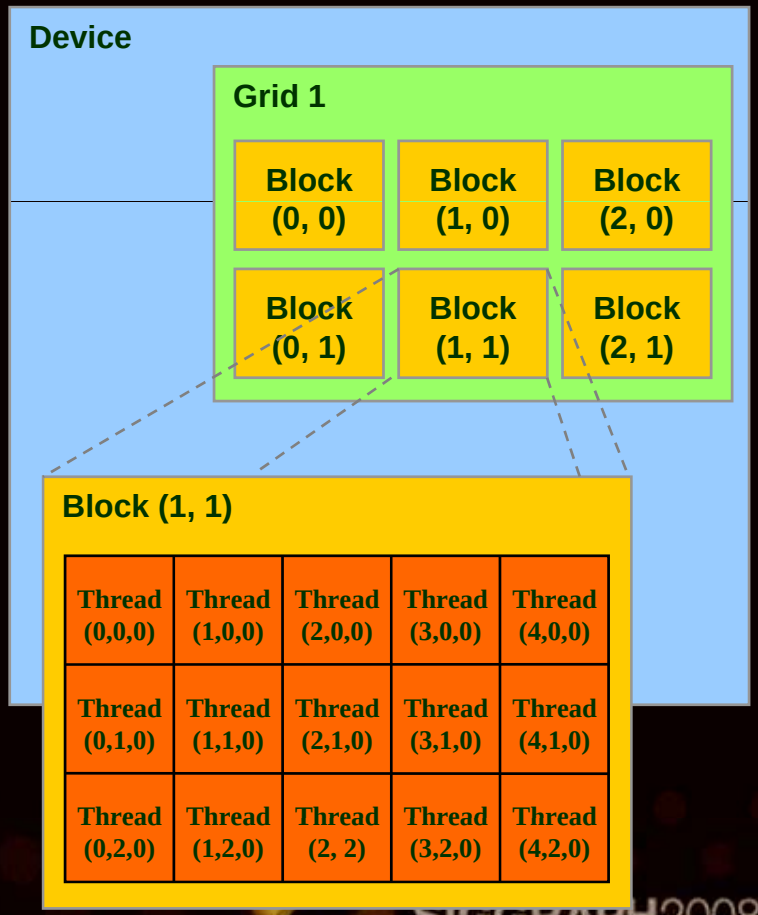
- C with minimal extensions
- CUDA goals:
  - scale code to 100s of cores
  - scale code to 1000s of parallel threads
  - enable heterogeneous computing:
    - CPU + GPU
- CUDA defines:
  - programming model
  - memory model

CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
  - synchronize their execution
  - communicate via shared memory
- Parallel code is written for a thread
  - each thread is free to execute a unique code path
  - built-in thread and block ID variables
- CUDA threads vs CPU threads
  - CUDA thread switching is free
  - CUDA uses many threads per core

IDs and Dimensions

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



Example: Increment Array Elements

CPU code

```
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, 16);
}
```

Example: Increment Array Elements

CPU code

```
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, 16);
}
```

CUDA code

```
__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()
{
  .....
    increment_gpu<<< 4, 4 >>>(a, b, 16);
}
```

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let’s assume N=16, blockDim=4 -> 4 blocks

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```

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

Minimal Kernel for 2D data

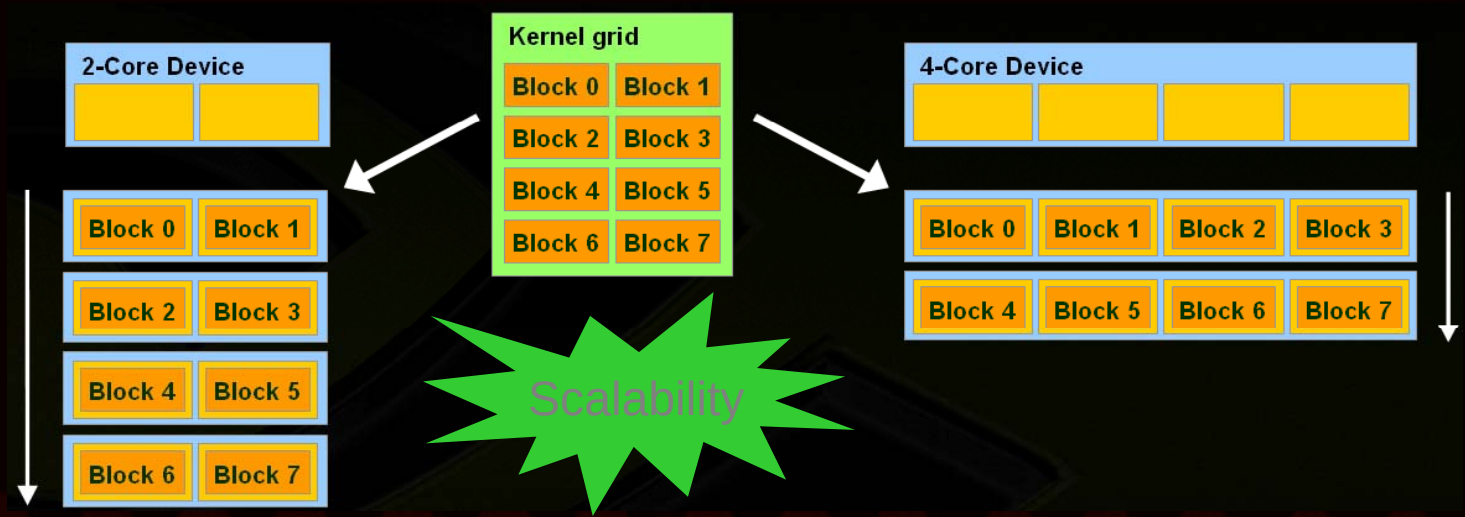
```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * w + ix;

    d_a[idx] = value;
}
```

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

Hierarchy of concurrent threads

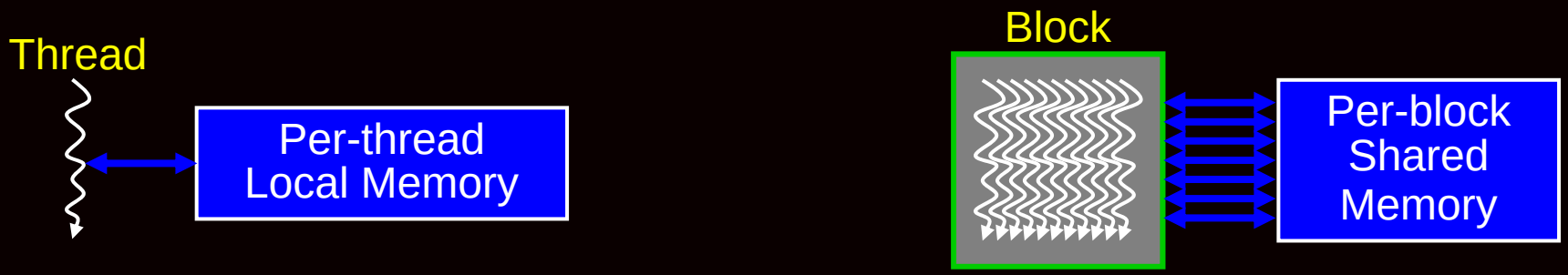
- **Thread blocks can run in any order**
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices



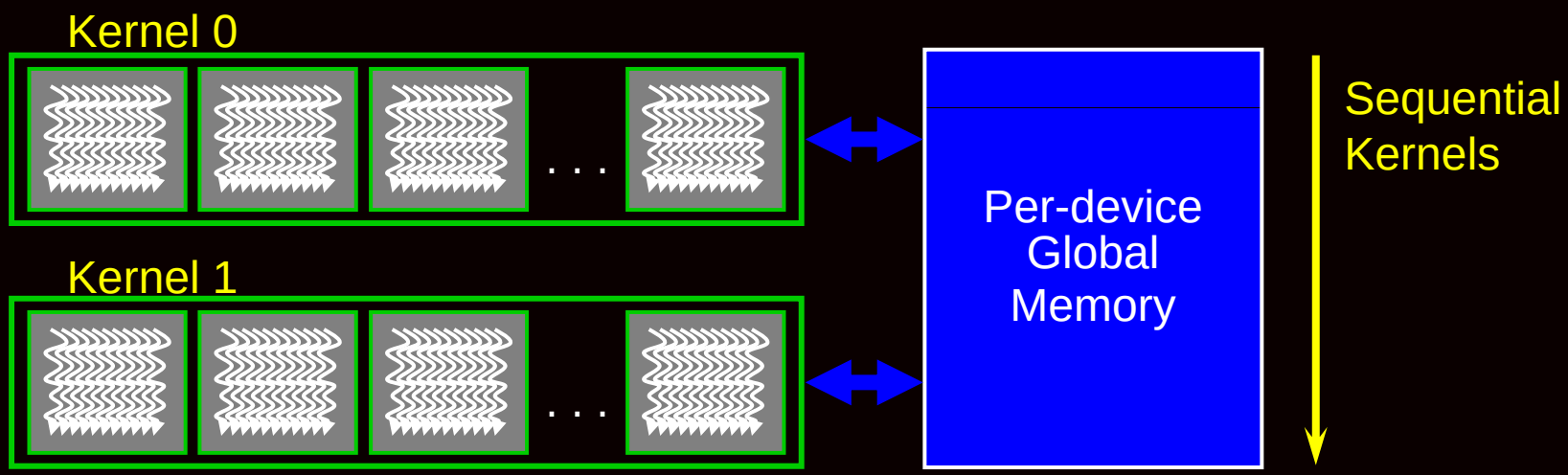
Memory Model

- **Local storage**
  - each thread has own local storage
  - data lifetime = thread lifetime
- **Shared memory**
  - each thread block has own shared memory
    - accessible only by threads within that block
  - data lifetime = block lifetime
- **Global (device) memory**
  - accessible by all threads as well as host (CPU)
  - data lifetime = from allocation to deallocation
- **Host (CPU) memory**
  - not directly accessible by CUDA threads

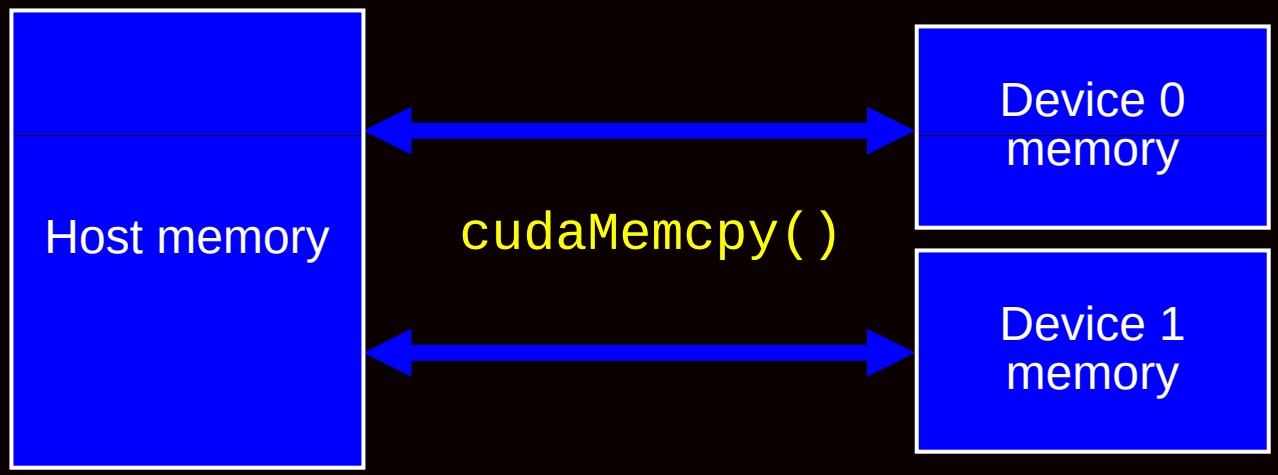
Memory model



Memory model



Memory model



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, N);


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

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

Additional CUDA Features

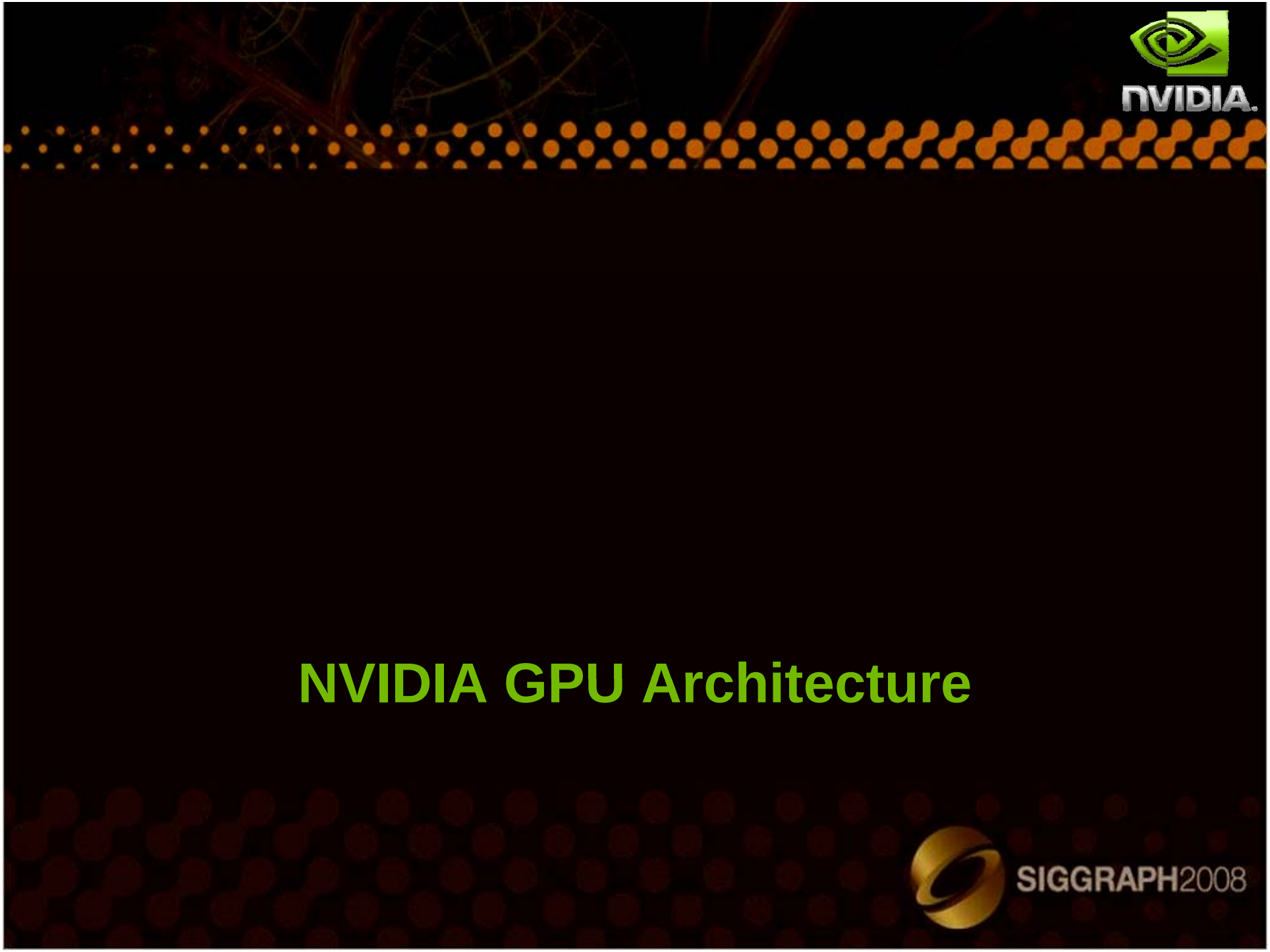
- CUDA exposes texture hardware
  - cached
  - 1D, 2D, 3D textures
  - filtering essentially free
- Constant memory
  - Read-only
  - Cached
- Graphics interop

Graphics Interoperability

- Data sharing between CUDA and shader code
  - buffer objects
  - surfaces
- OpenGL and DirectX
- PhysX:
  - ported to CUDA for GPU execution

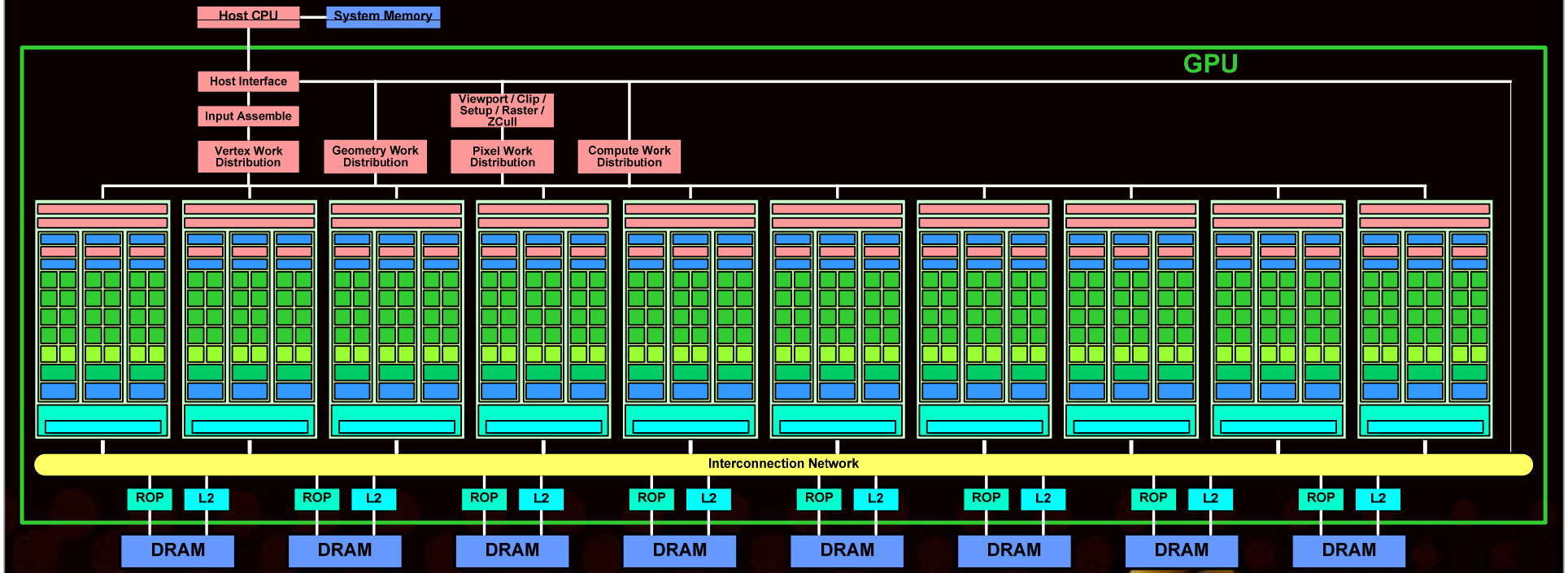
CUDA Advantages over Legacy GPGPU

- **Random access byte-addressable memory**
  - Threads can access any memory location
- **Unlimited access to memory**
  - No need to ping-pong buffers
- **Shared memory and thread synchronization**
  - Threads can communicate via shared memory

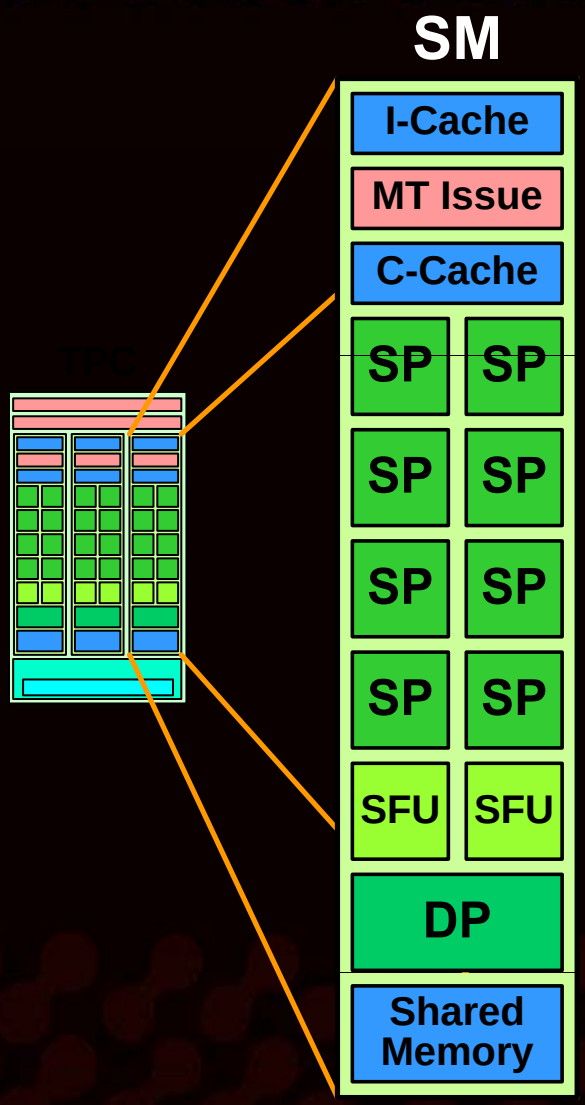

NVIDIA
NVIDIA GPU Architecture
SIGGRAPH2008

GT200 Compute Architecture

- 240 thread processors
- Adds double precision IEEE 754 floating point
- Shared memory atomics
- Asynchronous memory CPU-GPU memory transfers



Tesla Multiprocessor



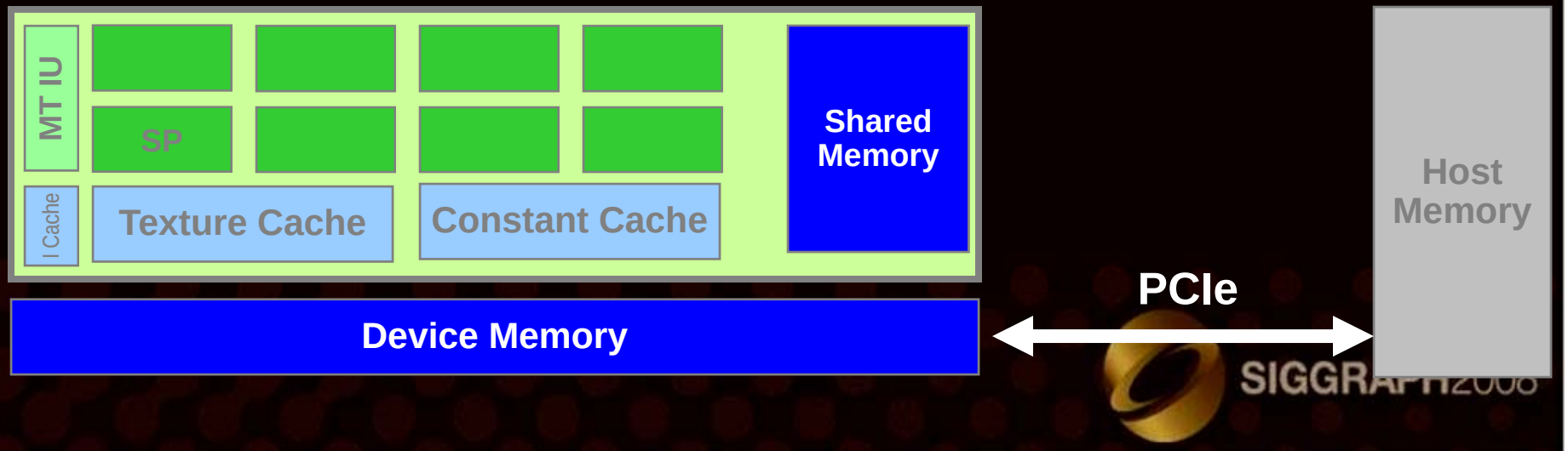
- Scalar register-based ISA
- Multithreaded Instruction Unit
  - Up to 1024 concurrent threads
  - Hardware thread scheduling
  - In-order issue
- 8 SP Thread Processors
  - IEEE 754 32-bit floating point
  - 32-bit and 64-bit integer
  - 16K 32-bit registers
- 2 SFU Special Function Units
- Double Precision Unit
  - IEEE 754 64-bit floating point
  - Fused multiply-add
- 16KB Shared Memory

Hardware and Programming Model

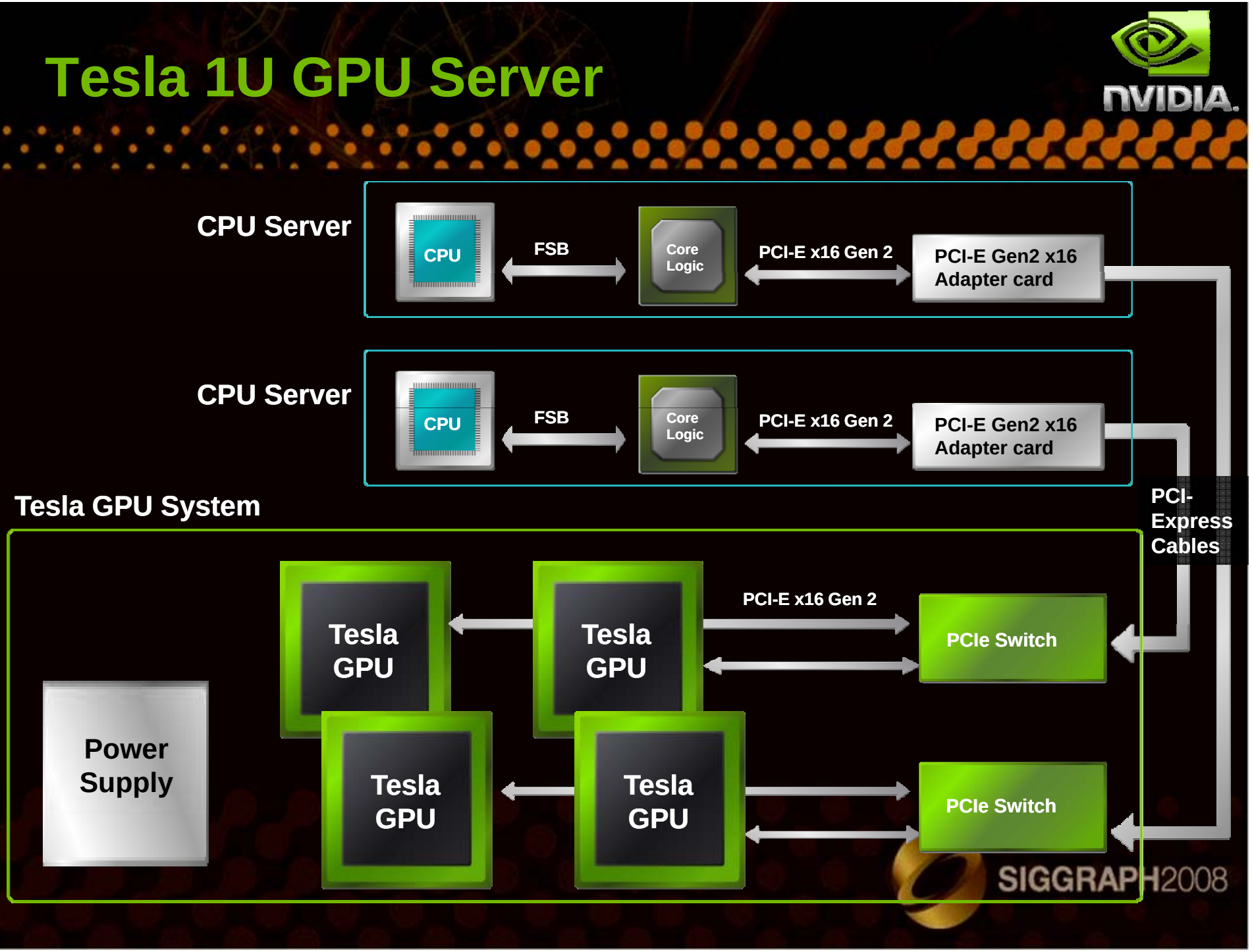
- **Thread blocks are partitioned among multiprocessors**
  - a block runs to completion
  - a block doesn't run until resources are available
- **Allocation of hardware resources**
  - shared memory is partitioned among blocks
  - registers are partitioned among threads
- **Hardware thread scheduling**
  - any thread not waiting for something can run
  - context switching is free – every cycle

Memory Architecture

- Direct load/store access to device memory
  - treated as the usual linear sequence of bytes (i.e., not pixels)
  - 140 GB/s bandwidth, not cached
- Texture & constant caches are read-only access paths
- On-chip shared memory shared among threads of a block
  - important for communication amongst threads
  - provides low-latency temporary storage



Tesla 1U GPU Server




NVIDIA.
CUDA Development Resources
SIGGRAPH2008

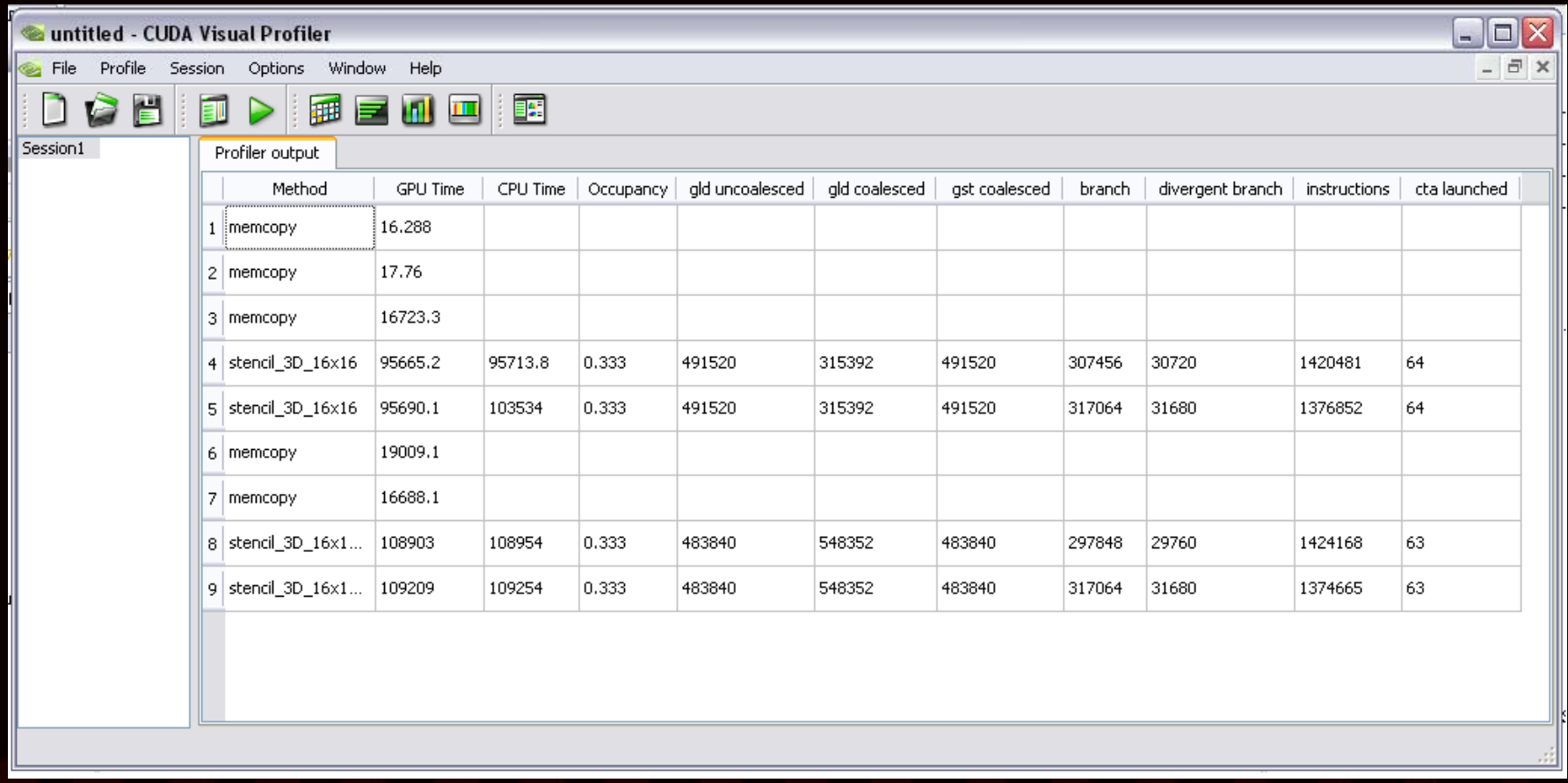
CUDA Programming Resources

- CUDA toolkit
  - compiler and libraries
  - free download for Windows, Linux, and MacOS
- CUDA SDK
  - code samples
  - whitepapers
- Instructional materials on CUDA Zone
  - slides and audio
  - parallel programming course at University of Illinois UC
  - tutorials
- Development tools
- Libraries

GPU Tools

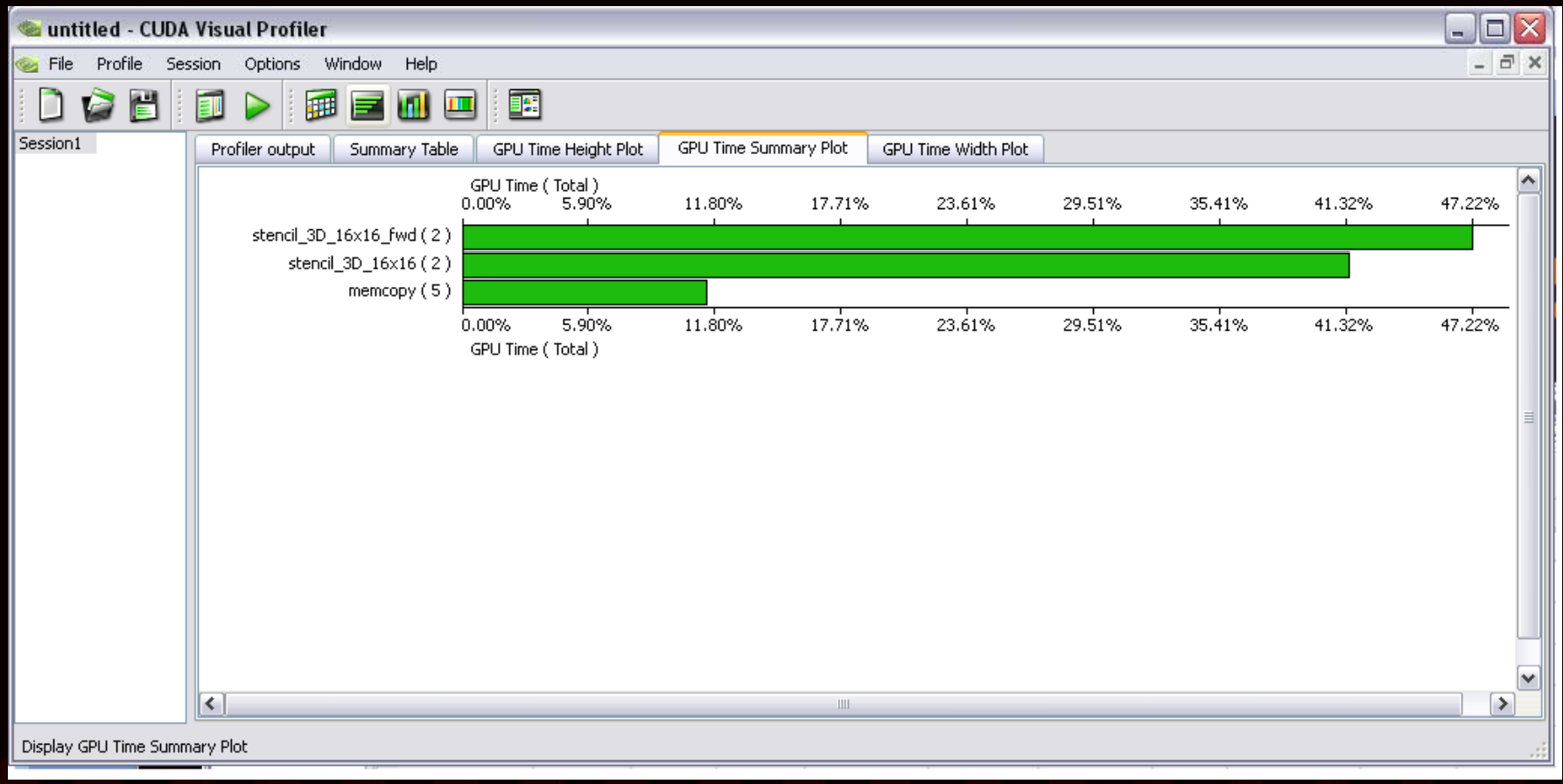
- **Profiler**
  - available now for all supported OSs
  - command-line or GUI
  - sampling signals on GPU for:
    - Memory access patterns
    - Execution (serialization, divergence)
- **Debugger**
  - runs on the GPU
  - what you would expect:
    - Breakpoints, examine and set variables, etc.
- **Emulation mode**
  - compile and execute in emulation on CPU
  - allows CPU-style debugging in GPU source

CUDA Profiler



	Method	GPU Time	CPU Time	Occupancy	gld uncoalesced	gld coalesced	gst coalesced	branch	divergent branch	instructions	cta launched
1	memcopy	16.288									
2	memcopy	17.76									
3	memcopy	16723.3									
4	stencil_3D_16x16	95665.2	95713.8	0.333	491520	315392	491520	307456	30720	1420481	64
5	stencil_3D_16x16	95690.1	103534	0.333	491520	315392	491520	317064	31680	1376852	64
6	memcopy	19009.1									
7	memcopy	16688.1									
8	stencil_3D_16x1...	108903	108954	0.333	483840	548352	483840	297848	29760	1424168	63
9	stencil_3D_16x1...	109209	109254	0.333	483840	548352	483840	317064	31680	1374665	63

CUDA Profiler



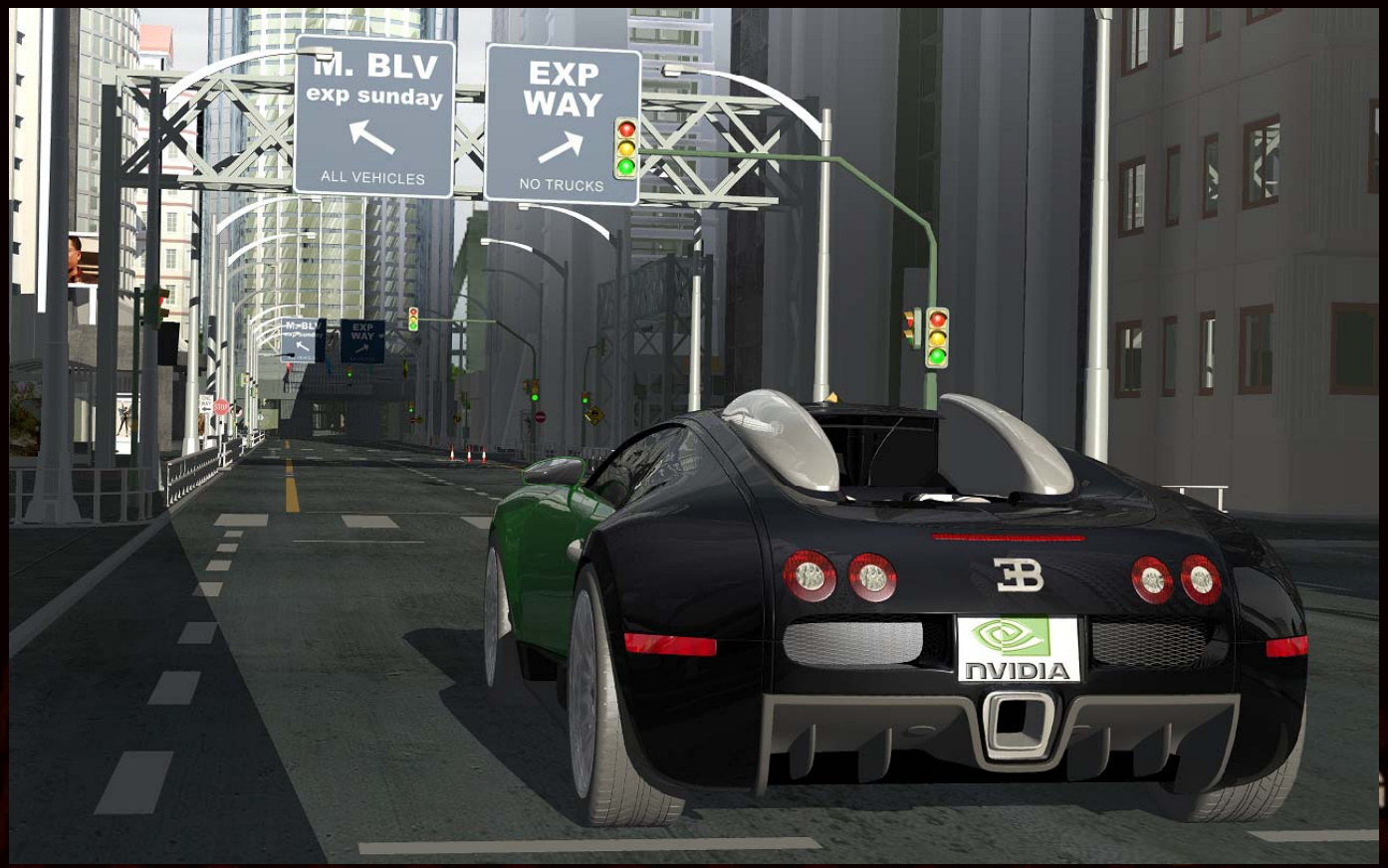
CUDA Libraries

- **BLAS**
  - real: level-1, level-2, level-3
  - complex: level-1
- **CUFFT**
  - mimics fftw API
  - 1D, 2D, 3D transforms, batched 1D
  - R2C, C2R, C2C
- **Features**
  - require NO programming in CUDA
  - callable from C/C++ or Fortran
  - open source
  - continually optimized


NVIDIA.
Some Sample Compute Applications
SIGGRAPH2008

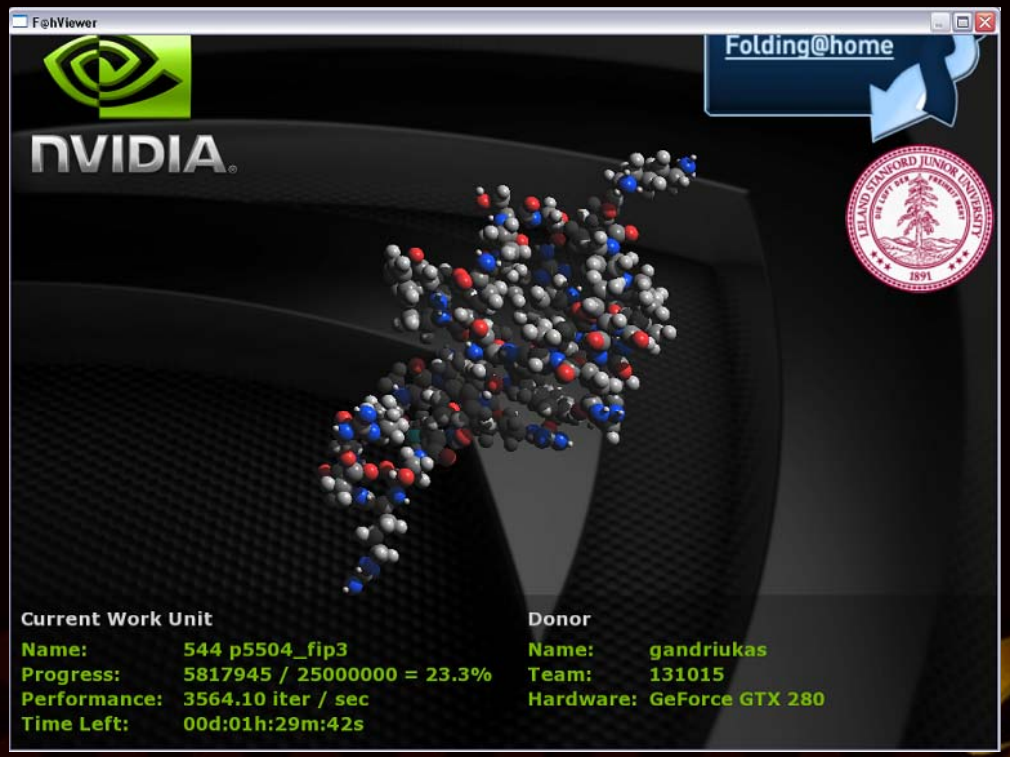
Interactive Ray Tracer

- Demo at NVIDIA exhibit booth
- More details in the next session (5:00pm today)

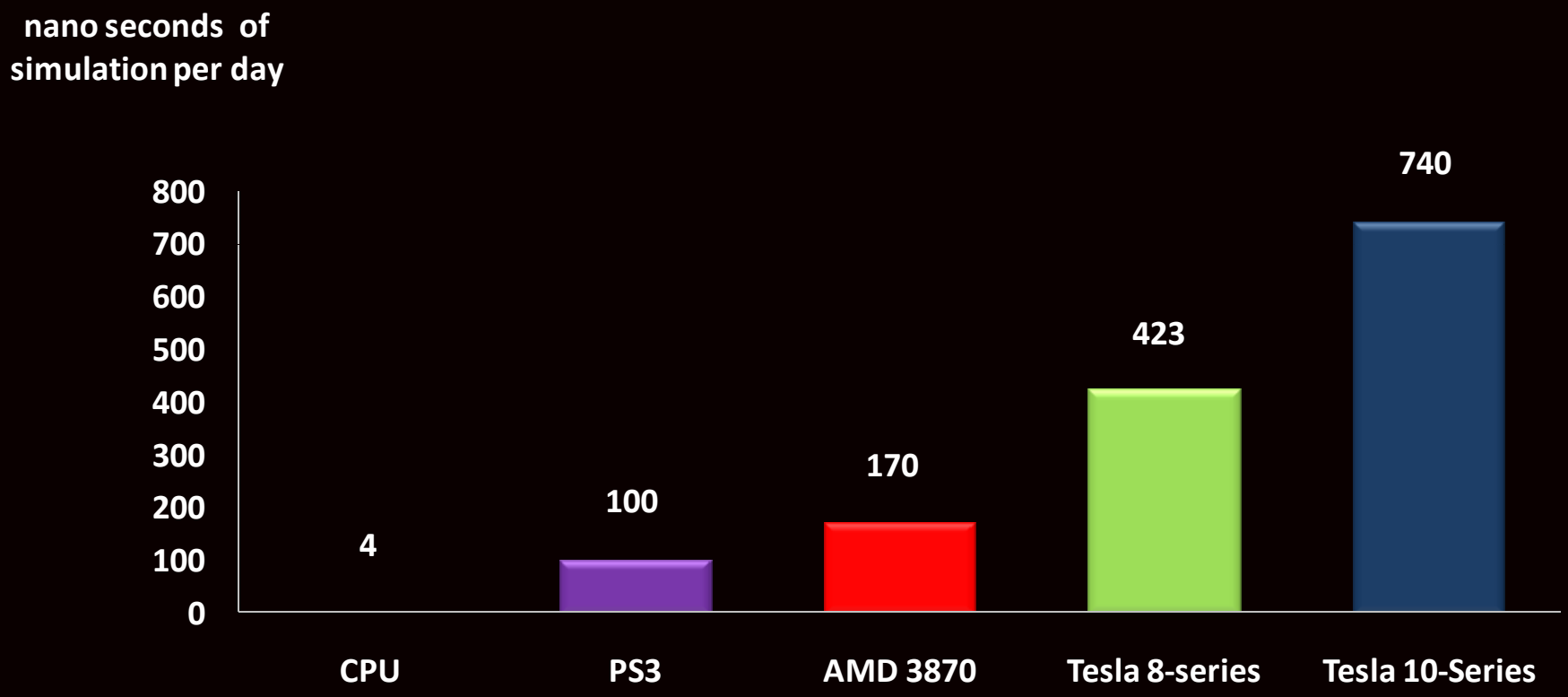


Folding at Home

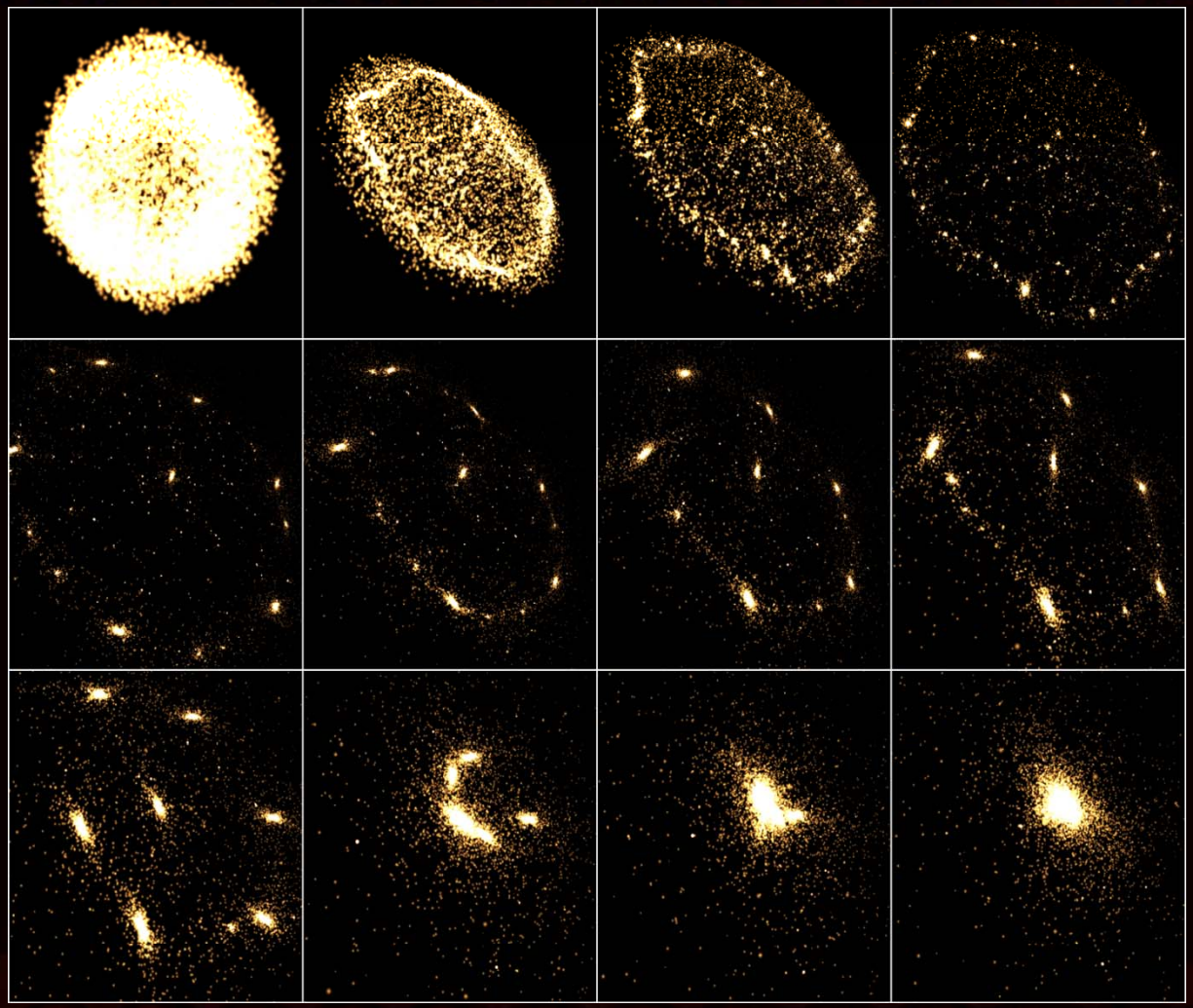
- **NVIDIA team total production:**
  - top 10% in 2 weeks with 10 GPUs
  - top 0.1% in less than a month with additional GPUs



Folding at Home Performance



CUDA N-Body Simulation

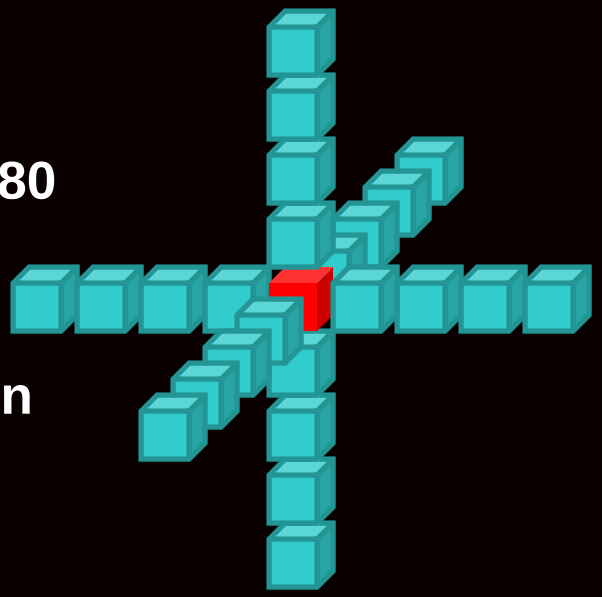


23B interactions/s
30K bodies

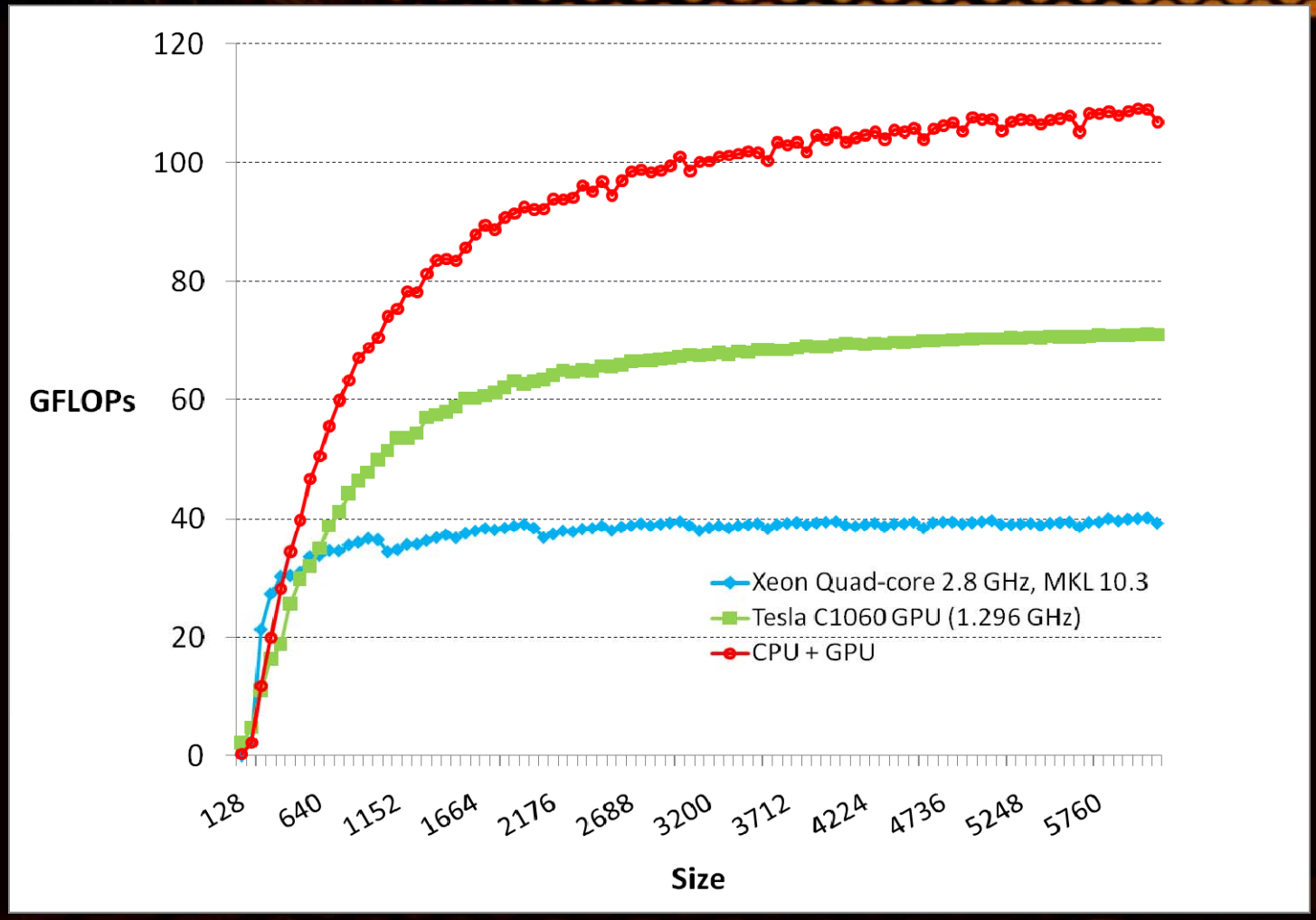
GeForce GTX 280:
470 GFlops/s

Finite Difference

- 3D finite difference
  - 25-point stencil (8th order in space)
  - 29 flops per output element
- Performance (512x512x400 data)
  - ~4,100 Melements/s on a single GTX 280
  - ~8,000 Melements/s on 2 GTX 280
    - Includes inter-GPU communication
  - ~200 Melements/s on a quad-core Xeon
    - E5462: 2.8GHz, 1600 MHz FSB

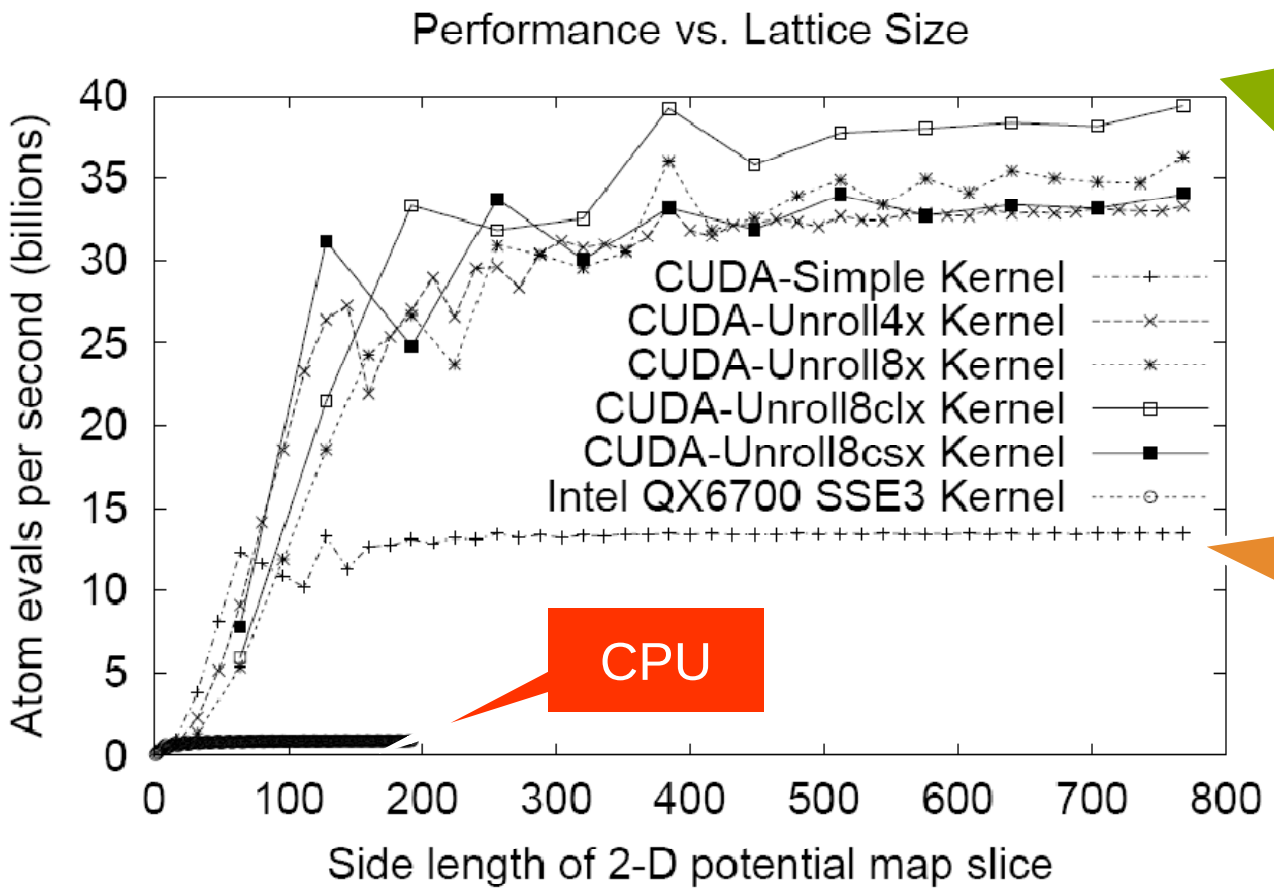


DGEMM Performance



GPU performance includes data copies over PCIe gen-2

Direct Coulomb Summation (VMD)



CUDA-Unroll8clx: fastest GPU kernel, 44x faster than CPU, 291 GFLOPS on GeForce 8800GTX

CUDA-Simple: 14.8x faster, 33% of fastest GPU kernel

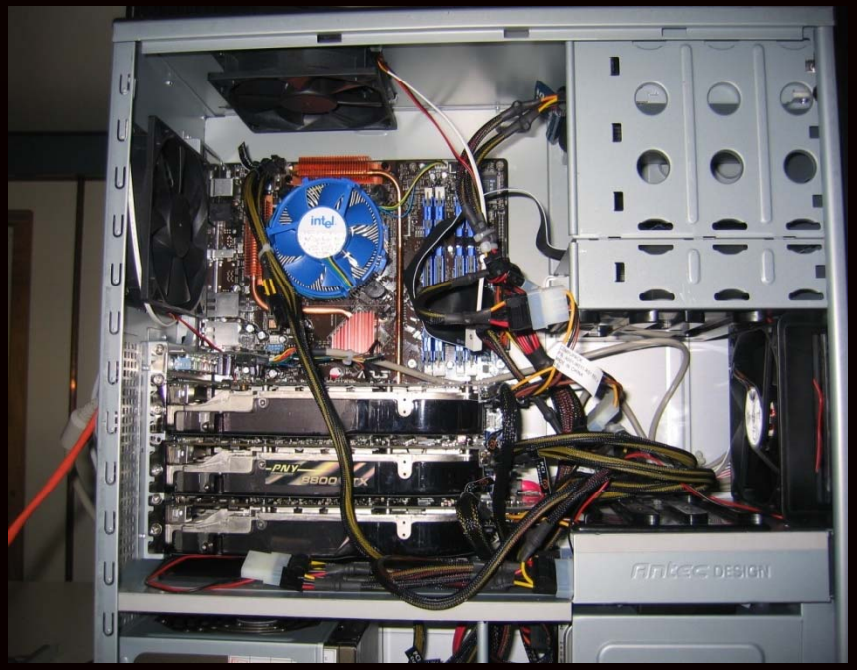
GPU computing. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips. *Proceedings of the IEEE*, 2008. In press.

http://www.ks.uiuc.edu/Research/namd/

SIGGRAPH2008

Multi-GPU DCS Performance

- **Effective memory bandwidth scales with GPUs**
  - PCIe bus bandwidth not a bottleneck for this algorithm
- **3-GPU DCS achieves:**
  - 117 GEvals/sec,
  - 863 GFLOPS
  - Power: 700 W running flat out on all 4 cores, and 3 GPUs
- **A 4-GPU (dual D870) achieves:**
  - 157 GEvals/sec
  - 1.16 TFLOPS



Quad-core 2.67 GHz Intel QX6700

3x G80 GPUs

RHEL4 Linux

Courtesy of: James Phillips, John Stone, Klaus Schulten

http://www.ks.uiuc.edu/Research/namd/


QUESTIONS?
NVIDIA.
SIGGRAPH2008