

Parallel Computing with CUDA

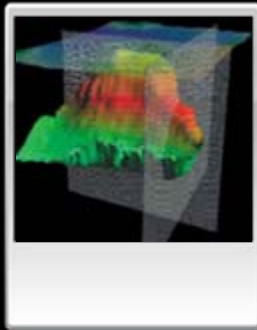


Agenda

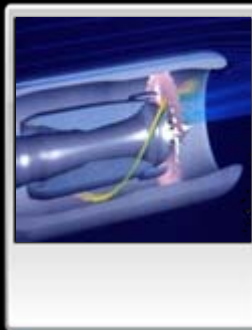
- **Introduction & Motivation**
- **The Myth of GPU Computing**
- **CUDA Programming Model**
- **Parallel Algorithms in CUDA**
- **CUDA Tools and Resources**
- **Sample Applications**

Introduction & Motivation

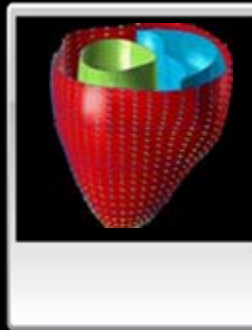
Future Science & Engineering Breakthroughs Hinge on Computing



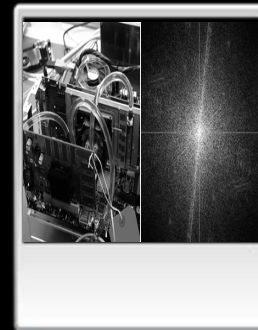
**Computational
Geoscience**



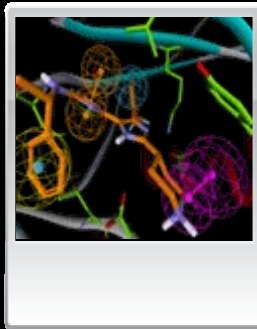
**Computational
Modeling**



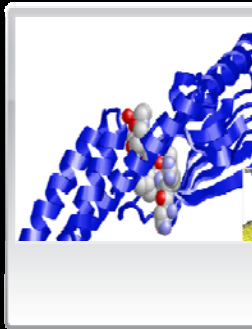
**Computational
Medicine**



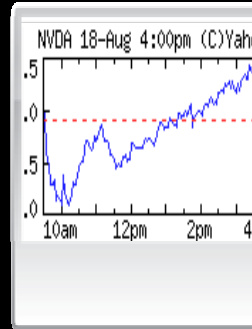
**Computational
Physics**



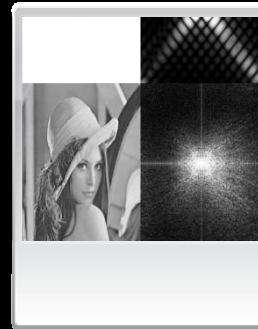
**Computational
Chemistry**



**Computational
Biology**



**Computational
Finance**



**Image
Processing**

Faster is not “Just faster”

2-3x

“Just faster”

Do a little more, wait a little less

Doesn't change how you work

5-10x

“Significant”

Worth upgrading

Worth rewriting (parts of) your application

100x+

“Fundamentally Different”

Worth considering a new platform

Worth re-architecting your application

Makes new applications possible

Drives down “time to discovery”

Creates fundamental changes in science

CPU has hit a wall

- Even **40%** increasing on performance is considered an achievement!
- Clock speed is stuck at 3GHz for years
- No matter how the cache is designed, it will never be large enough
- Adding more cores makes programming more difficult than ever. The magical compiler for multi-threading only exists on paper

Why GPU?

- **GPUs have evolved into highly parallel machines**
 - Increasing performance much faster rate than CPU
 - Already provide 100s of cores
 - 2x more core means 2x more performance
 - Fully programmable in C
 - **NO graphics programming knowledge needed**
 - Lots of compute power and memory bandwidth
 - Widely available
 - **Over 100M CUDA-capable GPUs shipped as of 2008**
- **GPUs enable parallel computing for the masses!**

Parallel Computing on GPU

● GPUs are massively multithreaded manycore chips

- NVIDIA GPU products have up to **240** scalar processors
- Over **23,000** concurrent threads in flight
- **1 TFLOP** of performance (Tesla)

● Enabling new science and engineering

- By drastically reducing time to discovery
- Engineering design cycles: from days to minutes, weeks to days

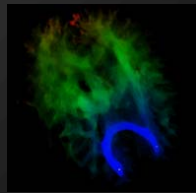
● Enabling new computer science

- By reinvigorating research in parallel algorithms, programming models, architecture, compilers, and languages

**The complexity of the problem is hidden
by the simplicity of the solution.**

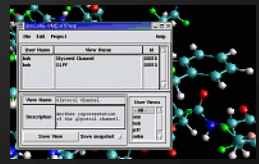
CUDA

CUDA Use Cases



146X

Volumetric white matter visualization



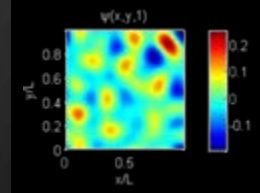
36X

Ionic placement for molecular dynamics



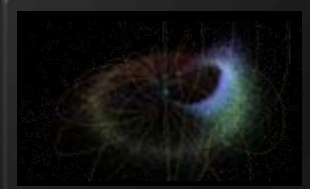
18X

Transcoding HD H.264 video stream



17X

Isotropic turbulence simulation in Matlab



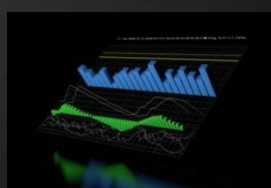
100X

Astrophysics n-body simulation



149X

Financial simulation of LIBOR Model



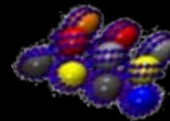
47X

M-script API for Linear Algebra



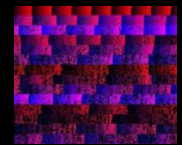
20X

Ultrasound imaging for cancer



24X

Molecular dynamics



30X

String matching for gene sequences

The Myth of GPU Computing

Myth of GPU Computing

- **GPUs layer normal programs on top of graphics**
- **GPU architectures are**
 - **Very wide (1000s) SIMD machines**
 - **...on which branching is impossible or prohibitive**
 - **...with 4-wide vector registers**
- **GPUs are power-inefficient**
- **GPUs don't do real floating point**

Myth of GPU Computing

- GPUs layer normal programs on top of graphics

No: CUDA compiles directly into the hardware

- GPU architectures are

- Very wide (1000s) SIMD machines

- ...on which branching is impossible or prohibitive

- ...with 4-wide vector registers

- GPUs are power-inefficient

- GPUs don't do real floating point

Myth of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPU architectures are
 - Very wide (1000s) SIMD machines **No, warps are 32-wide**
 - ...on which branching is impossible or prohibitive
 - ...with 4-wide vector registers
- GPUs are power-inefficient
- GPUs don't do real floating point

Myth of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPU architectures are
 - ~~Very wide (1000s) SIMD machines~~
 - ...on which branching is impossible or prohibitive **Nope**
 - ...with 4-wide vector registers
- GPUs are power-inefficient
- GPUs don't do real floating point

Myth of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPU architectures are
 - ~~Very wide (1000s) SIMD machines~~
 - ~~...on which branching is impossible or prohibitive~~
 - ...with 4-wide vector registers **No. All are scalars.**
- GPUs are power-inefficient
- GPUs don't do real floating point

Myth of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPU architectures are
 - ~~Very wide (1000s) SIMD machines~~
 - ~~...on which branching is impossible or prohibitive~~
 - ~~...with 4-wide vector registers~~
- GPUs are power-inefficient
 - No. 4x ~ 10x perf/watt advantage**
- GPUs don't do real floating point

Double Precision Floating Point

	NVIDIA Tesla T10	SSE2	Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	All 4 IEEE, round to nearest, zero, Inf, -Inf	All 4 IEEE, round to nearest, zero, inf, -inf	All 4 IEEE, round to nearest, zero, inf, -inf
Denormal handling	Full speed	Supported, costs 1000's of cycles	Supported only for results, not input operands (input denormals flushed-to-zero)
NaN support	Yes	Yes	Yes
Overflow and Infinity support	Yes	Yes	Yes
Flags	No	Yes	Yes
FMA	Yes	No	Yes
Square root	Software with low-latency FMA-based convergence	Hardware	Software only
Division	Software with low-latency FMA-based convergence	Hardware	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit + step
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit + step
$\log_2(x)$ and 2^x estimates accuracy	23 bit	No	No

GPU Single Floating Point Features

	G80	SSE	IBM Altivec	Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	Round to nearest and round to zero	All 4 IEEE, round to nearest, zero, inf, -inf	Round to nearest only	Round to zero/truncate only
Denormal handling	Flush to zero	Supported, 1000's of cycles	Supported, 1000's of cycles	Flush to zero
NaN support	Yes	Yes	Yes	No
Overflow and Infinity support	Yes, only clamps to max norm	Yes	Yes	No, infinity
Flags	No	Yes	Yes	Some
Square root	Software only	Hardware	Software only	Software only
Division	Software only	Hardware	Software only	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit	12 bit
log2(x) and 2^x estimates accuracy	23 bit	No	12 bit	No

Do GPUs do real IEEE 754 FP?

- **G8x/GT200 GPU FP is IEEE 752**
 - Comparable to other processors
 - More precise / usable in some ways
 - Less precise in other ways
- **GPU FP getting better every generation**
 - Double precision supported in GT200
 - Goal: best of class in by 2009

Myth of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPU architectures are
 - ~~Very wide (1000s) SIMD machines~~
 - ~~...on which branching is impossible or prohibitive~~
 - ~~...with 4-wide vector registers~~
- ~~GPUs are power inefficient~~
- ~~GPUs don't do real floating point~~

CUDA Programming Model

CUDA Goal: Easy to Program

● Strategies:

- Let programmers focus on parallel algorithms
 - *not* mechanics of a parallel programming language
 - **Use C/C++** with minimal extensions
- **Enable heterogeneous systems** (i.e., CPU+GPU)
 - CPU & GPU are separate devices with separate DRAMs
- Simple parallel abstractions
 - Simple barrier synchronization
 - Shared memory semantics
 - **Hardware-managed** hierarchy of threads

Goal: Performance per millimeter

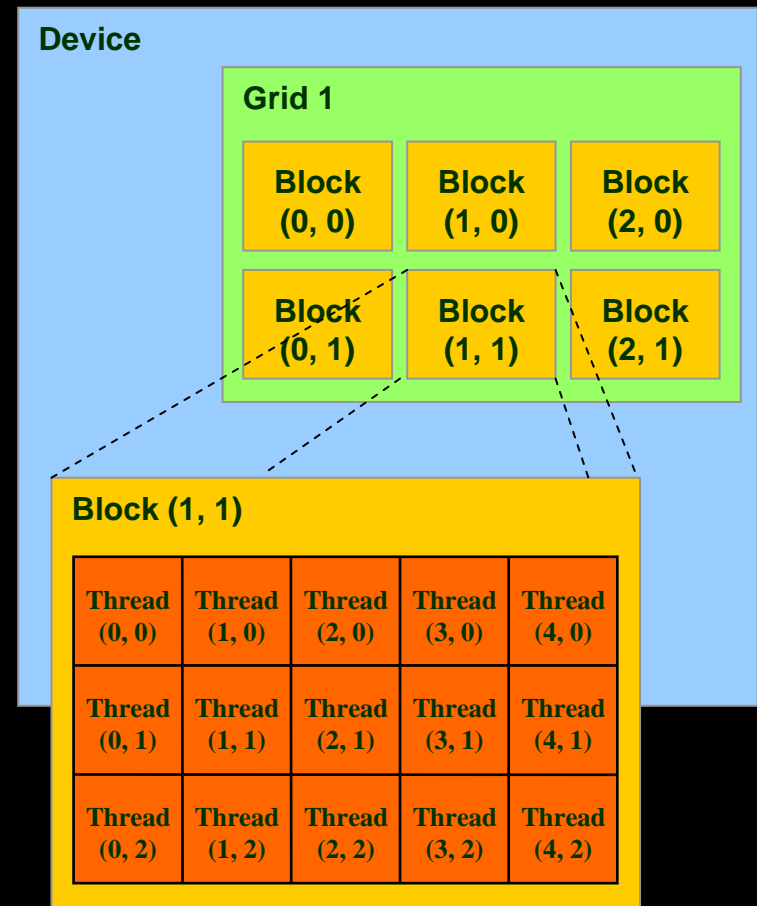
- For GPUs, performance == throughput
- Strategy: hide latency with computation not cache
 - Heavy multithreading
- Implication: need many threads to hide latency
 - Occupancy – typically need 128 threads/SM minimum
 - Multiple thread blocks/SM good to minimize effect of barriers
- Strategy: Single Instruction Multiple Thread (**SIMT**)
 - Balances performance with ease of programming

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

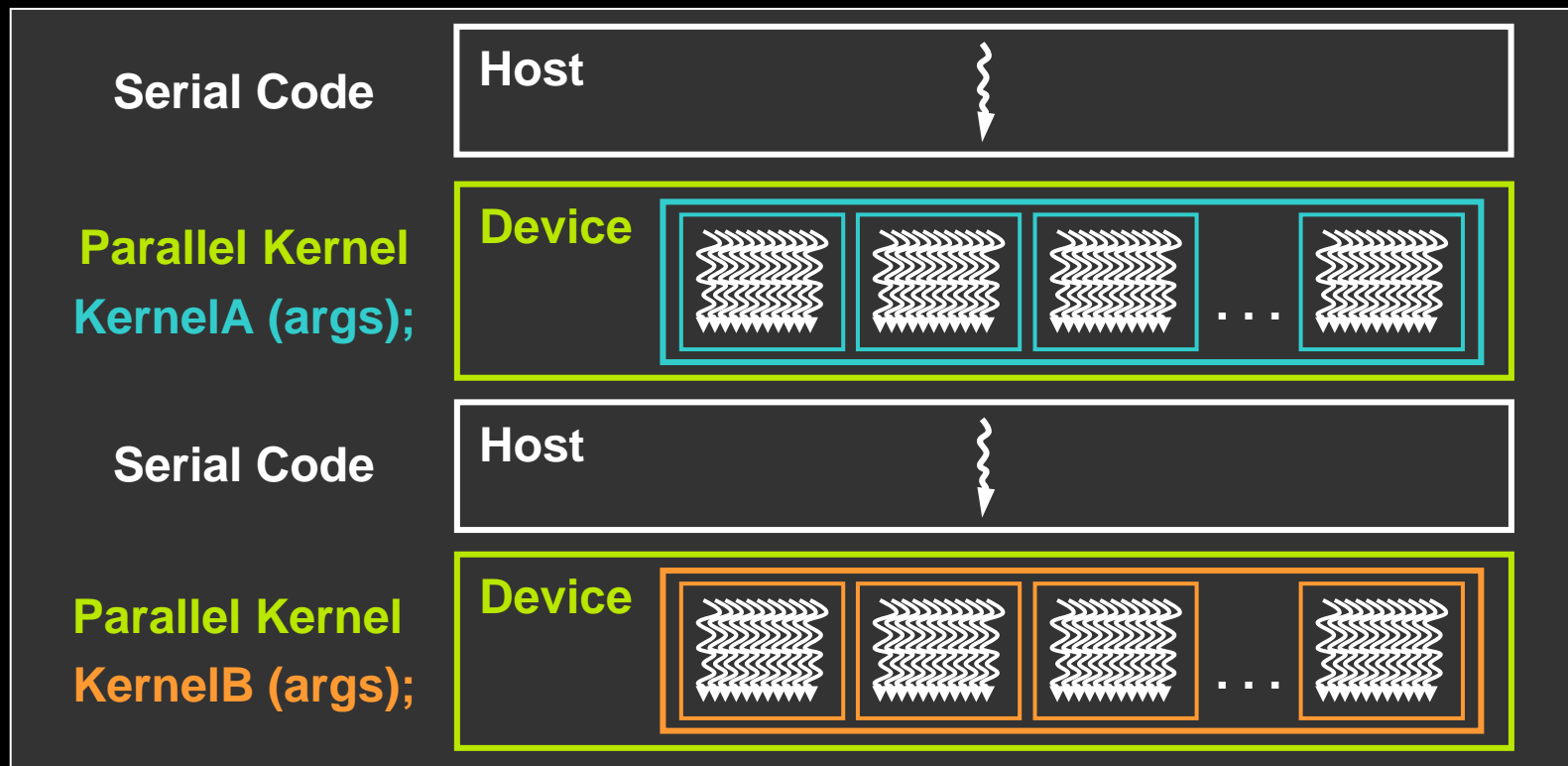
IDs and Dimensions

- **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`



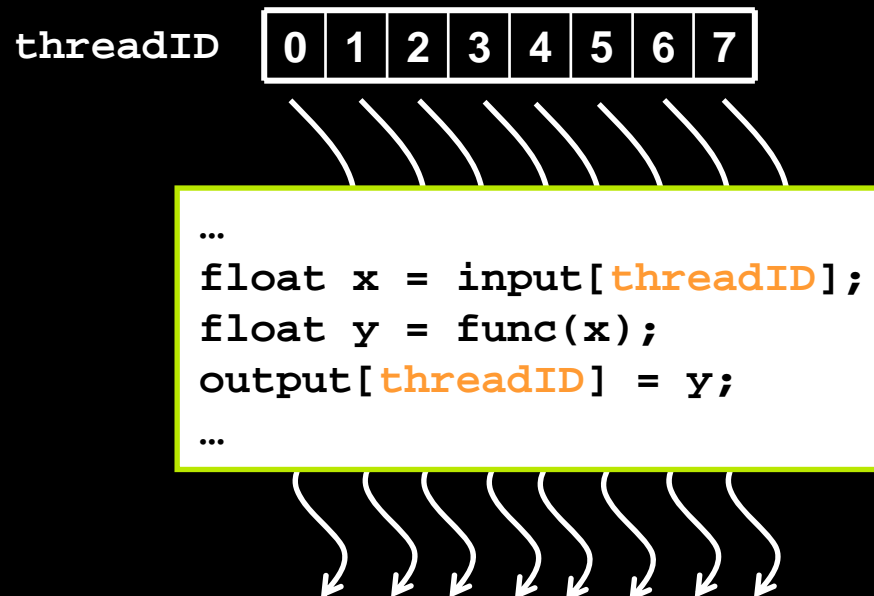
Heterogeneous Programming

- **CUDA = serial program with parallel kernels, all in C**
 - Serial C code executes in a **host** thread (i.e. **CPU** thread)
 - Parallel kernel C code executes in many **device** threads across multiple processing elements (i.e. **GPU** threads)



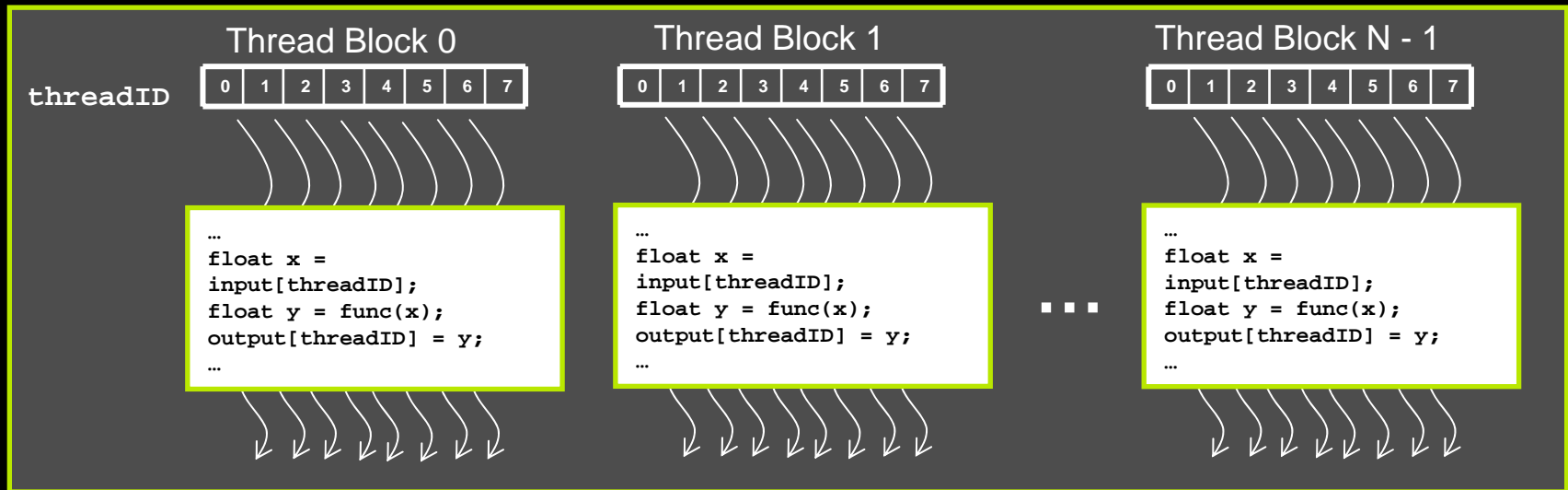
Kernel = Many Concurrent Threads

- One kernel is executed at a time on the device
- Many threads execute each kernel
 - Each thread has its own program counter, variables (registers), processor state, etc.
 - Threads work different data based on its **threadID**



Hierarchy of Concurrent Threads

- **Thread block = virtualized multiprocessor**
 - **Kernel = grid of thread blocks**



- **Threads in the same block can be synchronized with barriers**

```
scratch[threadID] = begin[threadID];  
__syncthreads();  
int left = scratch[threadID - 1];
```

**Threads
wait at the barrier
until all threads
in the same block
reach the barrier**

Example: Increment Array Elements

Increment N-element vector a by scalar b



Let's assume $N=16$, $\text{blockDim}=4$ \rightarrow 4 blocks

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



$\text{blockIdx}.x=0$
 $\text{blockDim}.x=4$
 $\text{threadIdx}.x=0,1,2,3$
 $\text{idx}=0,1,2,3$



$\text{blockIdx}.x=1$
 $\text{blockDim}.x=4$
 $\text{threadIdx}.x=0,1,2,3$
 $\text{idx}=4,5,6,7$



$\text{blockIdx}.x=2$
 $\text{blockDim}.x=4$
 $\text{threadIdx}.x=0,1,2,3$
 $\text{idx}=8,9,10,11$



$\text{blockIdx}.x=3$
 $\text{blockDim}.x=4$
 $\text{threadIdx}.x=0,1,2,3$
 $\text{idx}=12,13,14,15$

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, 16);
}
```

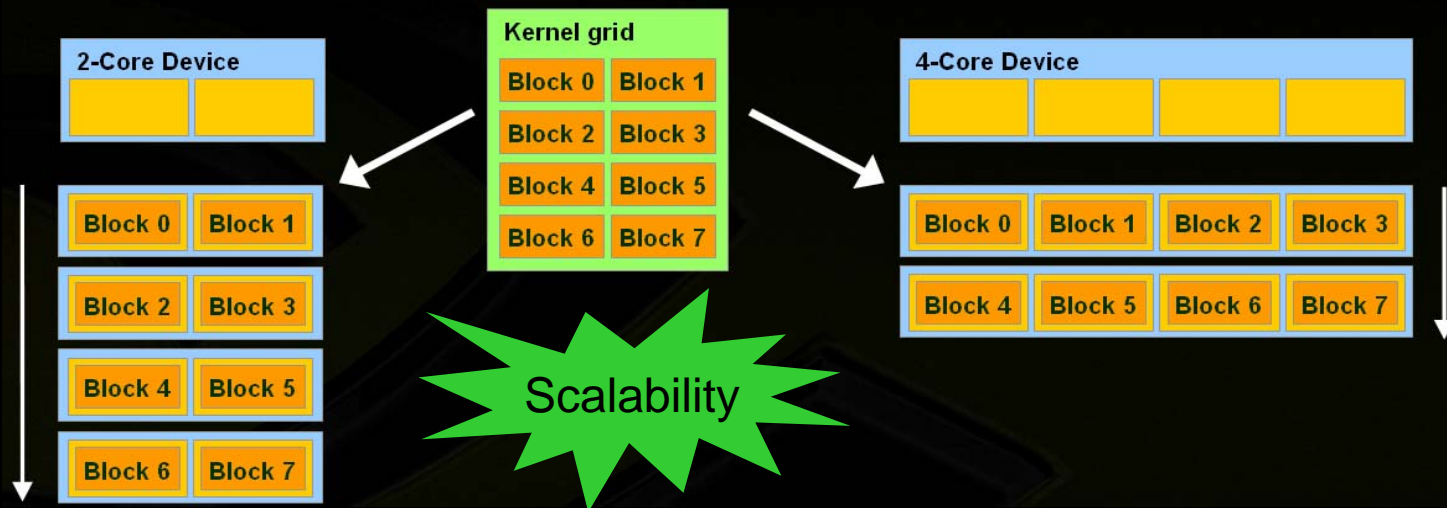
CUDA program

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

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

Scalability: Make Blocks Independent

- **Thread blocks can run in any order**
 - Concurrently or sequentially
 - Hardware does not synchronize between blocks
- **This independence gives scalability:**
 - A kernel scales across any number of parallel cores

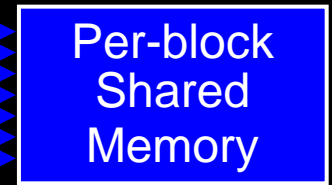
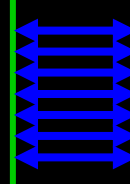
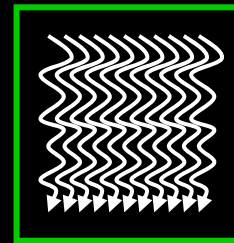


Memory Model of Threads and Thread Blocks

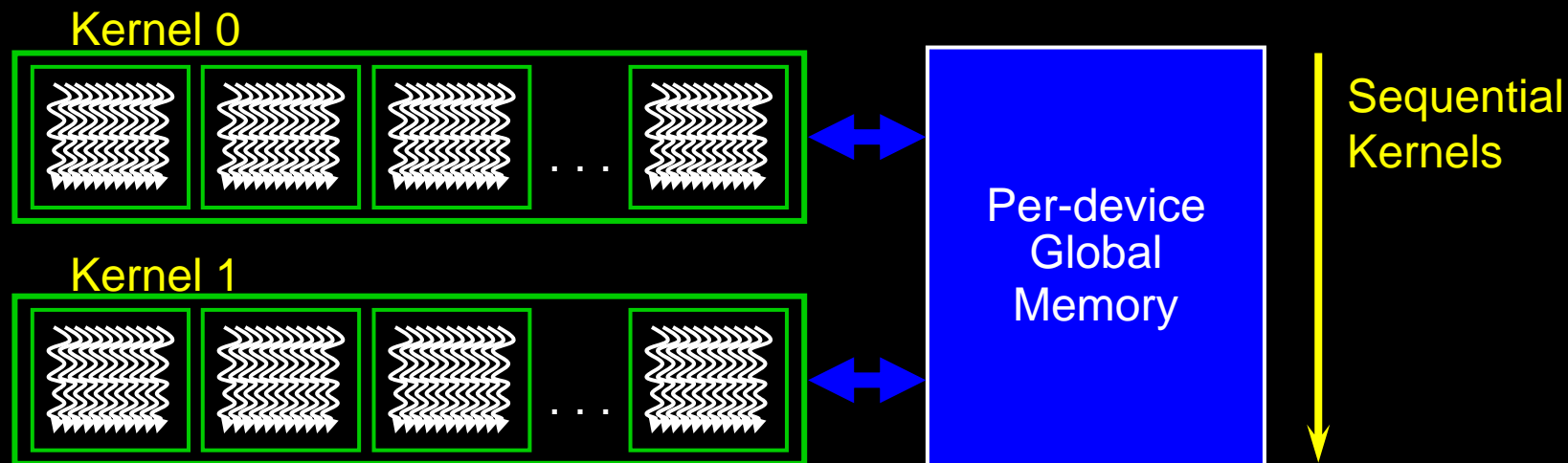
Thread



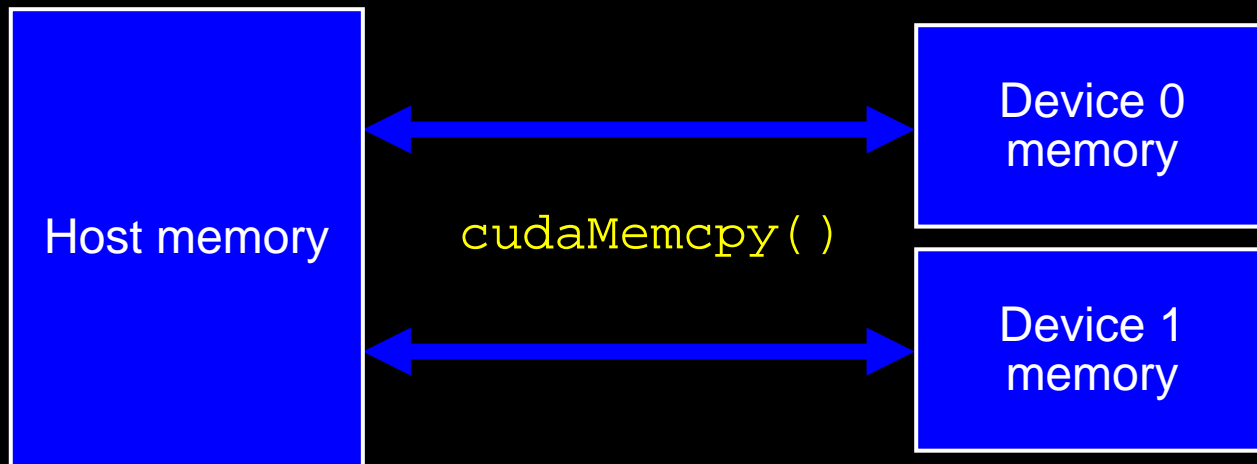
Block



Memory Model with Each GPU



Memory Model of Host with Multiple GPUs



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);

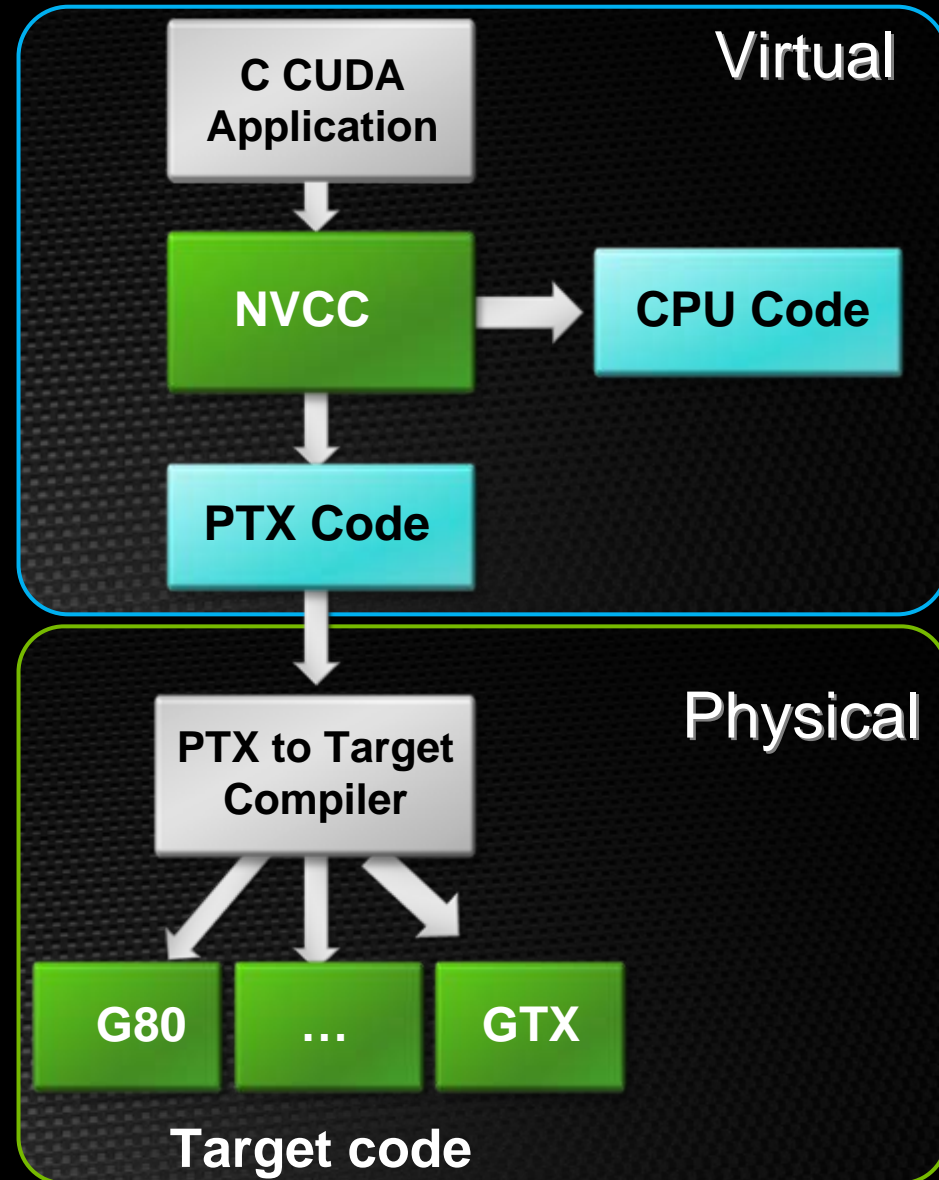
// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b, N);

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

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

Compiling CUDA

- Any source file containing CUDA language extensions must be compiled with **NVCC**
 - NVCC separates code running on the host from code running on the device
- Two-stage compilation:
 - Virtual ISA
 - Parallel Thread eXecution
 - Device-specific binary object



Debugging Using the Device Emulation Mode

- An executable compiled in **device emulation mode** (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
 - No need of any device and CUDA driver
 - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
 - Use host native debug support (breakpoints, inspection, etc.)
 - Access any device-specific data from host code and vice-versa
 - Call any host function from device code (e.g. `printf`) and vice-versa
 - Detect deadlock situations caused by improper usage of `__syncthreads`

Parallel Algorithms in CUDA

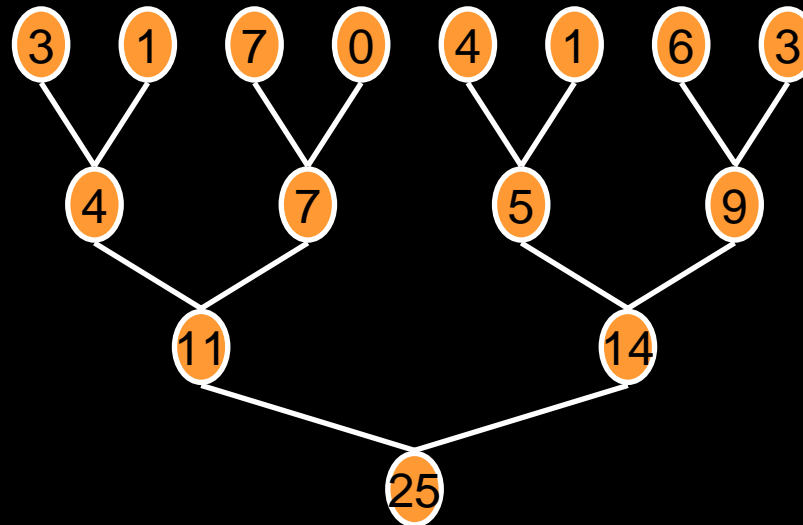
Common Situations in Parallel Computation

- **Many parallel threads need to generate a single result value**
 - **Reduction**
- **Many parallel threads that need to partition data**
 - **Split**
- **Many parallel threads and variable output per thread**
 - **Compact / Allocate**

Parallel Reductions

- **Common Data Parallel Operation**

- **Reduce** vector to a single value
- **Operator:** +, *, min/max, AND/OR
- **Tree-based implementation**



Split Operation

- Given an array of true and false elements (and payloads)

Flag

T	F	F	T	F	F	T	F
---	---	---	---	---	---	---	---

Payload

3	1	7	0	4	1	6	3
---	---	---	---	---	---	---	---

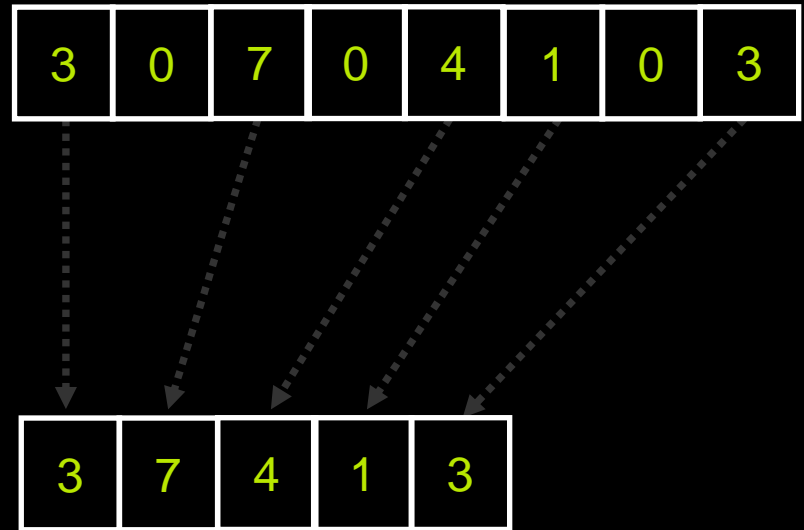
- Return an array with all true elements at the beginning

T	T	T	F	F	F	F	F
---	---	---	---	---	---	---	---

3	0	6	1	7	4	1	3
---	---	---	---	---	---	---	---

Variable Output Per Thread (1): Compact

- Remove null elements



Variable Output Per Thread (2): Allocation

- Allocate Variable Storage Per Thread

2	1	0	3	2
---	---	---	---	---

A	C
B	

D	G
E	H
F	

Parallel Prefix Sum (Scan)

- Given an array $A = [a_0, a_1, \dots, a_{n-1}]$ and a binary associative operator \oplus with identity I ,

$$\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})]$$

- Example: if \oplus is addition, then scan on the set

[3 1 7 0 4 1 6 3]

returns the set

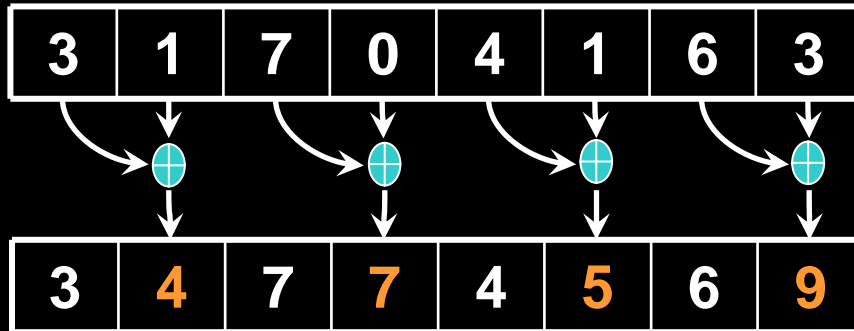
[0 3 4 11 11 15 16 22]

Build the Sum Tree

3	1	7	0	4	1	6	3
---	---	---	---	---	---	---	---

Assume array is already in shared memory

Build the Sum Tree

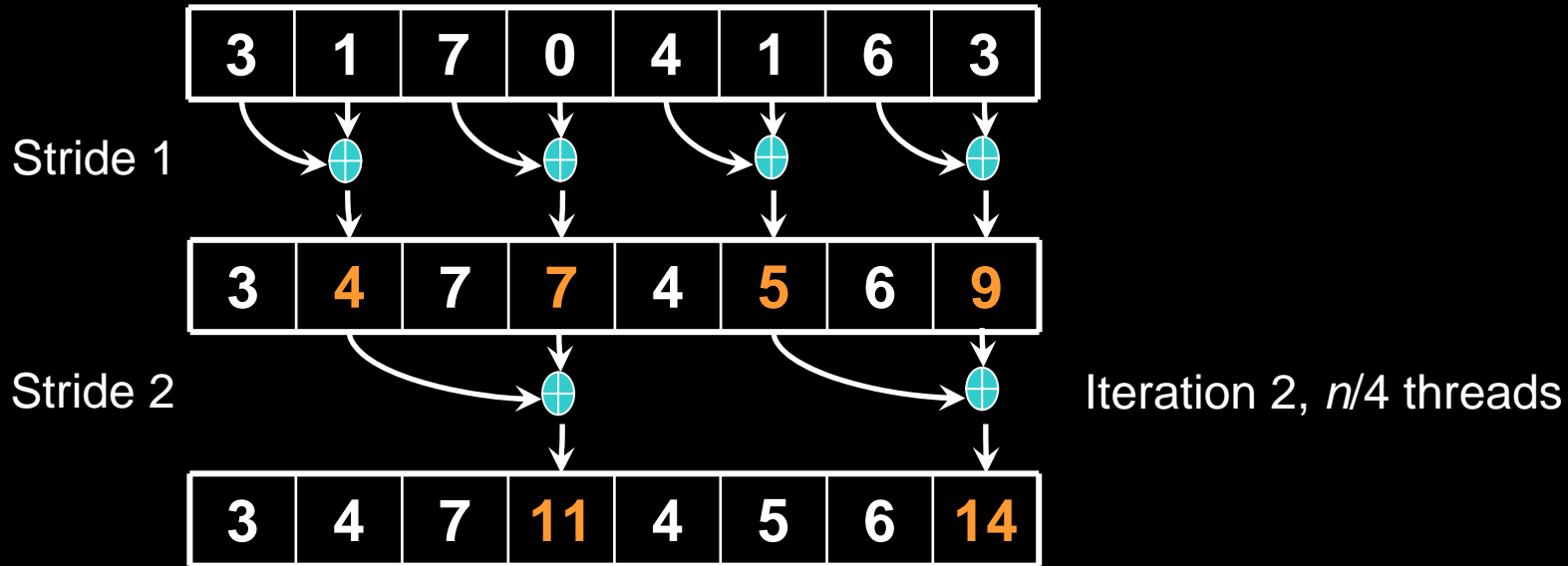


Iteration 1, $n/2$ threads

Each \oplus corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value

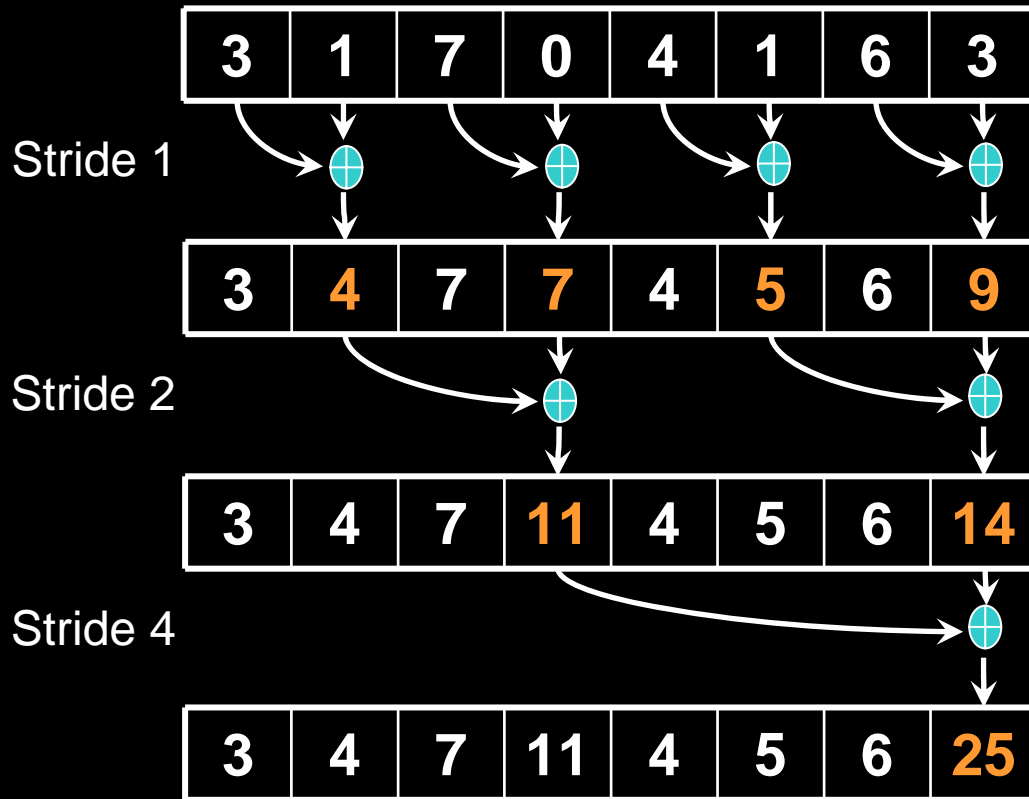
Build the Sum Tree



Each \oplus corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value

Build the Sum Tree



Iteration $\log(n)$, 1 thread

Each \oplus corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

Zero the Last Element

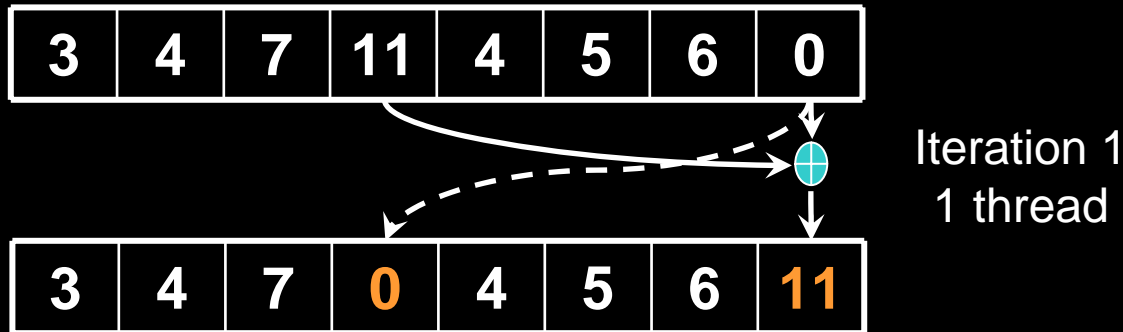
3	4	7	11	4	5	6	0
---	---	---	----	---	---	---	---

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

Build Scan From Partial Sums

3	4	7	11	4	5	6	0
---	---	---	----	---	---	---	---

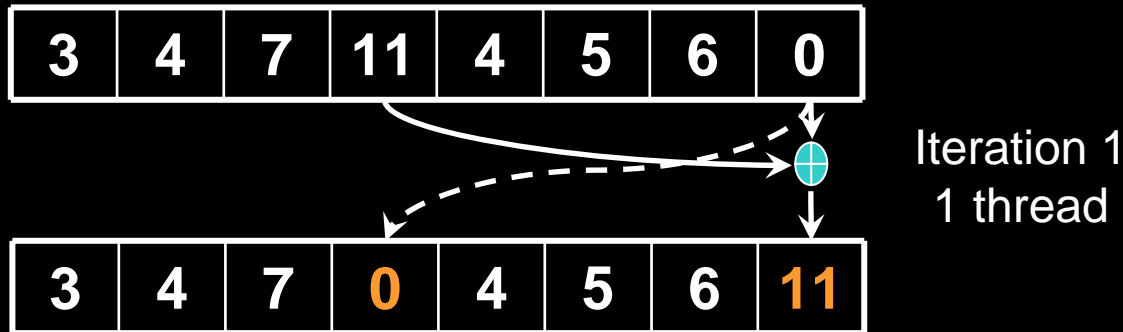
Build Scan From Partial Sums



Each \oplus corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

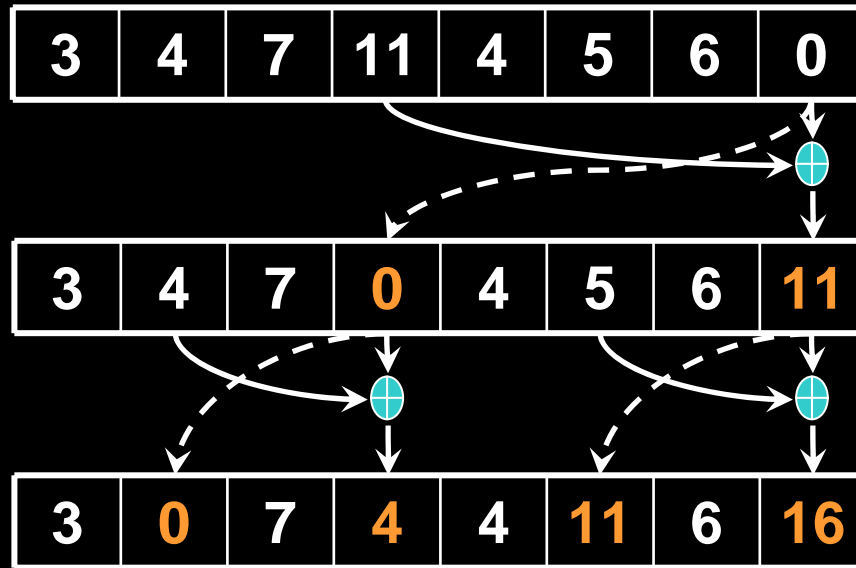
Build Scan From Partial Sums




Each \oplus corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Scan From Partial Sums

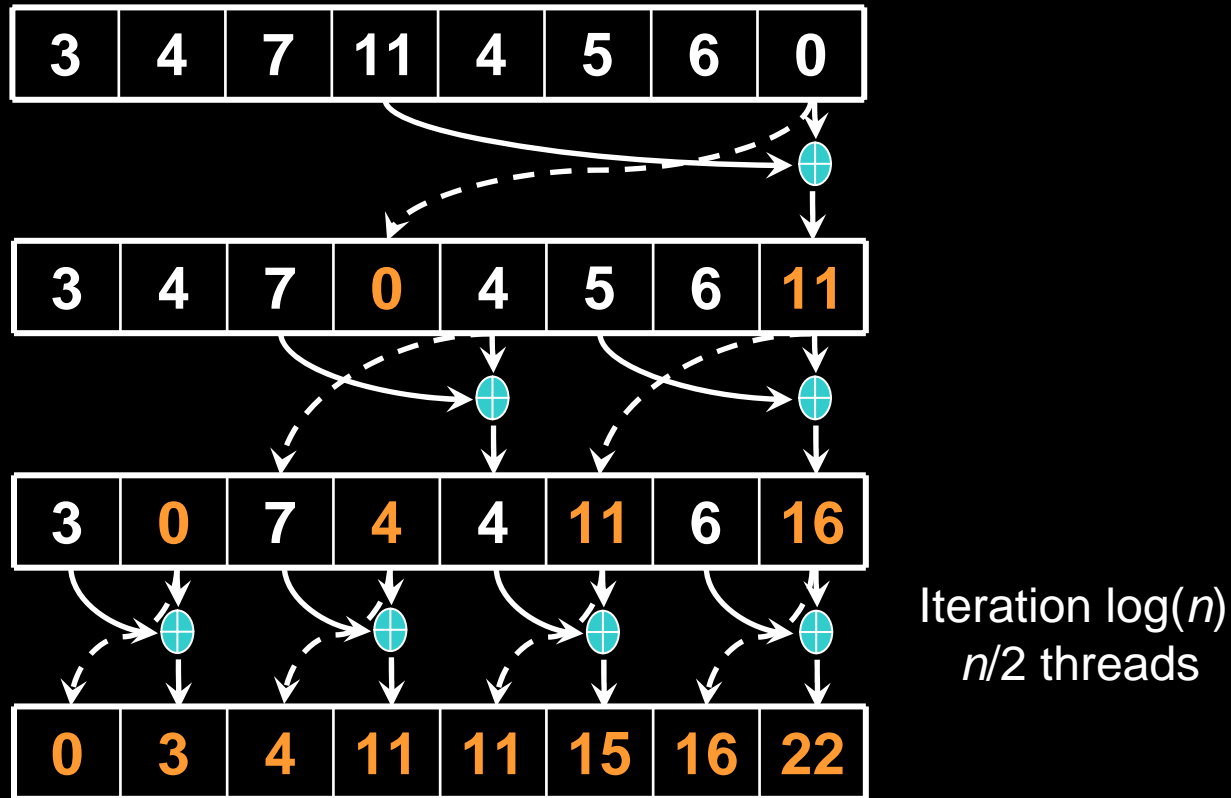


Iteration 2
2 threads

Each  corresponds to a single thread.

Iterate $\log(n)$ times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Scan From Partial Sums

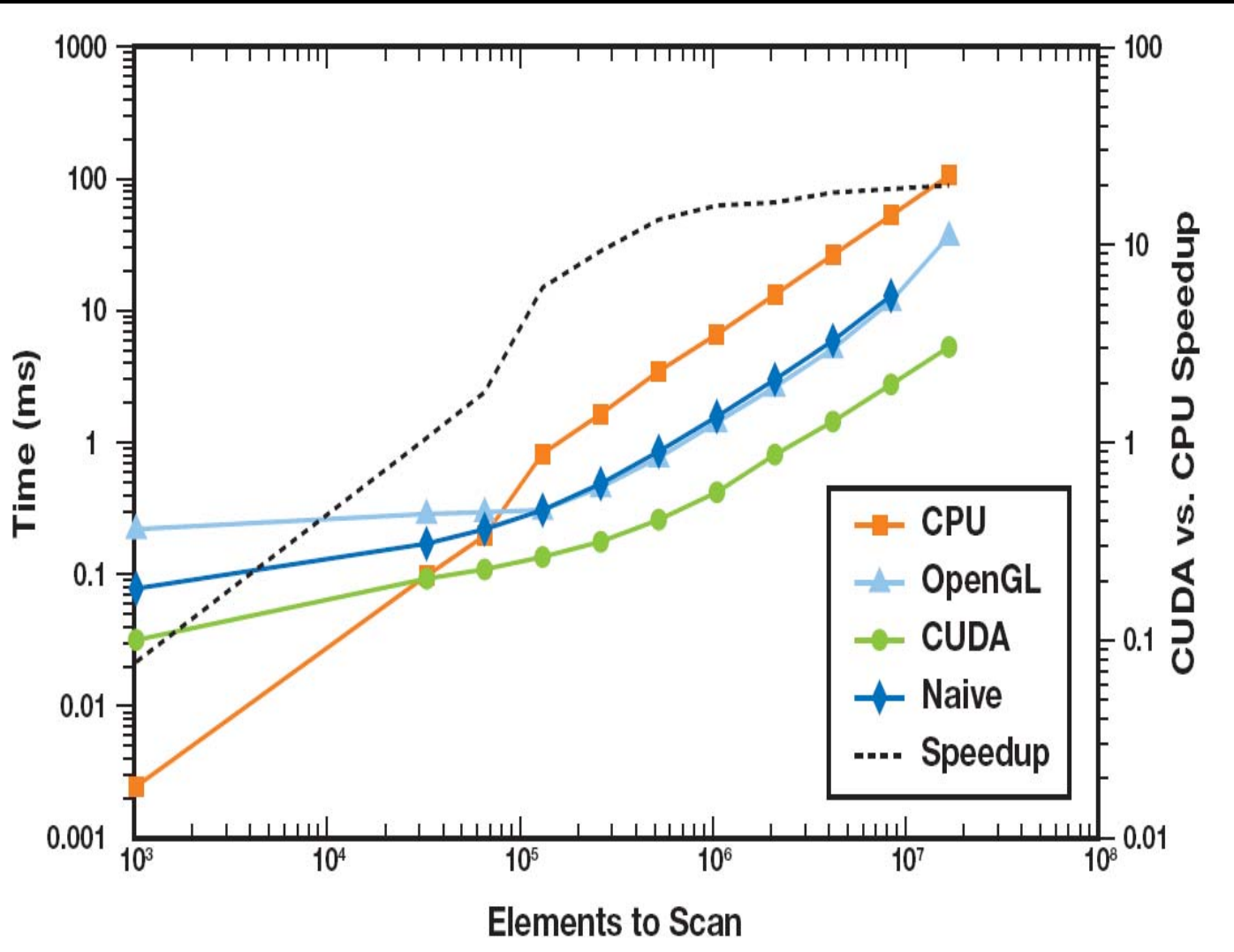


Done! We now have a completed scan that we can write out to device memory.

Total steps: $2 * \log(n)$.

Total work: $2 * (n-1)$ adds = $O(n)$ **Work Efficient!**

CUDA Scan Performance



**GPU vs.
CPU**

20x

**CUDA
vs.
OpenGL**

7x

GeForce 8800 GTX, Intel Core2 Duo Extreme 2.93 GHz

Application: Stream Compaction

A	B	C	D	E	F	G	H
---	---	---	---	---	---	---	---

Input: we want to preserve the gray elements

1	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

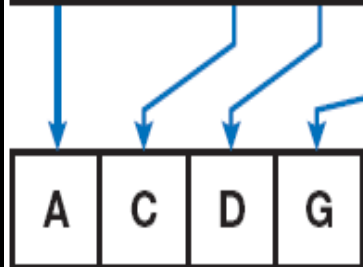
Set a "1" in each gray input

0	1	1	2	3	3	3	4
---	---	---	---	---	---	---	---

Scan

A	B	C	D	E	F	G	H
---	---	---	---	---	---	---	---

Scatter input to output, using scan result as scatter address



1M elements:
~0.6-1.3ms

16M elements:
~8-20ms

Perf depends on # elements retained

Ap

100	111	010	110	011	101	001	000
-----	-----	-----	-----	-----	-----	-----	-----

Input

0	1	0	0	1	1	1	0
---	---	---	---	---	---	---	---

Split based on least significant bit b

1	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

e = Set a "1" in each "0" input

0	1	1	2	3	3	3	0
---	---	---	---	---	---	---	---

f = Scan the 1s

totalFalses = e[max] + f[max]

0-0+4 =4	1-1+4 =4	2-1+4 =5	3-2+4 =5	4-3+4 =5	5-3+4 =6	6-3+4 =7	7-3+4 =8
-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

t = i - f + totalFalses

0	4	1	2	5	6	7	3
---	---	---	---	---	---	---	---

d = b ? t : f

100	111	010	110	011	101	001	000
-----	-----	-----	-----	-----	-----	-----	-----

Scatter input using d as scatter address

100	010	110	000	111	011	101	001
-----	-----	-----	-----	-----	-----	-----	-----

**Sort 4M 32-bit integers:
165ms**

Perform split operation on each bit using scan

Can also sort each block and merge

- Slower due to cost of merge

CUDA Tools and Resources

CUDA Programming Resources

● **CUDA Toolkit**

- **Compiler and libraries**
- **Free download for Windows, Linux, and Mac OSX**

● **CUDA SDK**

- **Code samples**
- **Whitepapers**

● **Instruction materials**

- **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 parameters
 - Execution (serialization, divergence)

● Debugger

- Demo shown at SC07
- Runs on the GPU

● Emulation mode

- Compile and execute in emulation on CPU
- Allows CPU-style debugging in GPU source

New Features

● **CUDA 2.02 Beta**

- **Beta available on the NVIDIA website**
- **Support for GeForce GTX 260 & 280:**
 - **Double precision**
 - **Integer atomic operations in shared memory**
- **New features:**
 - **3D textures**
 - **Video Decoding Interface with Compute 1.1+ GPU**
 - **Improved and extended Direct3D interoperability**

● **CUDA implementation on multi-core CPUs**

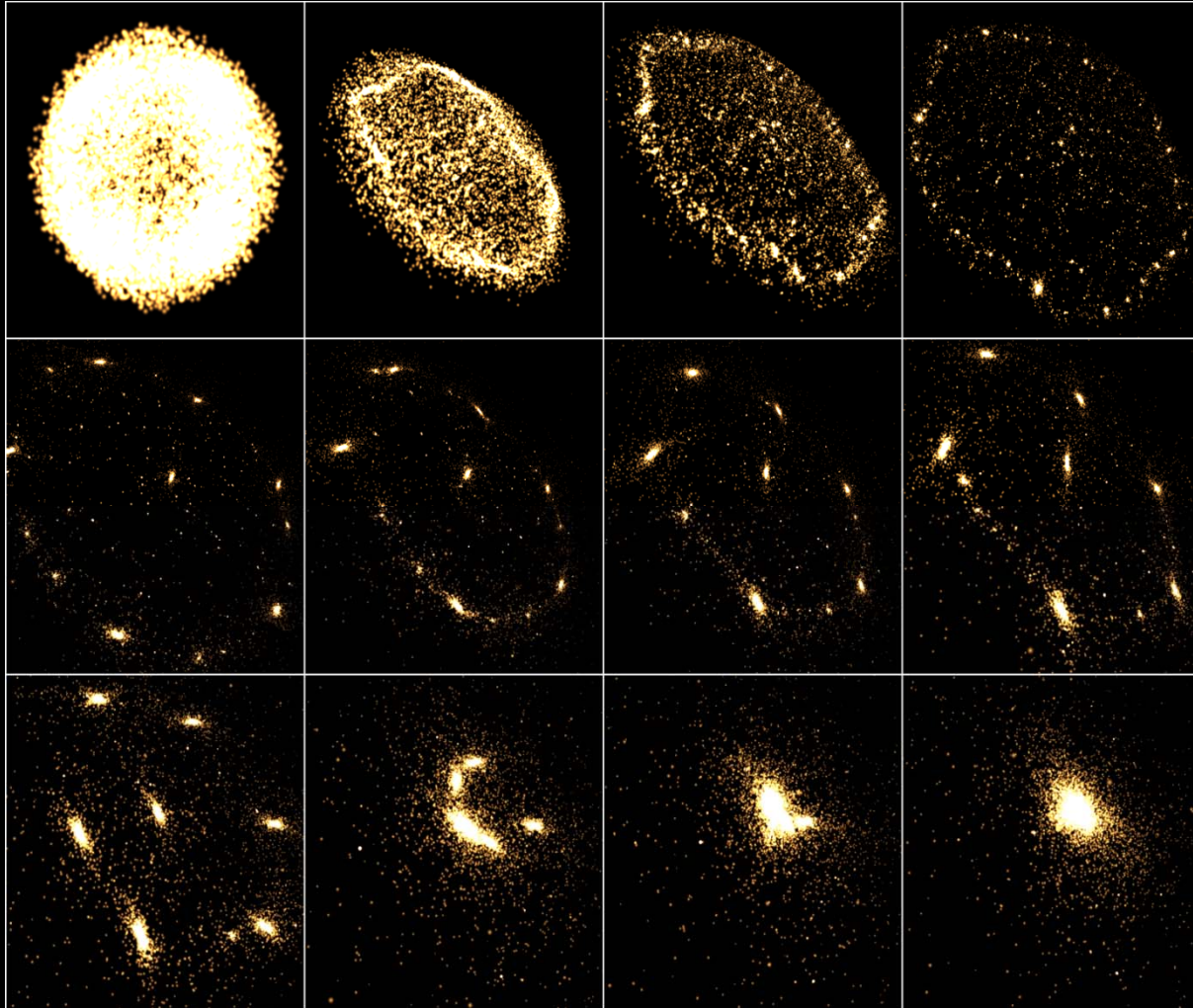
- **Beta coming soon**

Double Precision

- **NVIDIA GPUs (G8x and G9x) are single precision**
 - IEEE 32-bit floating-point precision (“FP32”)
 - You can use double, but it gets demoted to float
- **NVIDIA GPUs (GT200) have double precision**
 - IEEE 64-bit floating-point precision (“FP64”)
 - Double precision will be slower (more register pressure and more cycles)
- **Be explicit about float and double!**
 - Use double only where needed

Sample Applications

CUDA N-Body Simulation



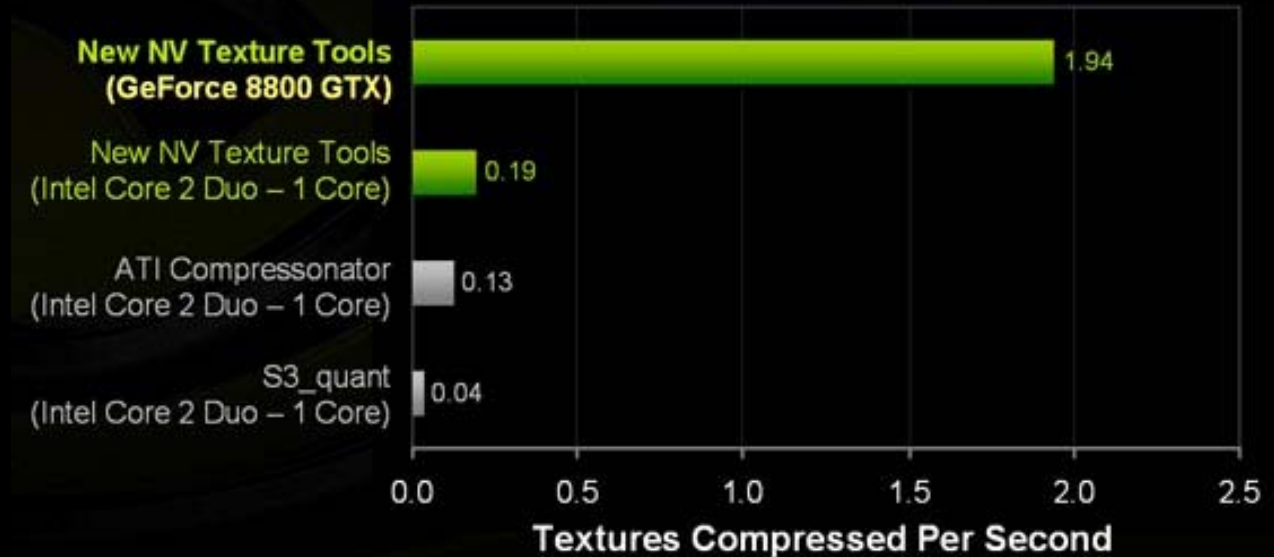
10B interactions/s
16K bodies

44 FPS
x 20 FLOPS / interaction
x $16K^2$ interactions / frame

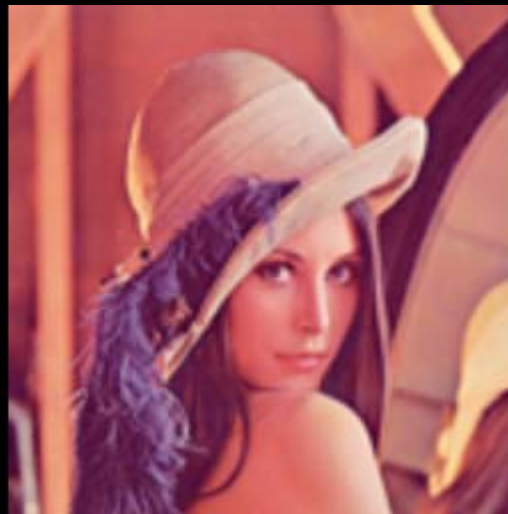
= 240 GFLOP/s on G80

DXT Compression

- Offline
- Runtime
- Real-Time



256x256 RGB = 256 kB



128x128 RGB = 64 kB



256x256 DXT1 = 32 kB

Histogram

- Representation of the distribution of colors in an image
- Applications:
 - Image Analysis
 - HDR Tone Mapping



Reinhard HDR Tonemapping operator



HDR in Valve's source engine

Histogram

- **CUDA Histogram is 300x faster than previous GPGPU approaches**

	64 bins	256 bins
CUDA ¹	6500 MB/s	3676 MB/s
R2VB ²	22.8 MB/s	42.6 MB/s
CPU ³	826 MB/s	1096 MB/s

¹ <http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#histogram64>

² Efficient Histogram Generation Using Scattering on GPUs, T. Sheuermann, AMD Inc, I3D 2007

³ Intel Core 2 @ 2.9 GHz

Game AI Breakdown

- 3 Main AI Computations to Accelerate

- Spatial reasoning
- Decision making
- Path finding

- Computation breakdown:

- For “bot” simulation:

- Spatial reasoning: 35%
- Path finding: 65%

- For crowd simulation:

- Spatial reasoning: 5%
- Path finding: 95%

- Decision making always negligible

- Biggest opportunity for GPU acceleration is **path finding**



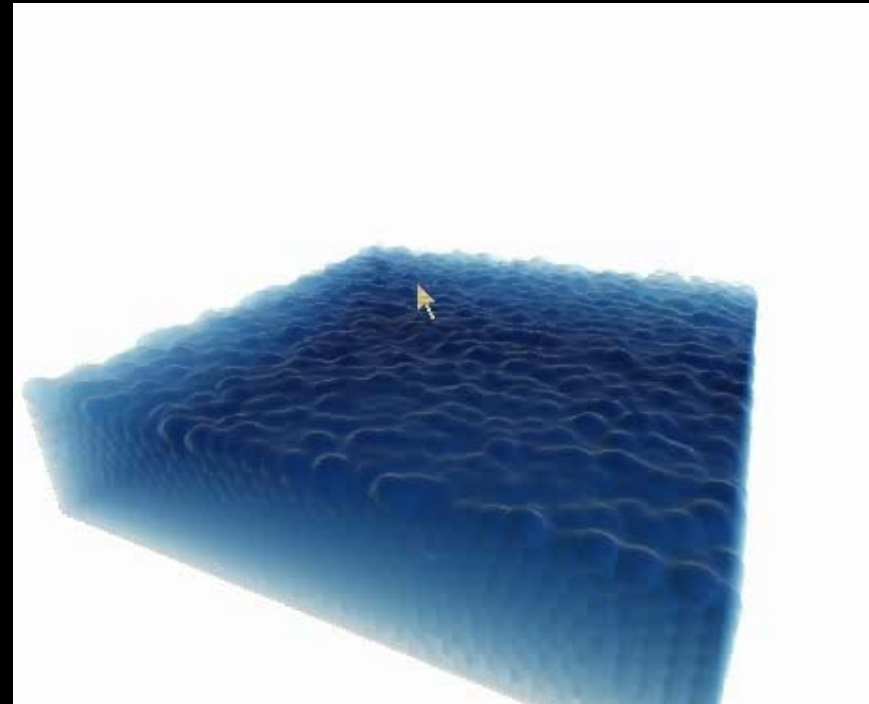
A* Algorithm

- **Commonly used path finding algorithm**
 - **A* itself is not very parallel**
 - **Parallelism comes from computing many separate A* paths in parallel**
 - **Many units moving simultaneously**
 - **Massive worlds with thousands of characters**

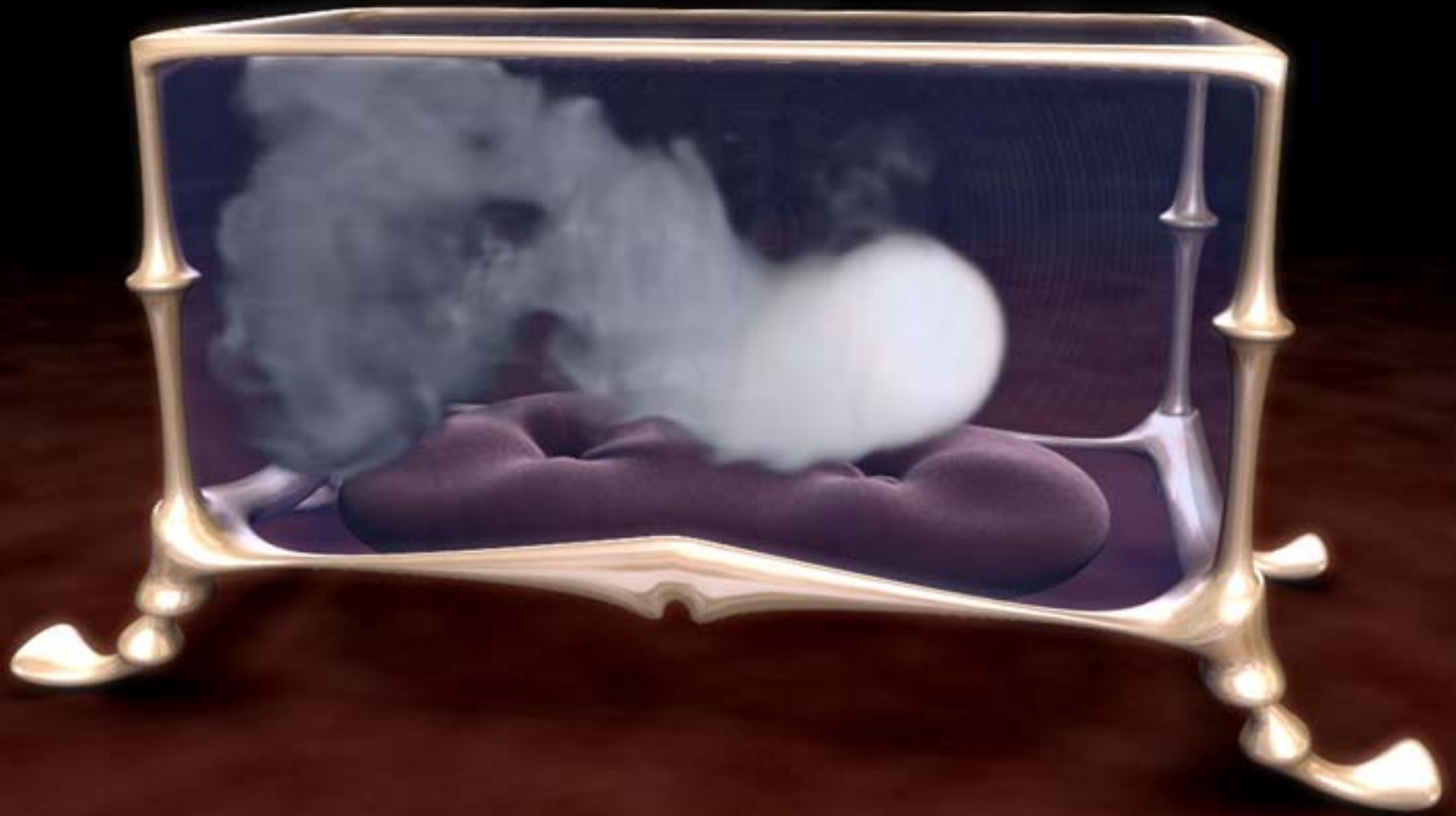


CUDA Particle-Based Fluid

- Grid-based fluids have limitations for use in games
 - Expensive, constrained to a box
- Smoothed Particle Hydrodynamics (SPH) simulates fluid as a collection of interacting particles
 - Localized collisions and pressure distribution
- CUDA enables **dynamic** construction of a uniform grid **data structure** to accelerate neighborhood computations
- 32K particles at 60fps
GeForce 8800 GTX



GPU Fluid Simulation



CUDA and Physics: PhysX implemented in Cuda

● Typical physics core simulation features

- Rigid Body Dynamics
- Universal Collision-Detection
- Joints, Springs and Motors
- Advanced ragdoll and vehicle constraints

● High performance realism (beyond the basics)

● Volumetric Fluids (SPH)

- CCD with rigid bodies and static geometry
- One- or two-way interaction with rigid bodies

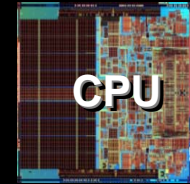
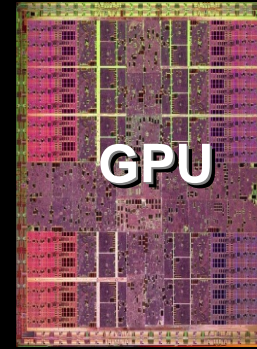
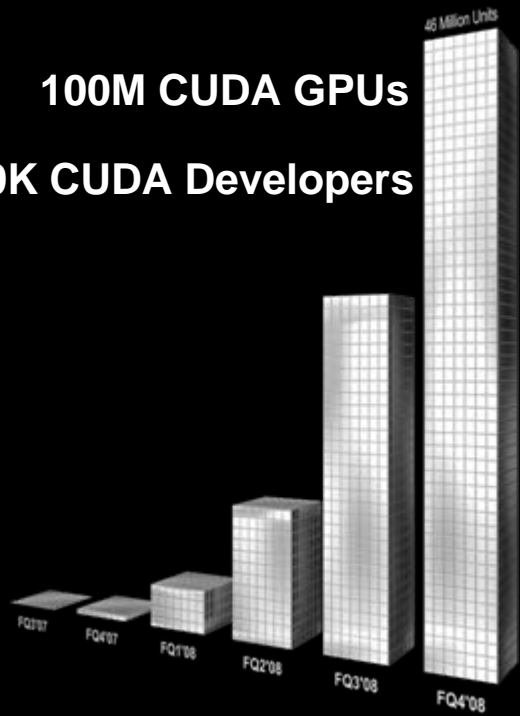
● Cloth and Soft Bodies

- Attachment and CCD with rigid bodies and static geometry
- Cloth self collision
- Tearing
- Derivatives: Sheet metal and vegetation

Where to go from here

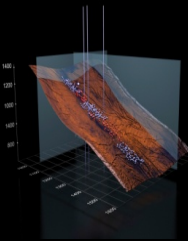
- **Get CUDA Toolkit, SDK, and Programming Guide:**
<http://developer.nvidia.com/CUDA>
- **CUDA works on all NVIDIA 8-Series GPUs (and later)**
 - GeForce, Quadro, and Tesla
- **Talk about CUDA:** <http://forums.nvidia.com>

100M CUDA GPUs
60K CUDA Developers

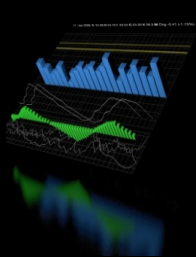


CUDA

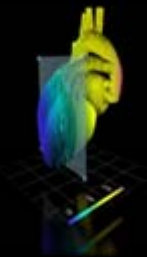
Heterogeneous Computing



Oil & Gas



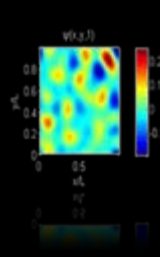
Finance



Medical



Biophysics



Numerics



Audio



Video



Imaging

Questions?