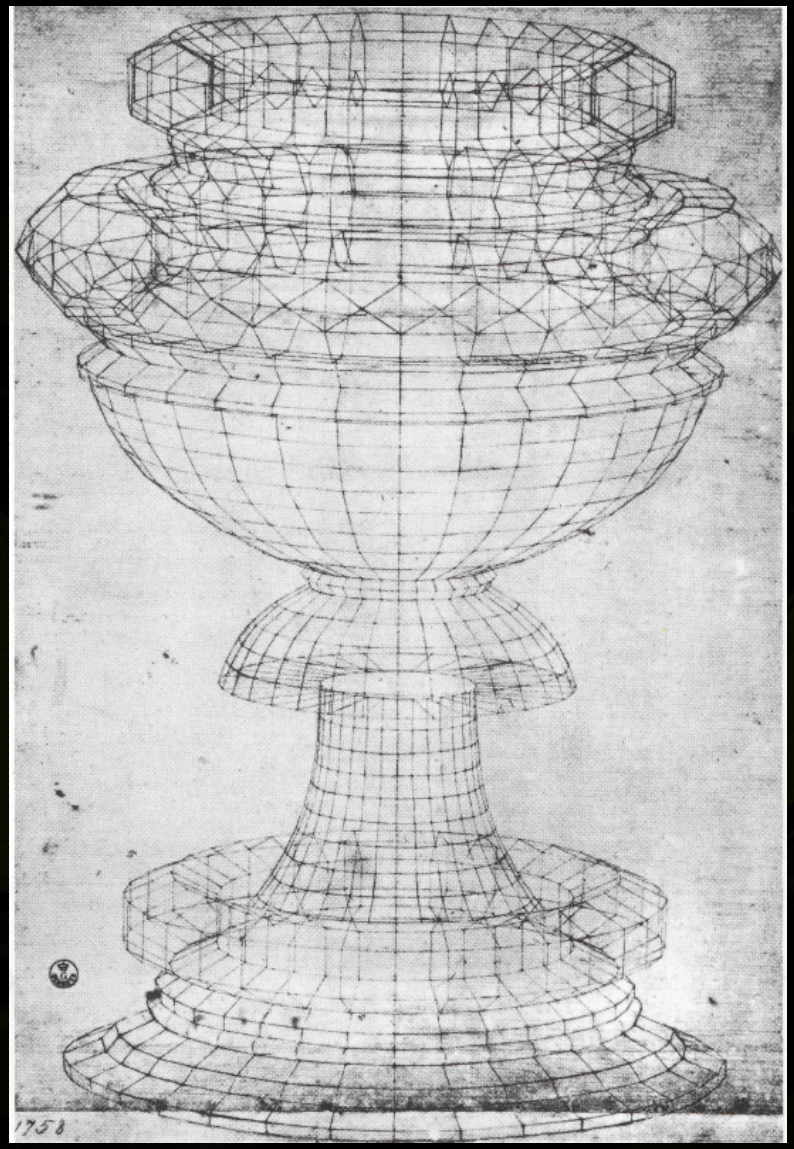

GPU Computing 3.0
The Past, Present, and
Future of GPU Computing
David Luebke
NVIDIA.

With Thanks To…

- Michael Garland, Mark Harris, John Danskin, Ignacio Castaño, David Kirk
  - and many others at NVIDIA
- Christian Eisenacher & Charles Loop, Eric Haines, Eric Sintorn & Ulf Assarsson, Anjul Patney & John Owens, DICE, CAPCOM, CryTek, Bungie
  - and many others not at NVIDIA

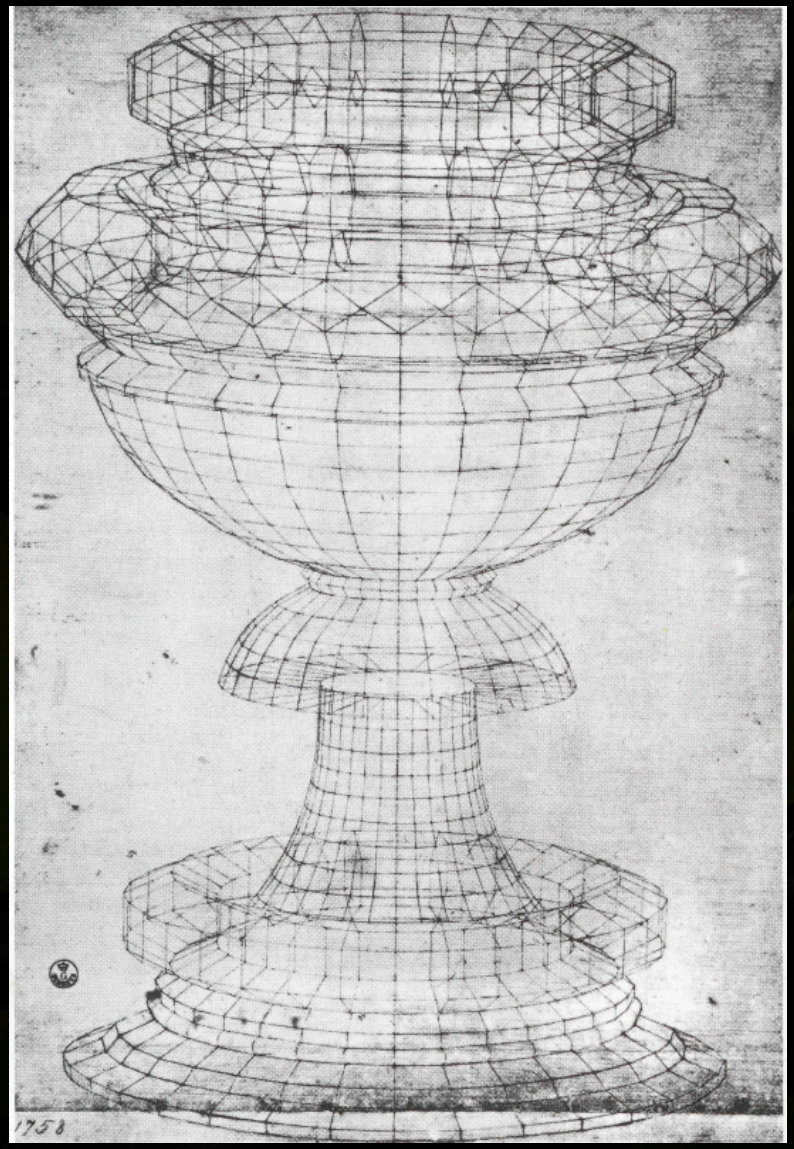

Evolution of
Graphics Hardware
NVIDIA.

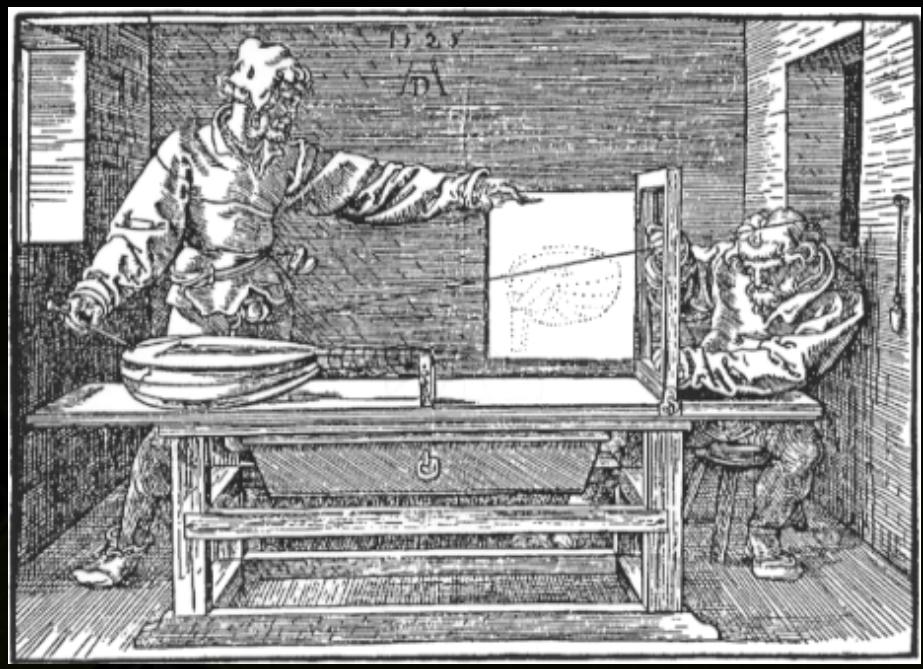
Early 3D Graphics



Perspective study of a chalice
Paolo Uccello, circa 1450

Early Graphics Hardware





Artist using a perspective machine
Albrecht Dürer, 1525

Perspective study of a chalice
Paolo Uccello, circa 1450

Early *Electronic* Graphics Hardware

SKETCHPAD: A Man-Machine Graphical Communication System
Ivan Sutherland, 1963

The Graphics Pipeline

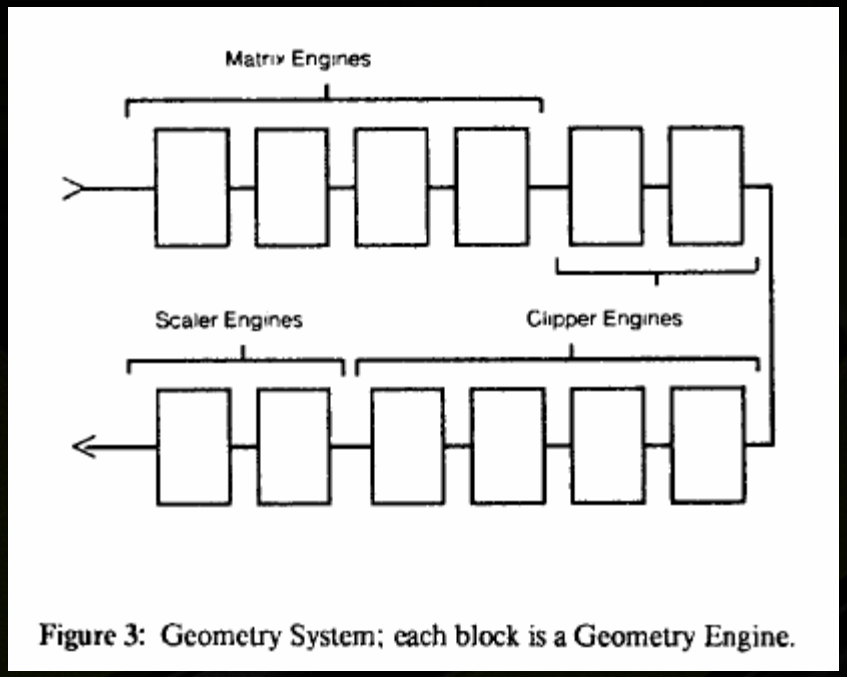


Figure 3: Geometry System; each block is a Geometry Engine.

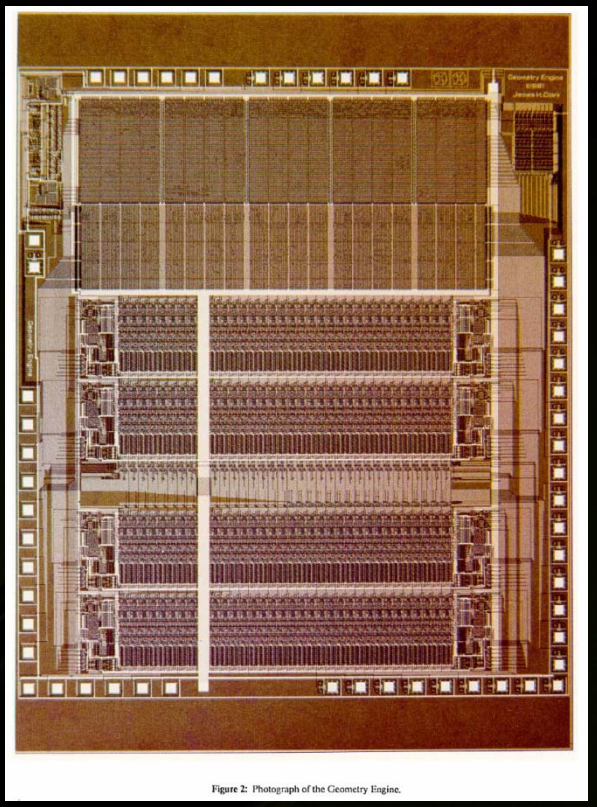
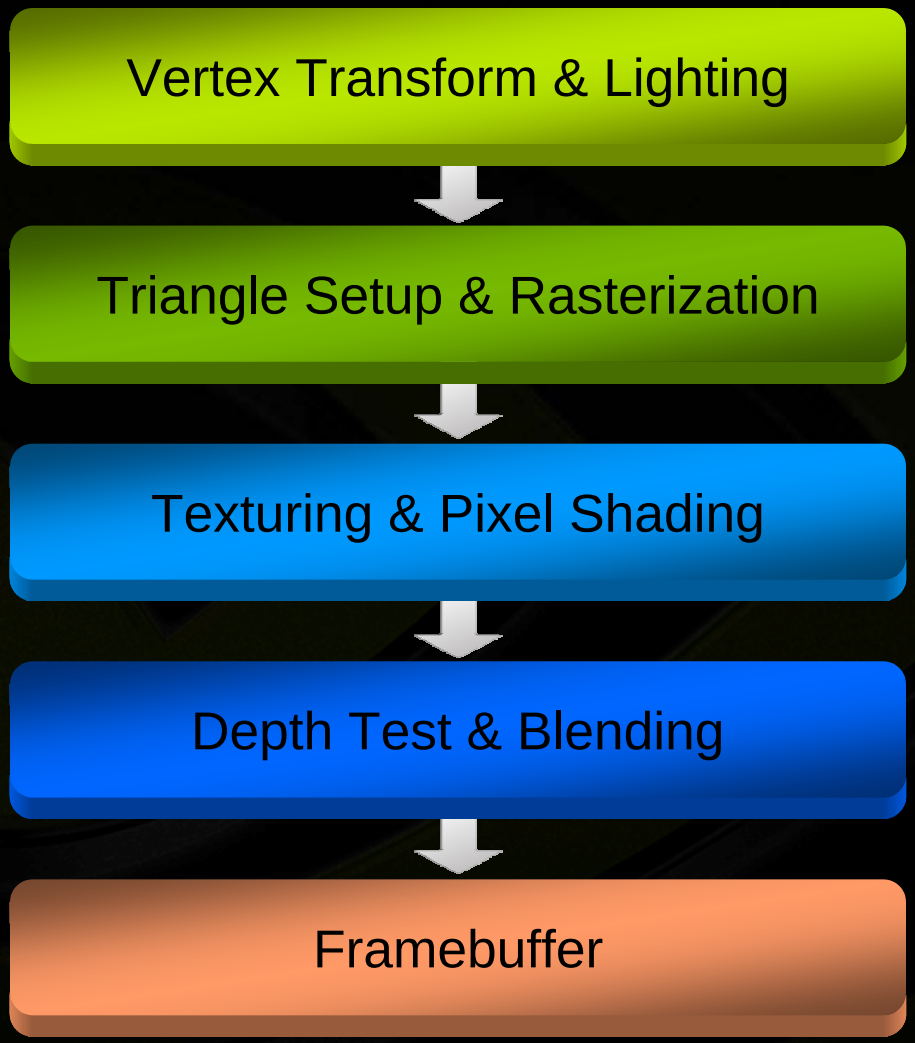


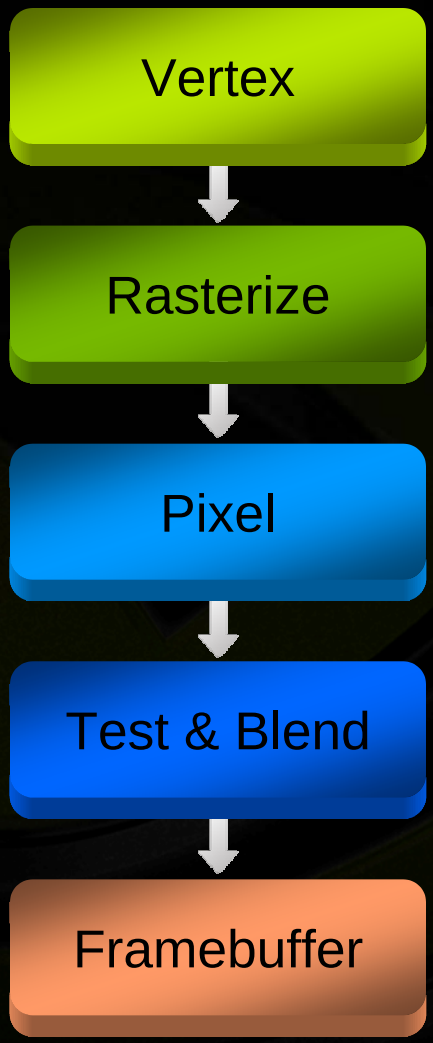
Figure 2: Photograph of the Geometry Engine.

The Geometry Engine: A VLSI Geometry System for Graphics
Jim Clark, 1982

The Graphics Pipeline

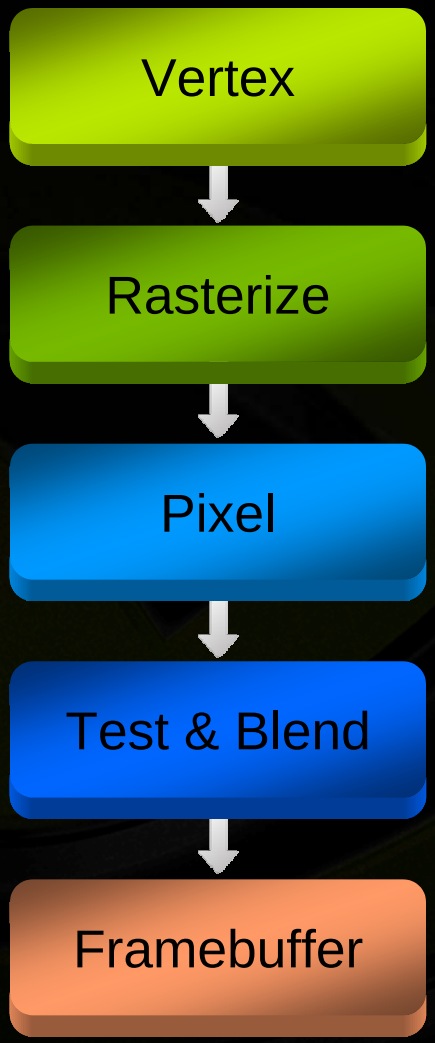


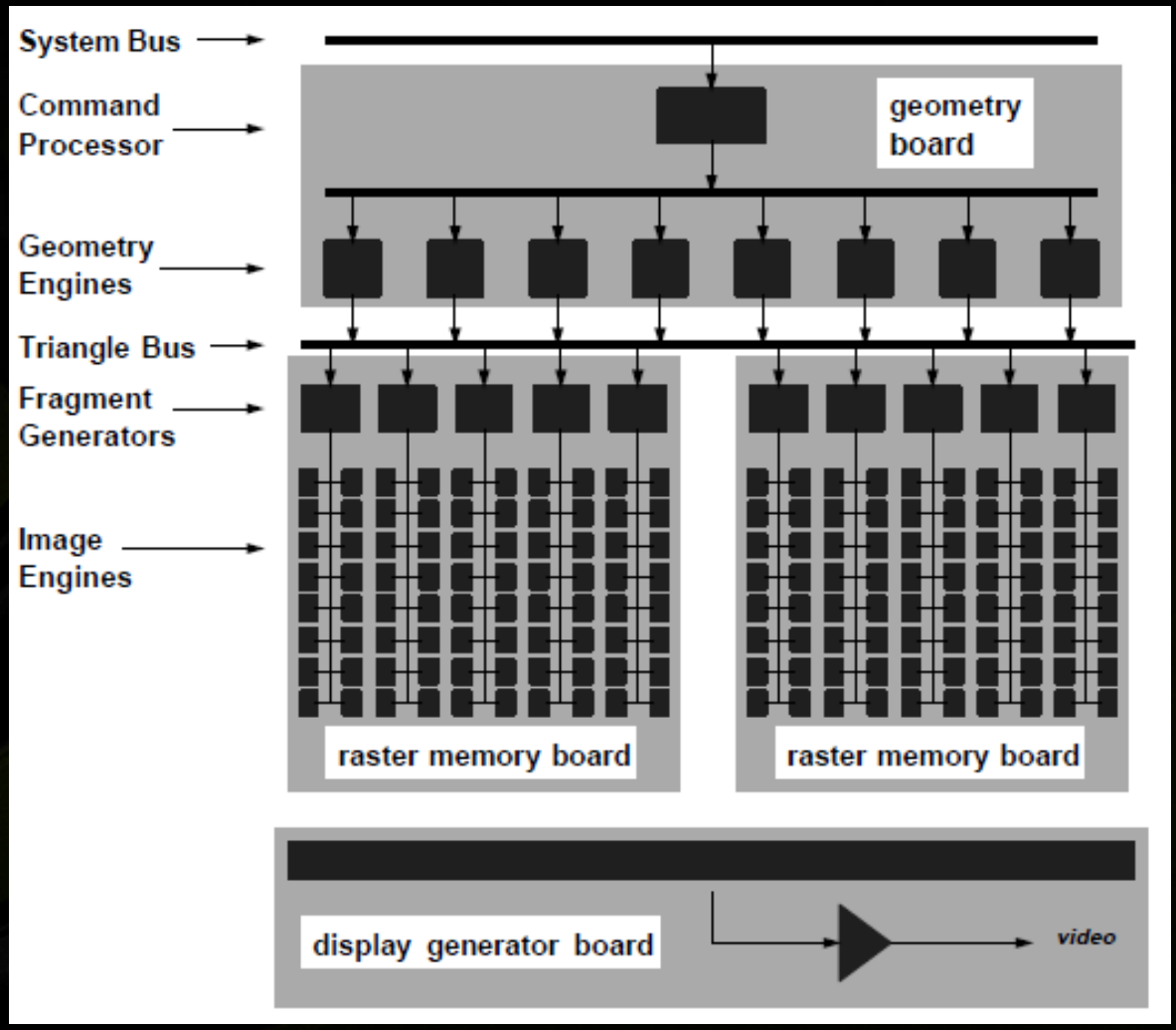
The Graphics Pipeline



- Key abstraction of real-time graphics
- Hardware used to look like this
- One chip/board per stage
- Fixed data flow through pipeline

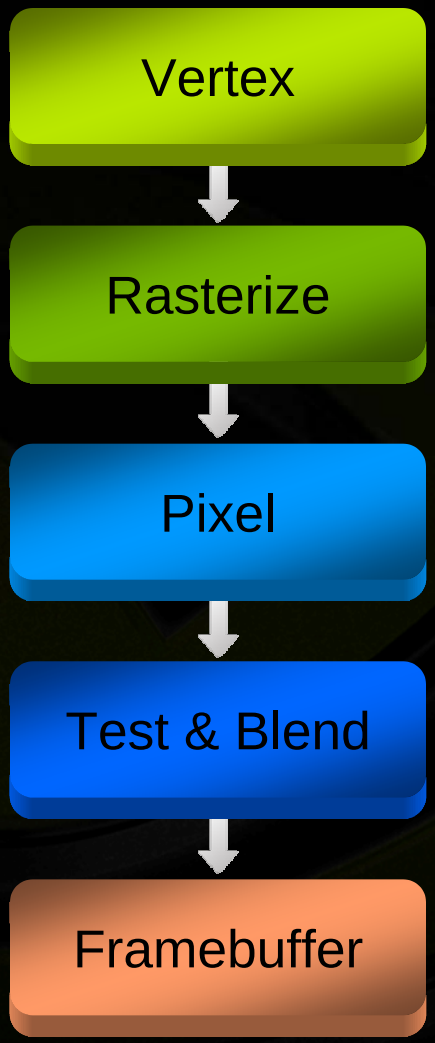
SGI RealityEngine (1993)

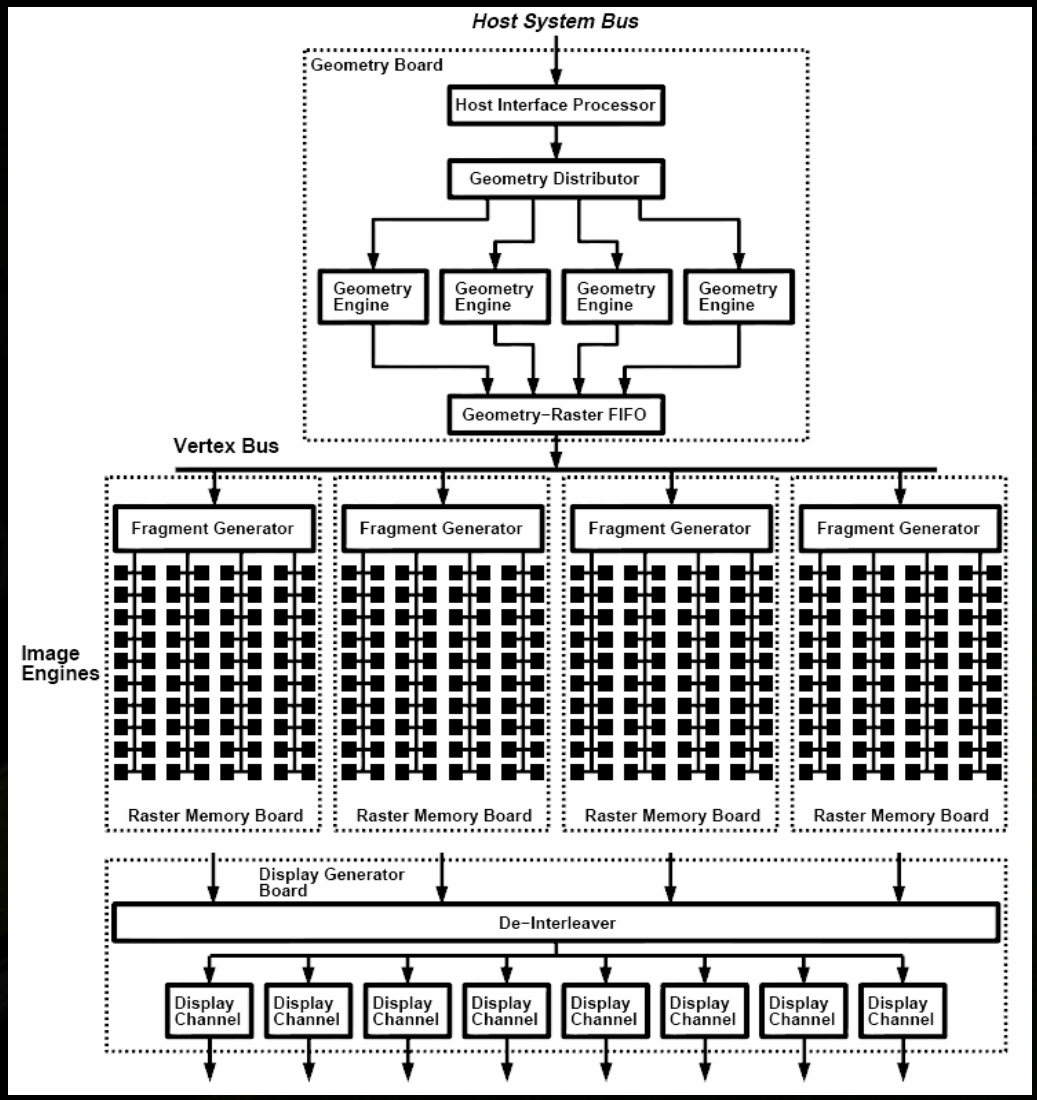




Kurt Akeley. *RealityEngine Graphics*, SIGGRAPH 93.

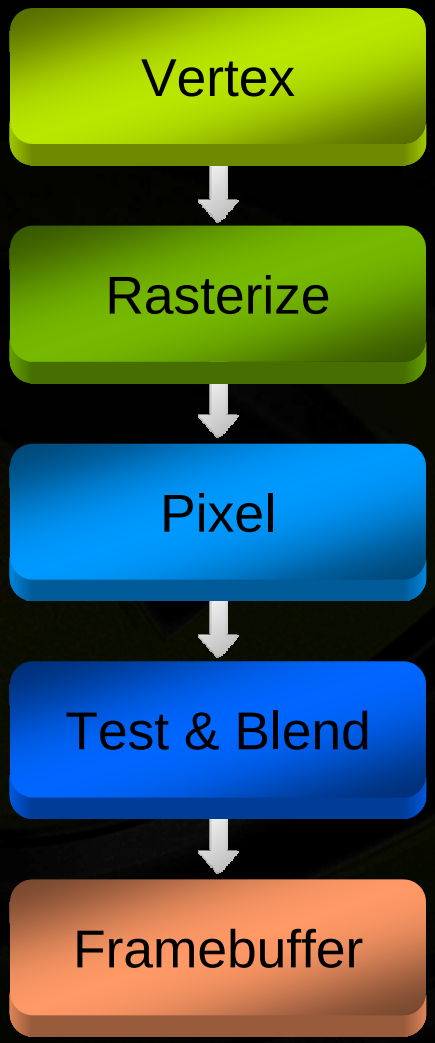
SGI InfiniteReality (1997)





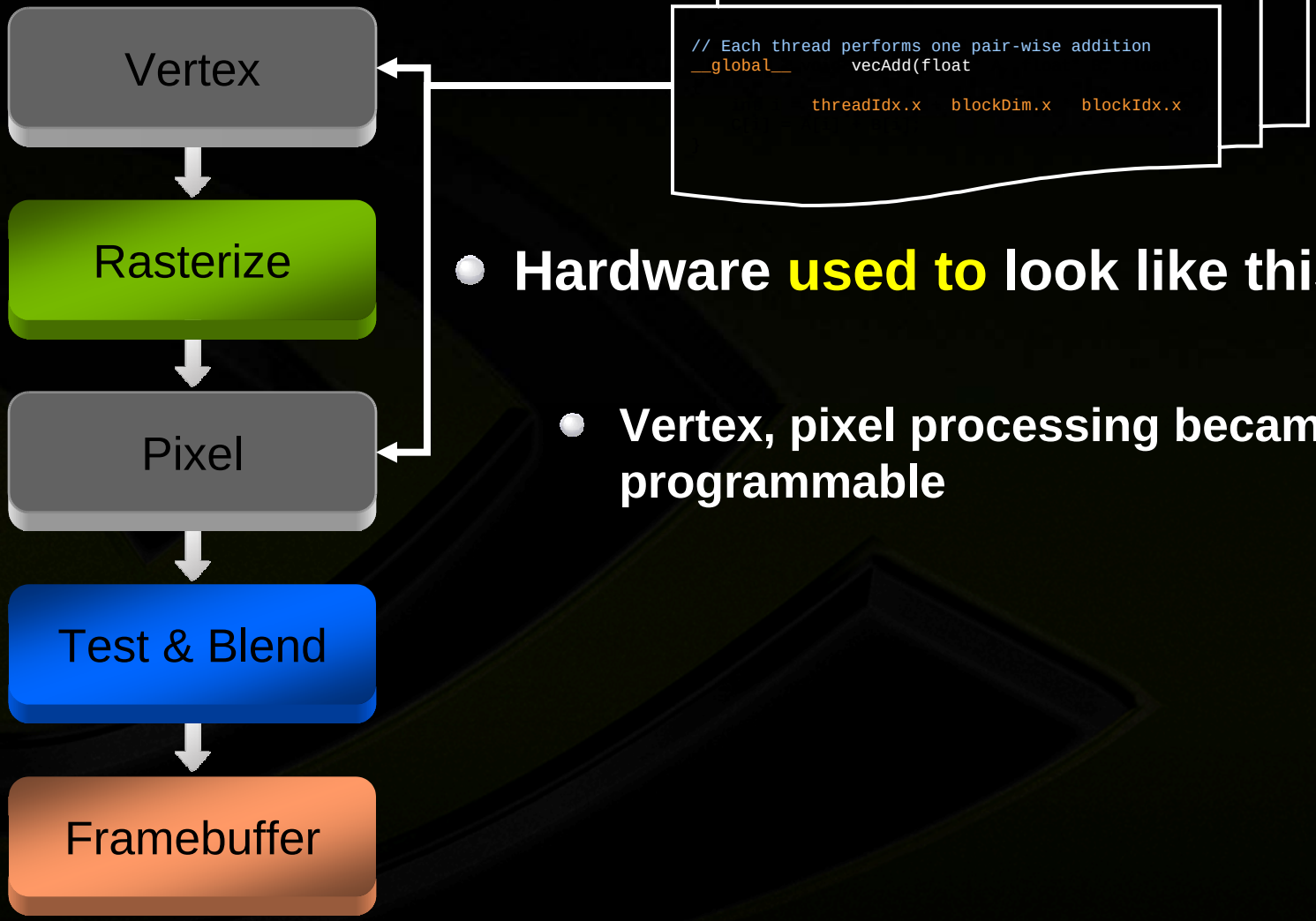
Montrym et al. *InfiniteReality: A real-time graphics system*, SIGGRAPH 97

The Graphics Pipeline



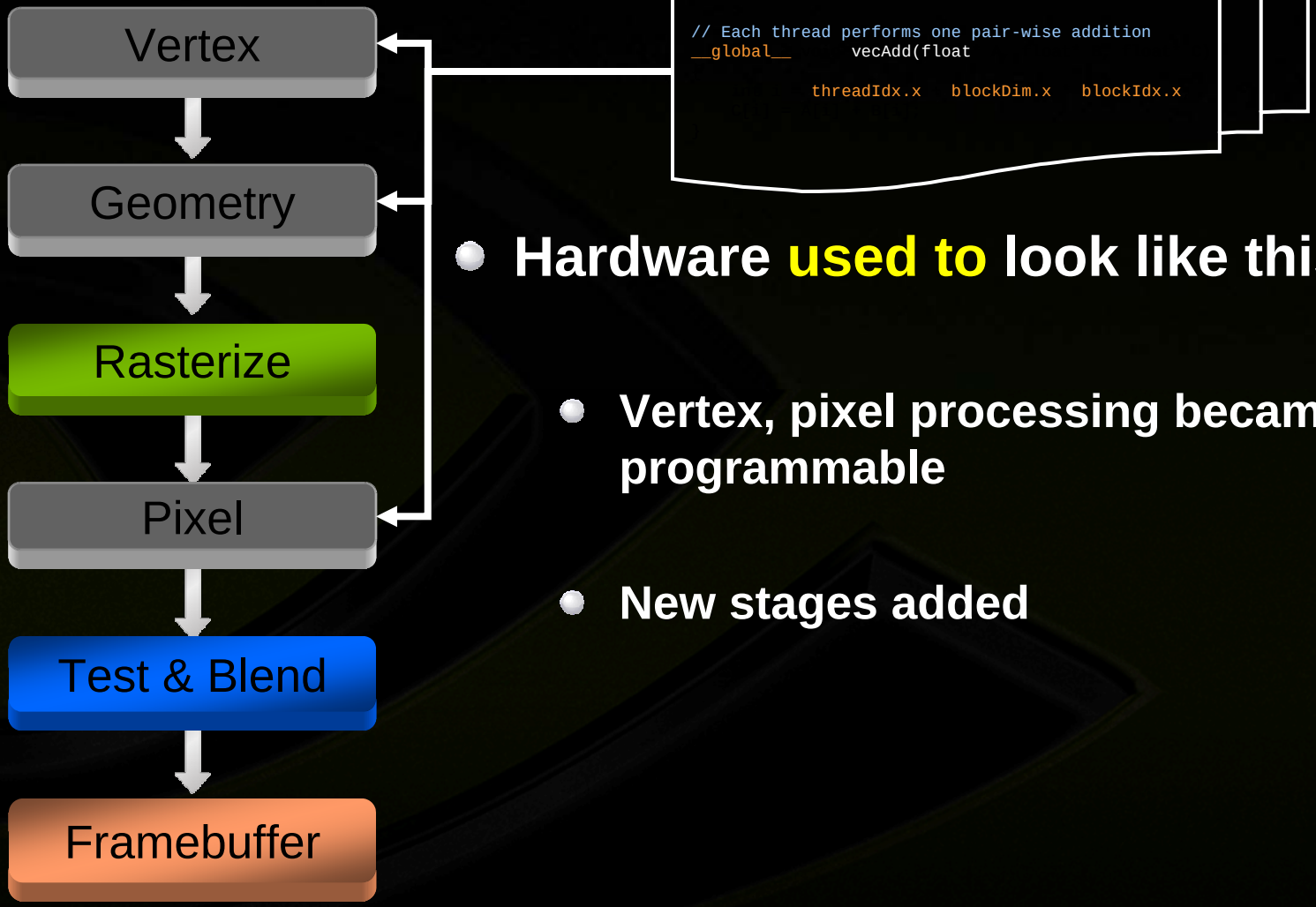
- Remains a useful abstraction
- Hardware used to look like this

The Graphics Pipeline



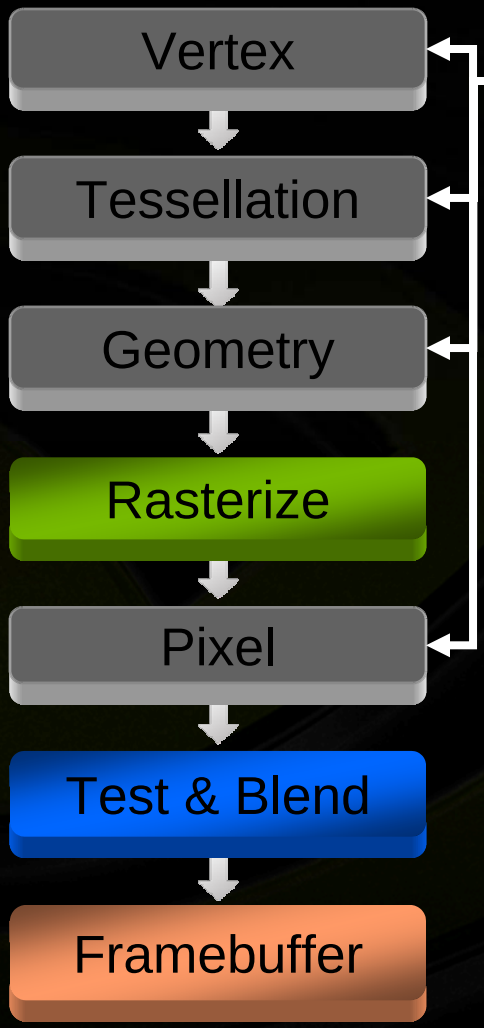
- **Hardware used to look like this:**
  - **Vertex, pixel processing became programmable**

The Graphics Pipeline



- **Hardware used to look like this**
  - **Vertex, pixel processing became programmable**
  - **New stages added**

The Graphics Pipeline

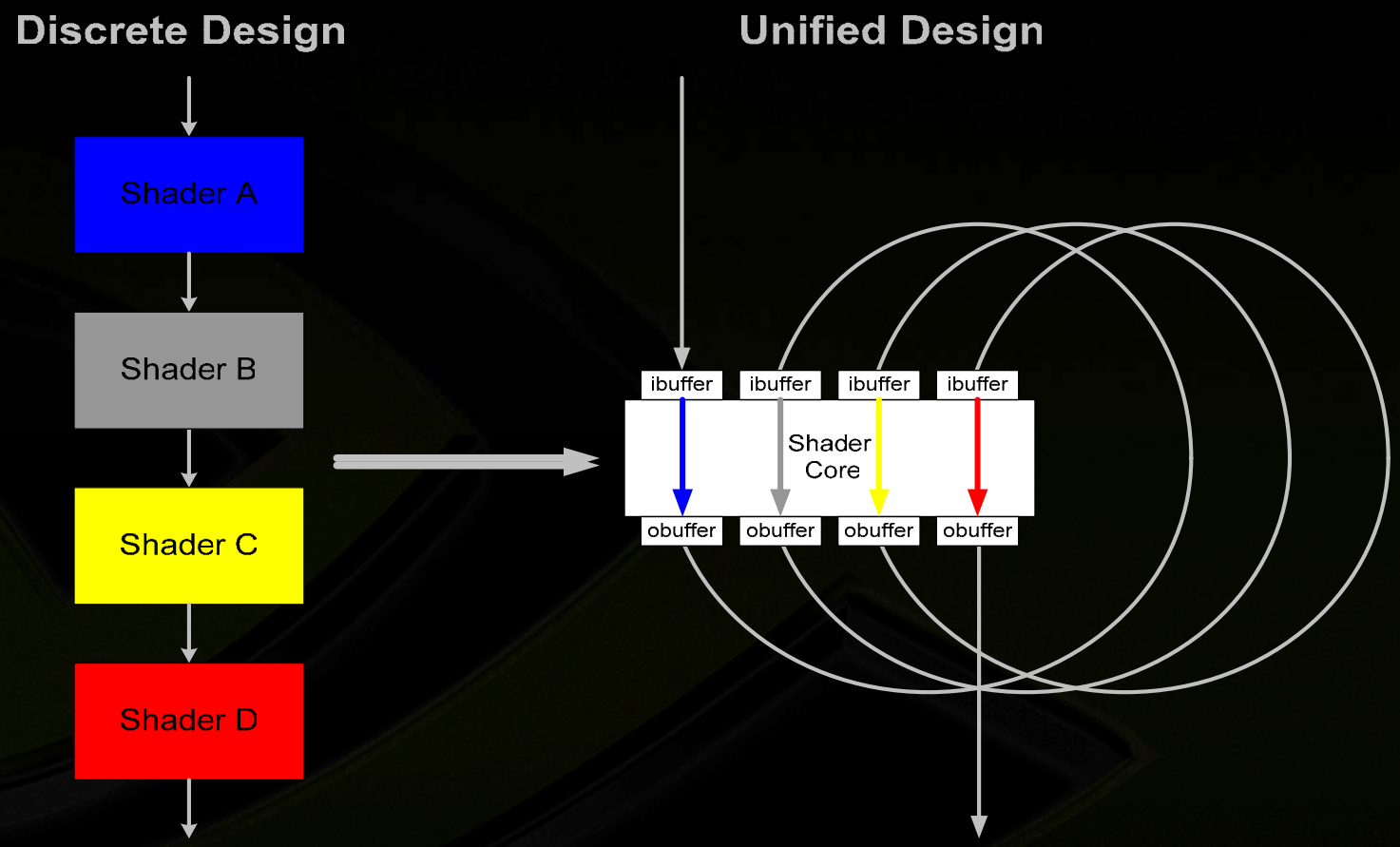



```
// Each thread performs one pair-wise addition
__global__      vecAdd(float

          threadIdx.x   blockDim.x   blockIdx.x
```

- **Hardware used to look like this**
  - **Vertex, pixel processing became programmable**
  - **New stages added**

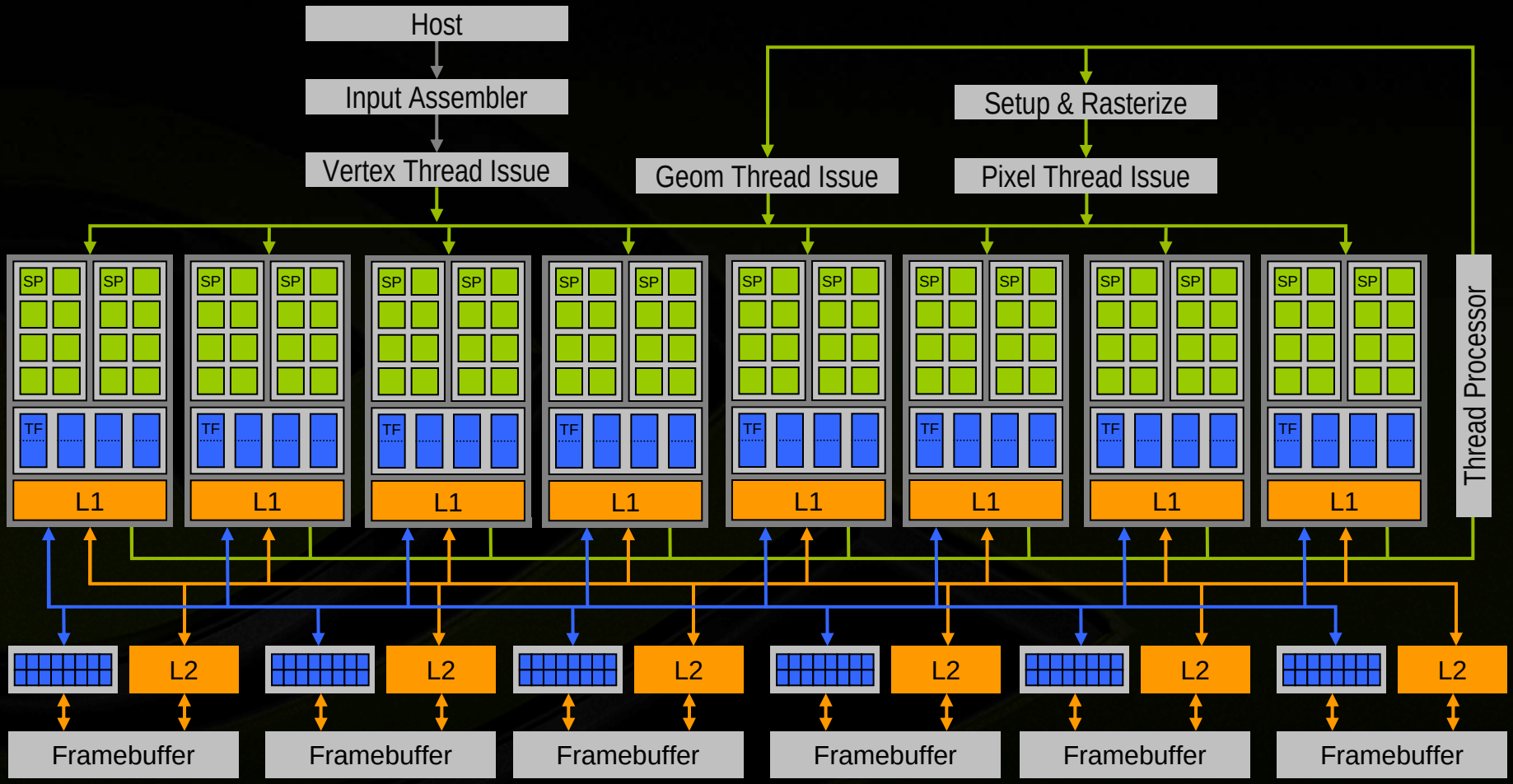
GPU architecture increasingly centers around shader execution

Modern GPUs: Unified Design

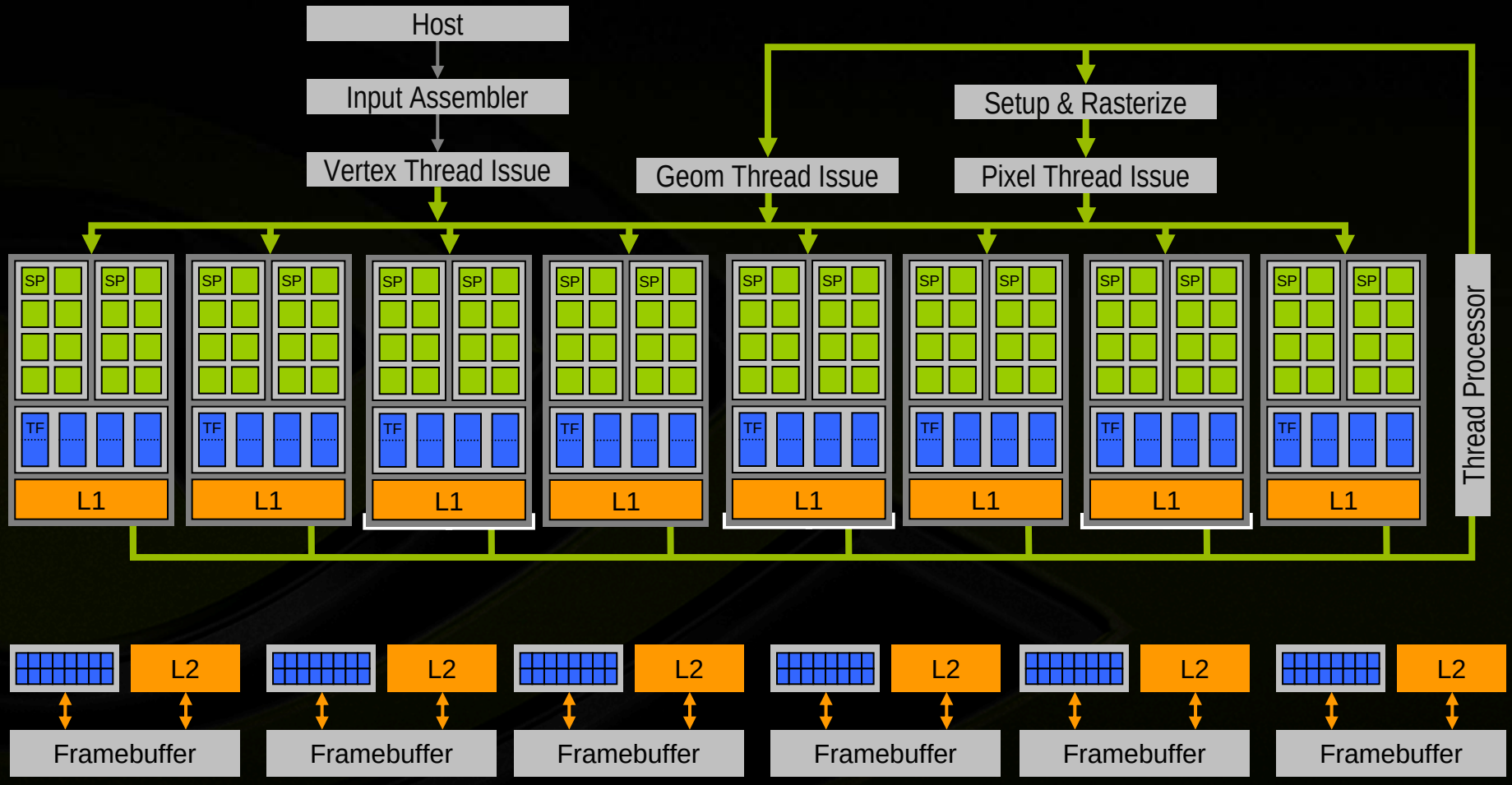


Vertex shaders, pixel shaders, etc. become *threads* running different programs on a flexible core

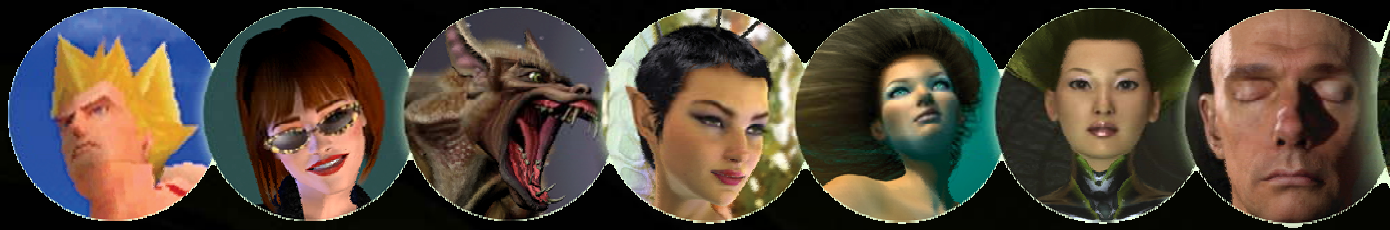
GeForce 8: Modern GPU Architecture



GeForce 8: Modern GPU Architecture



GPUs Today



1995
NV1
1 Million
Transistors

1999
GeForce 256
22 Million
Transistors

2002
GeForce4
63 Million
Transistors

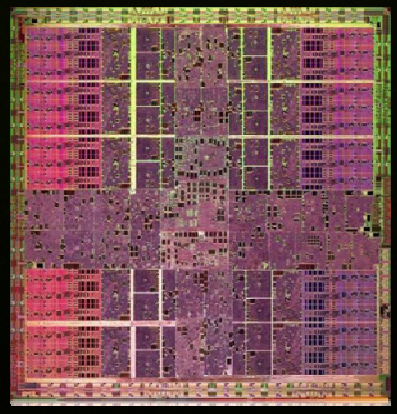
2003
GeForce FX
130 Million
Transistors

2004
GeForce 6
222 Million
Transistors

2005
GeForce 7
302 Million
Transistors

2006-2007
GeForce 8
754 Million
Transistors

2008
GeForce GTX 200
1.4 Billion
Transistors



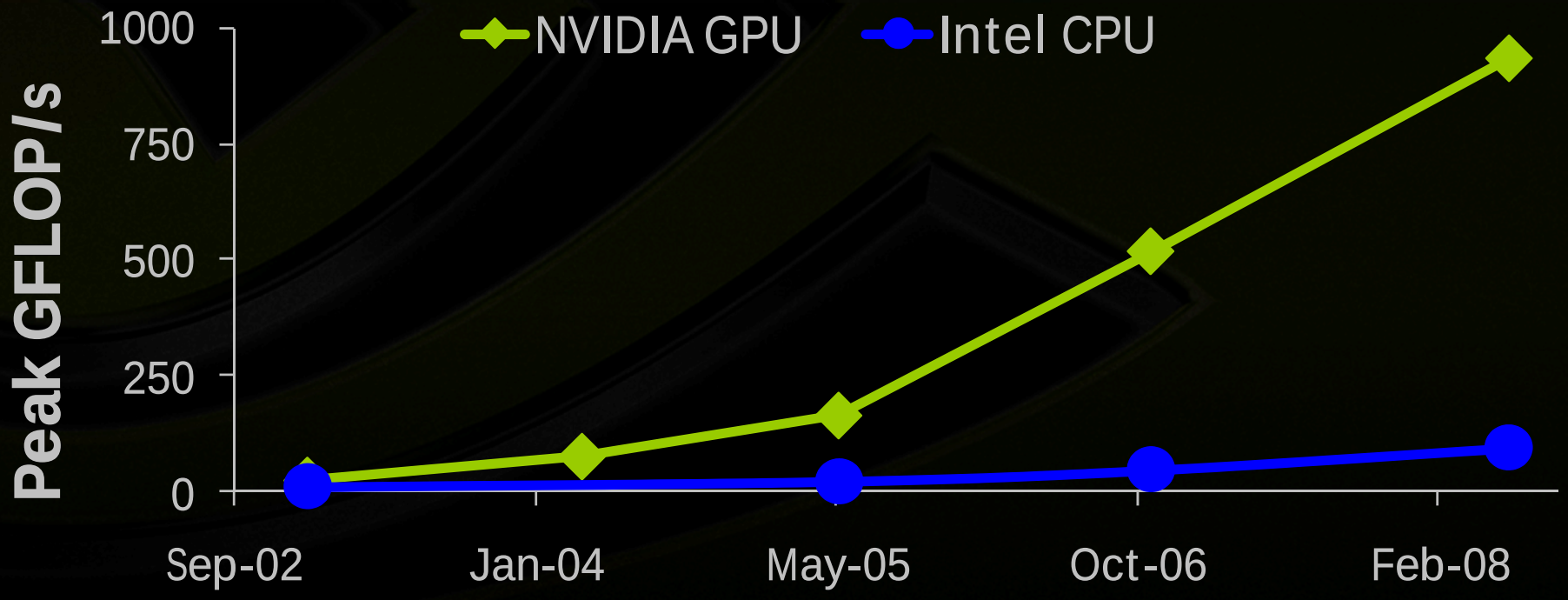
Lessons from Graphics Pipeline

- **Throughput is paramount**
- **Create, run, & retire lots of threads very rapidly**
- **Use multithreading to hide latency**

Why GPU Computing?

Fact:
nobody cares about theoretical peak

Challenge:
harness GPU power for real application performance




GPU Computing
Epochs
NVIDIA.

GPU Computing 1.0

(Ignoring prehistory: Ikonas, Pixel Machine, Pixel-Planes…)

GPU Computing 1.0: *compute pretending to be graphics*

- Disguise data as textures or geometry
- Disguise algorithm as render passes

→Trick graphics pipeline into doing your computation!

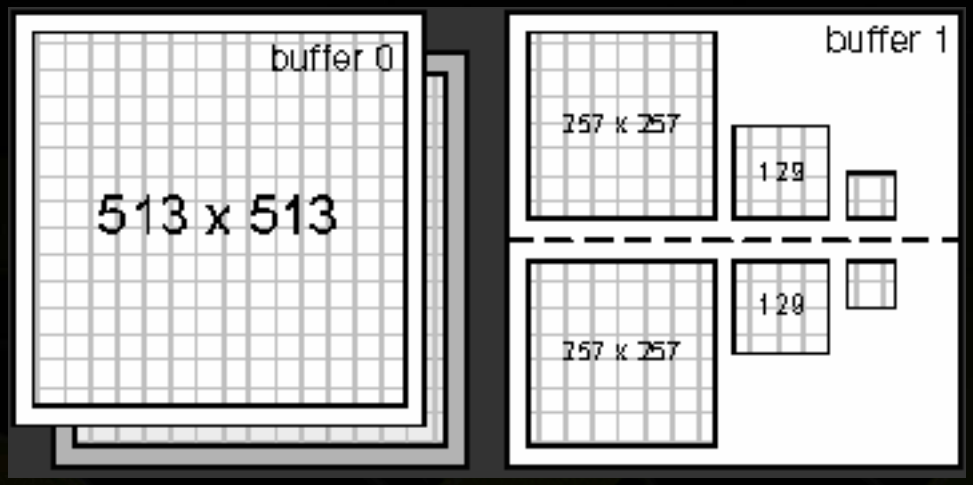
GPU Computing 1.0 – the GPGPU era

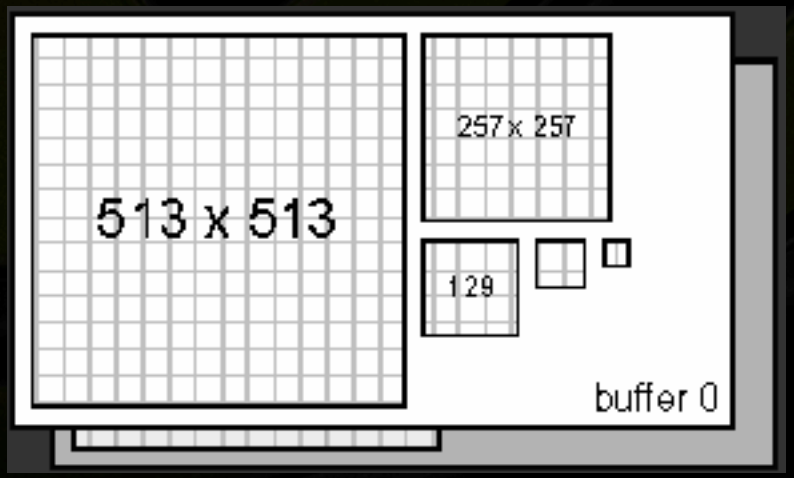
- Geometric computation
  - E.g. form factors, Voronoi diagrams, proximity calculations
  - Primarily vertex & rasterization hardware
- Domain computations
  - E.g. boiling, reaction-diffusion, fluids, image-processing
  - Primarily texture & pixel processing hardware
- Term *GPGPU* coined by Mark Harris

GPGPU Hardware & Algorithms

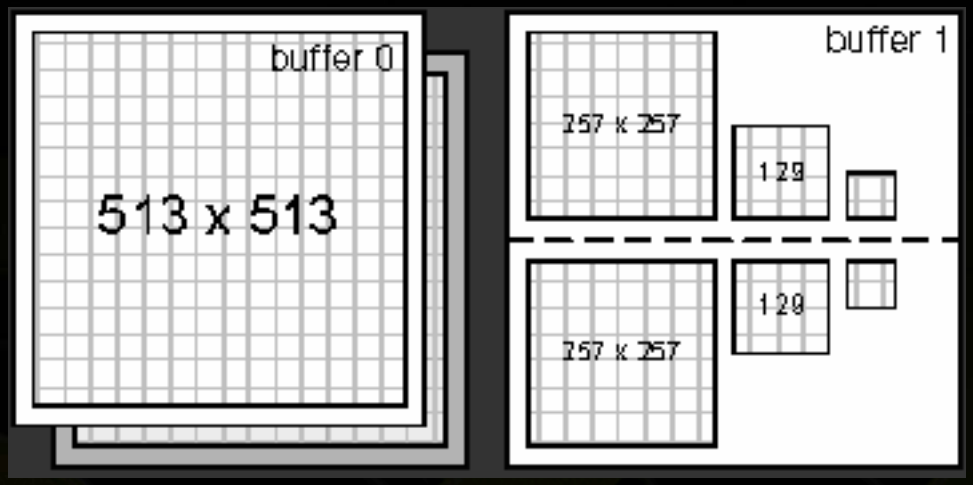
- GPUs get progressively more capable
  - Fixed-function – fancy depth + texture compositing
  - Register combiners → full programmable shading
  - Floating-point pixel hardware greatly extends reach
- Algorithms get more sophisticated
  - Cellular automata & relatives (CML, LBM)
  - PDEs & numeric solvers – multigrid, conjugate gradient, …
  - Clever graphics tricks – occlusion culling, depth/stencil, …
- High-level shading languages emerge
  - Cg, GLSL, HLSL

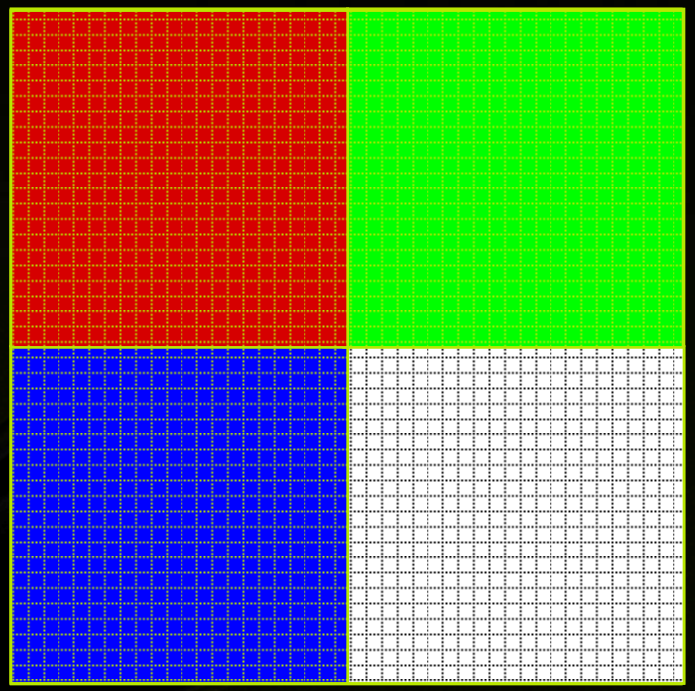
Typical GPGPU Constructs

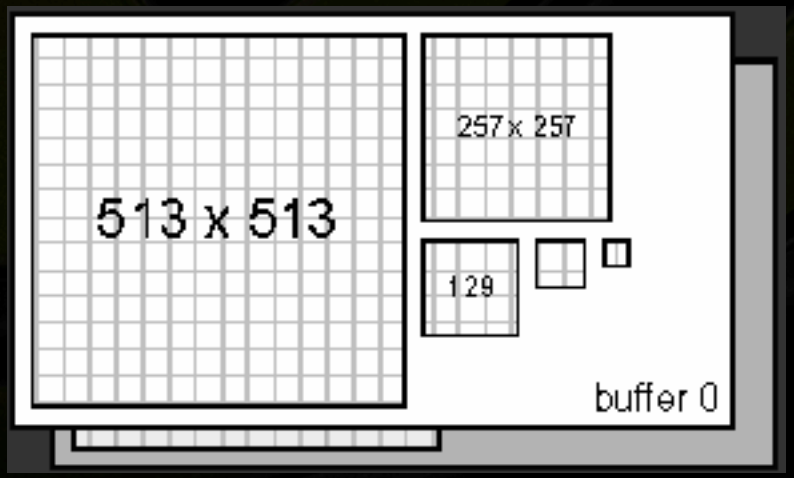




Typical GPGPU Constructs







GPGPU Evolves

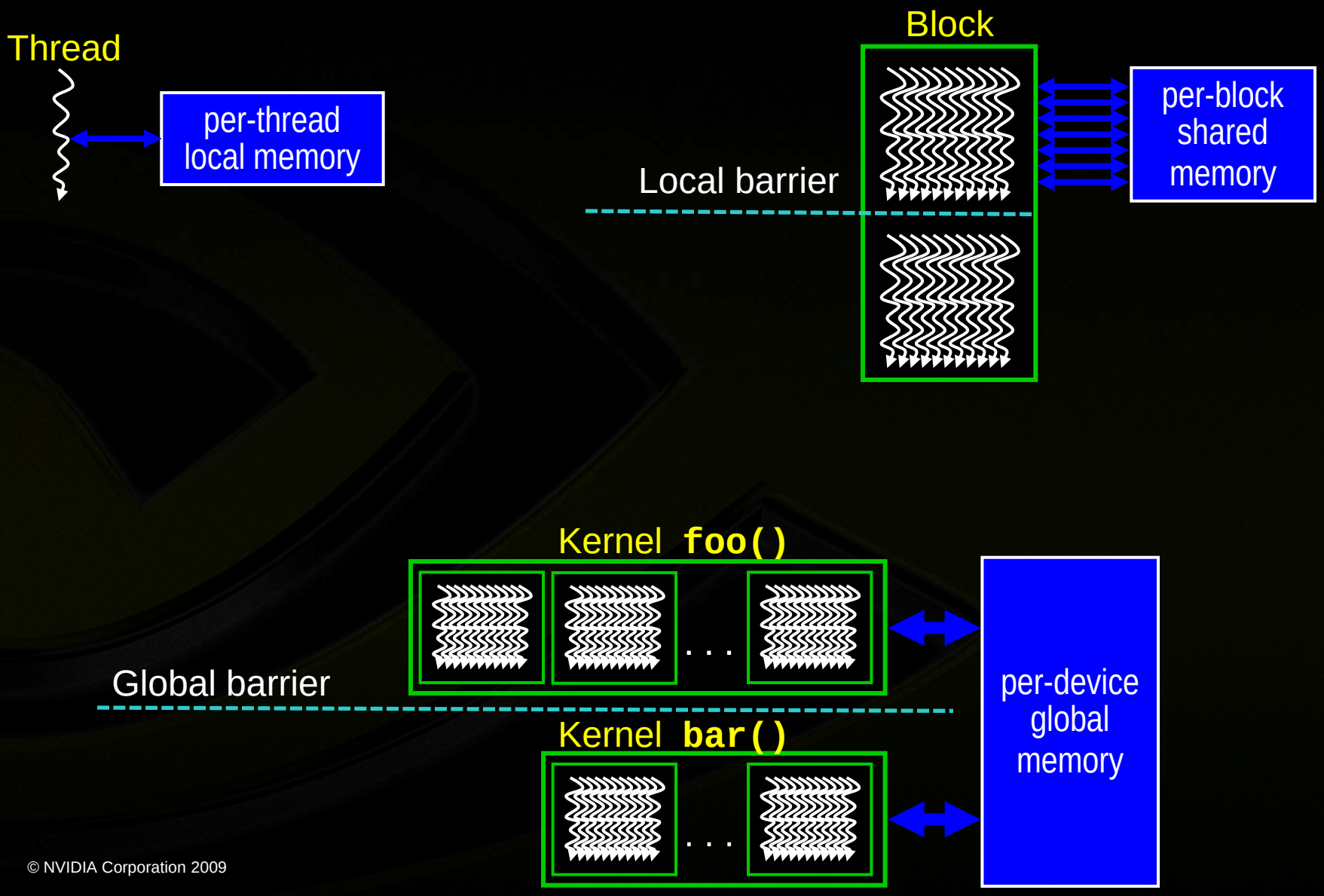
- Emergence of libraries & frameworks, e.g. Glift
- Comparison to optimized CPU codes
- Ian Buck's *BrookGPU* closes out GPGPU era
  - First attempt at complete high-level GPGPU language
  - Streams, shapes, kernels + restricted C syntax
  - Many proofs-of-concept; limited uptake outside Stanford

GPU Computing 2.0

GPU Computing 2.0: *direct compute*

- Program GPU directly, no graphics-based restrictions
  - Provide scatter, inter-thread communication, pointers
  - Enable standard languages like C
- *GPU Computing* supplants graphics-based *GPGPU*
- November 2006: NVIDIA introduces *CUDA*

CUDA In One Slide



C-for-CUDA Example


```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

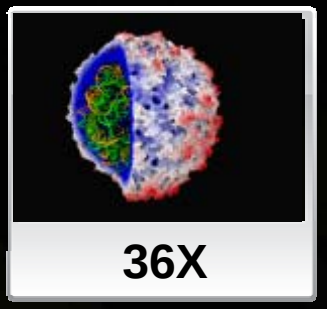
```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}

// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

CUDA Successes

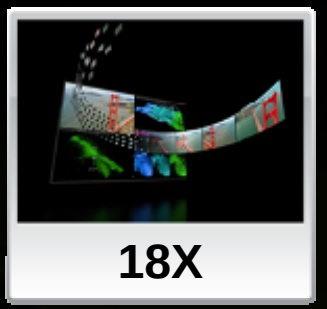
146X

Medical Imaging
U of Utah



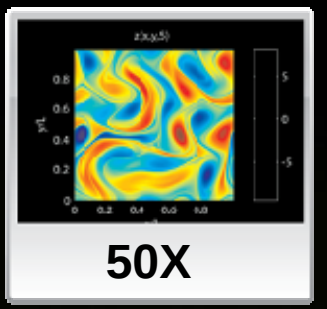
36X

Molecular Dynamics
U of Illinois



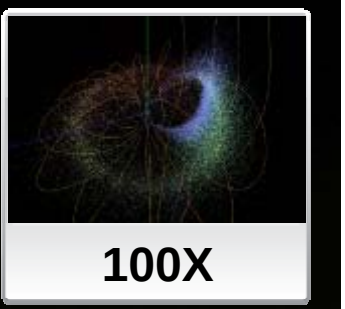
18X

Video Transcoding
Elemental Tech



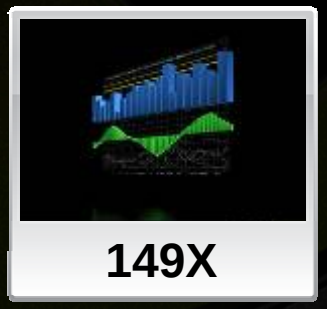
50X

Matlab Computing
AccelerEyes



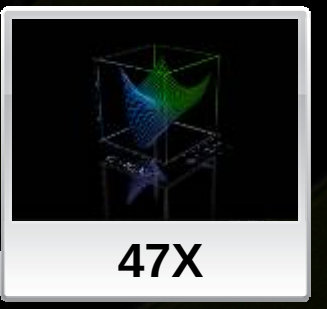
100X

Astrophysics
RIKEN



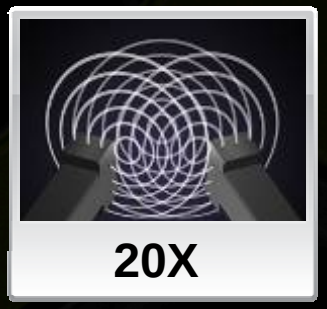
149X

Financial simulation
Oxford



47X

Linear Algebra
Universidad Jaime

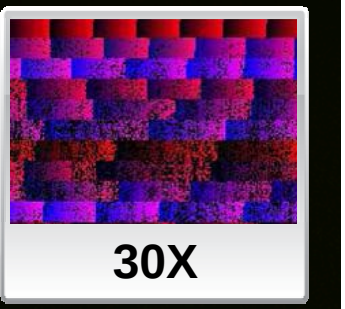


20X

3D Ultrasound
Techniscan

130X

Quantum Chemistry
U of Illinois



30X

Gene Sequencing
U of Maryland

Now Entering: GPU Computing 3.0

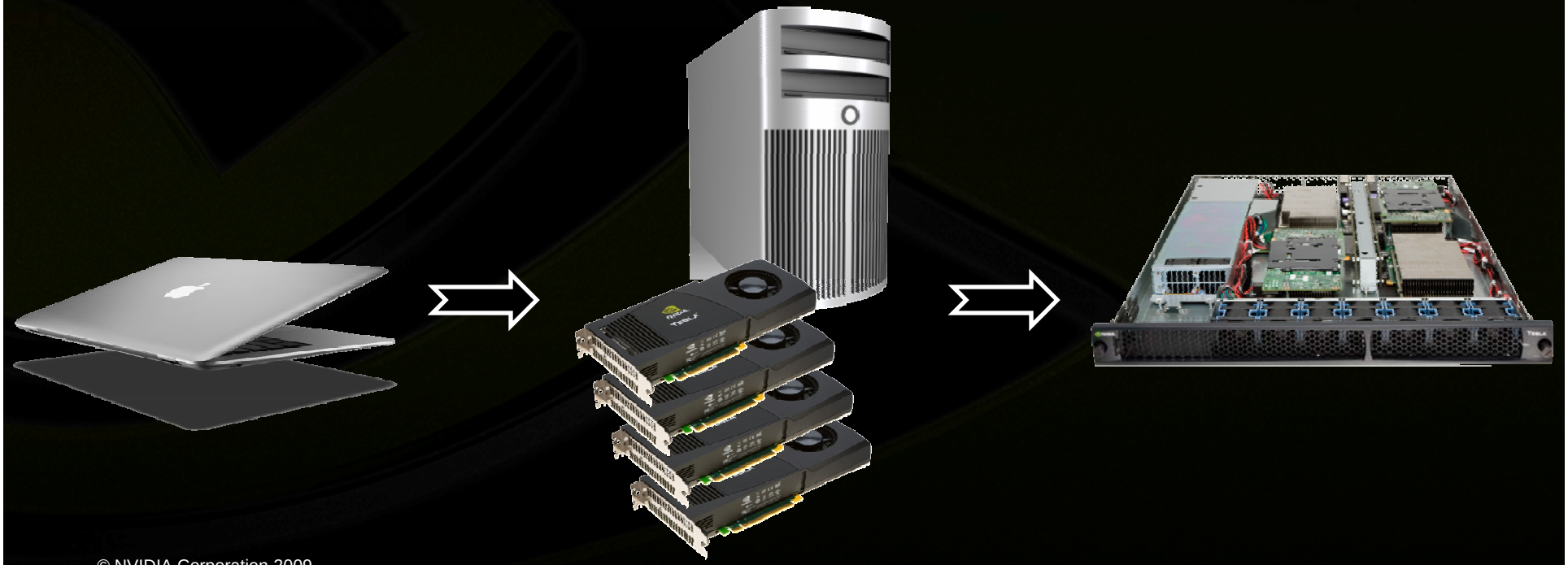
GPU Computing 3.0: ***an emerging ecosystem***

- **Hardware & product lines**
- **Algorithmic sophistication**
- **Cross-platform standards**
- **Education & research**
- **Consumer applications**
- **High-level languages**

GPU begins to "vanish" behind codes, platforms, apps

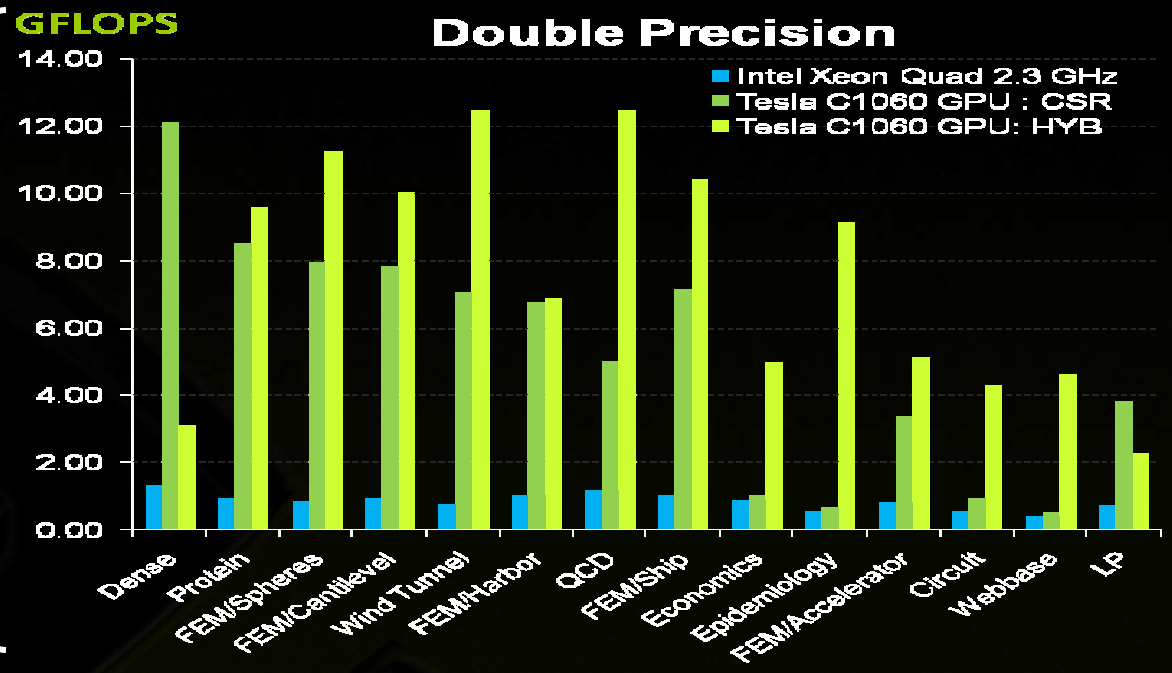
Hardware & Products

- GT200: double precision, coalescing, shmem atomics
- Product line from laptops to supercomputers



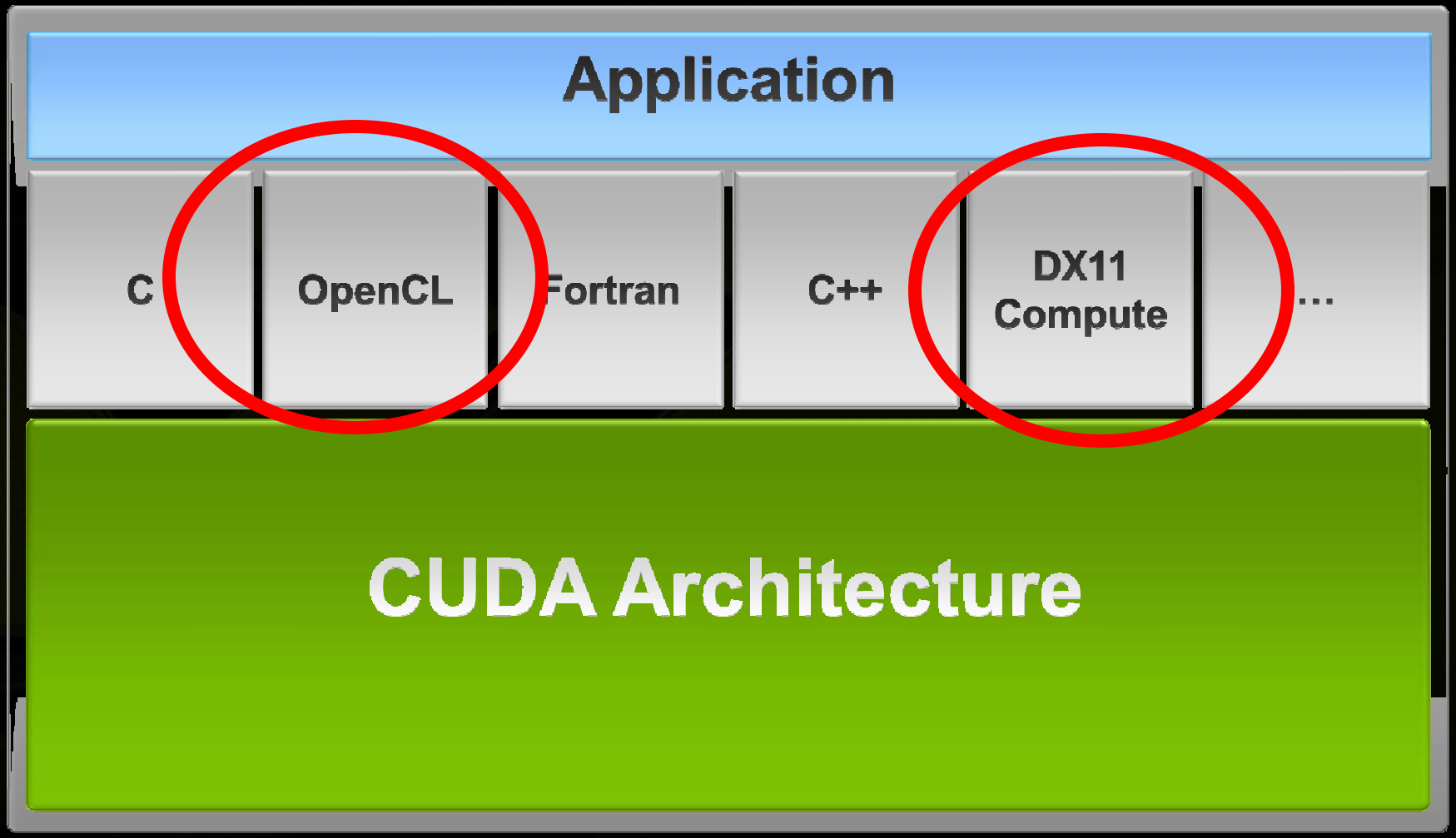
Algorithmic Sophistication

- Sort
- Sparse matrix
- Hash tables
- Fast multipole method
- Ray tracing (parallel tree traversal)

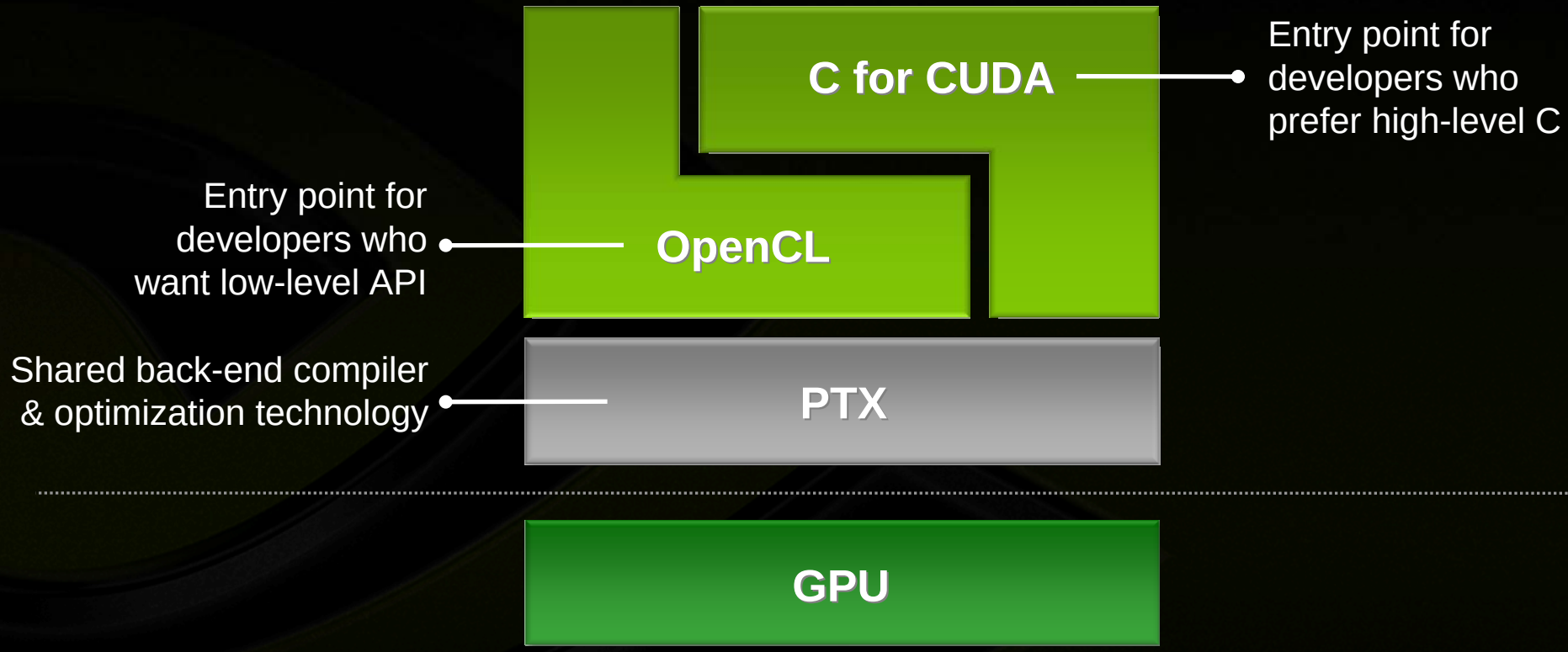


CPU Results from "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms", Williams et al, Supercomputing 2007

Cross-Platform Standards



OpenCL vs. C for CUDA



Different Host Code Styles

Calling a C function in nvcc

```
extern "C" void FFT1024( float2 *work, int batch )
{
    FFT1024_device<<< grid2D(batch), 64 >>>
      ( work, work );
}
```

nvcc compiler, CUDA runtime, PTX layer manage kernel resources and execution

OpenCL API-style programming

```
// create a compute context with GPU device
context = clCreateContextFromType(CL_DEVICE_TYPE_GPU);
// create a work-queue
queue = clCreateWorkQueue(context, NULL, NULL, 0);
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
sizeof(float)*2*num_entries, srcA);
memobjs[1] = clCreateBuffer(context,
CL_MEM_READ_WRITE,
sizeof(float)*2*num_entries, NULL);
// create the compute program
program = clCreateProgramFromSource(context, 1,
&fft1D_1024_kernel_src, NULL);
// build the compute program executable
clBuildProgramExecutable(program, false, NULL, NULL);
// create the compute kernel
kernel = clCreateKernel(program, “fft1D_1024”);
// create N-D range object with work-item dimensions
global_work_size[0] = n;
local_work_size[0] = 64;
range = clCreateNDRangeContainer(context, 0, 1,
global_work_size,
local_work_size);
// set the args values
clSetKernelArg(kernel, 0, (void *)&memobjs[0],
sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 1, (void *)&memobjs[1],
sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 2, NULL,
sizeof(float)*(local_work_size[0]+1)*16, NULL);
clSetKernelArg(kernel, 3, NULL,
sizeof(float)*(local_work_size[0]+1)*16, NULL);
// execute kernel
clExecuteKernel(queue, kernel, NULL, range, NULL, 0, NULL);
```

Source:
SIGGraph sneak preview
A Munshi, Apple Computer

FFT Kernel Example

C for CUDA (Written by Vasily Volkov, © UC

```
__global__ void FFT1024_device( float2 *dst, float2 *src )
{
  int tid = threadIdx.x; int iblock = blockIdx.y * gridDim.x + blockIdx.x;
  int index = iblock * 1024 + tid; src += index; dst += index;
  int hi4 = tid>>4; int lo4 = tid&15;int hi2 = tid>>4; int mi2 = (tid>>2)&3;int
      lo2 = tid&3;
  float2 a[16];
  __shared__ float smem[69*16];

  load<16>( a, src, 64 );
  FFT16( a );
  twiddle<16>( a, tid, 1024 );
  int il[] = {0,1,2,3, 16,17,18,19, 32,33,34,35, 48,49,50,51};
  transpose<16>( a, &smem[lo4*65+hi4], 4, &smem[lo4*65+hi4*4], il );
  FFT4x4( a );
  twiddle4x4( a, lo4 );
  transpose4x4( a, &smem[hi2*17 + mi2*4 + lo2], 69, &smem[mi2*69*4 +
      hi2*69 + lo2*17 ], 1, 0xE );
  FFT16( a );
  store<16>( a, dst, 64 );
}
```

Calculate Index

Load Data

FFT Kernel

OPENCL

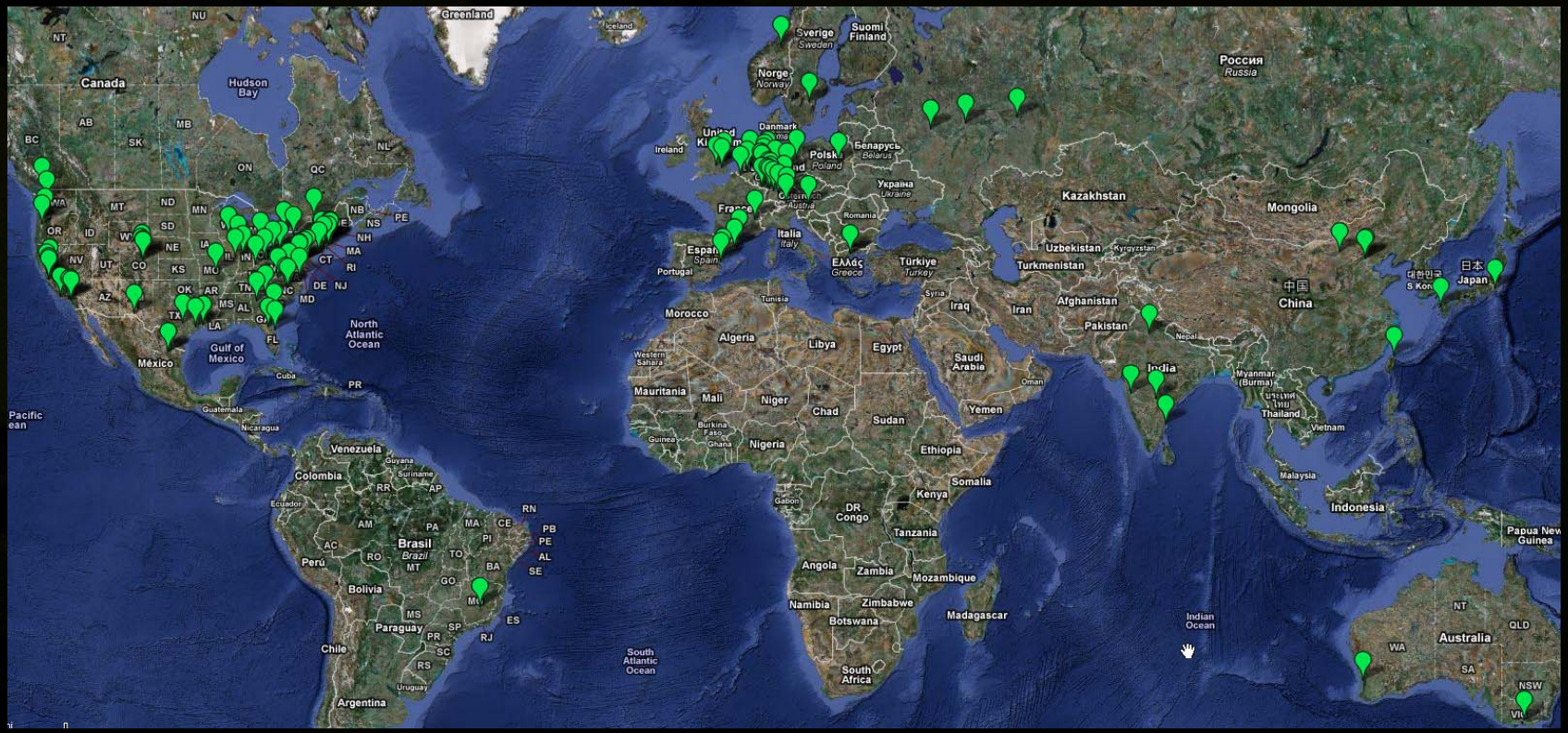
```
__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                          __local float *sMemx, __local float *sMemy)
{
  int tid = get_local_id(0);   int blockIdx = get_group_id(0) * 1024 + tid;
  float2 data[16];
  in = in + blockIdx;  out = out + blockIdx;

  globalLoads(data, in, 64); // coalesced global reads
  fftRadix16Pass(data);      // in-place radix-16 pass
  twiddleFactorMul(data, tid, 1024, 0);
  localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));
  fftRadix16Pass(data);              // in-place radix-16 pass
  twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication
  localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));
  fftRadix4Pass(data);
  fftRadix4Pass(data + 4); // four radix-4 function calls
  fftRadix4Pass(data + 8)
  fftRadix4Pass(data + 12);
  globalStores(data, out, 64); // coalesced global writes
 }
```

Languages & Platforms

- Fortran:
  - PGI Fortran to CUDA compiler (alpha release)
  - Fortran wrappers for CUDA, e.g. FLAGON (UMD)
  - Full Fortran support late this year
- MATLAB
  - JACKET by Accelereyes
- Also Available
  - PyCUDA : Python Wrapper for CUDA
  - Java wrappers, .NET integration
  - Haskell implementation

Education & Research

- 900+ research papers
- 60,000+ active developers
- 115+ universities teaching CUDA



GPU Begins to Vanish

- Ever-increasing number of codes & commercial packages “just go faster” when GPU is present
- Ex: bio/chem codes available & porting:
  - NAMD / VMD, GROMACS (alpha), HOOMD, GPU HMMER, MUMmerGPU, AutoDock…
  - LAMMPS, CHARMM, Q-Chem, Gaussian, AMBER…
- Consumer applications:
  - Photoshop, Badaboom, vReveal, Nero MoveIt, Folding@Home, …

Business Realities

Why are GPUs so cheap?

- A true commodity business
- Massive economies of scale
  - ~1,000,000 GPUs sold per week
  - ~100 per minute

Quiz

- G80 was the world's first GPU to support DirectX 10

Q: *What was the most important feature of G80?*

A: World's fastest DirectX 9 GPU

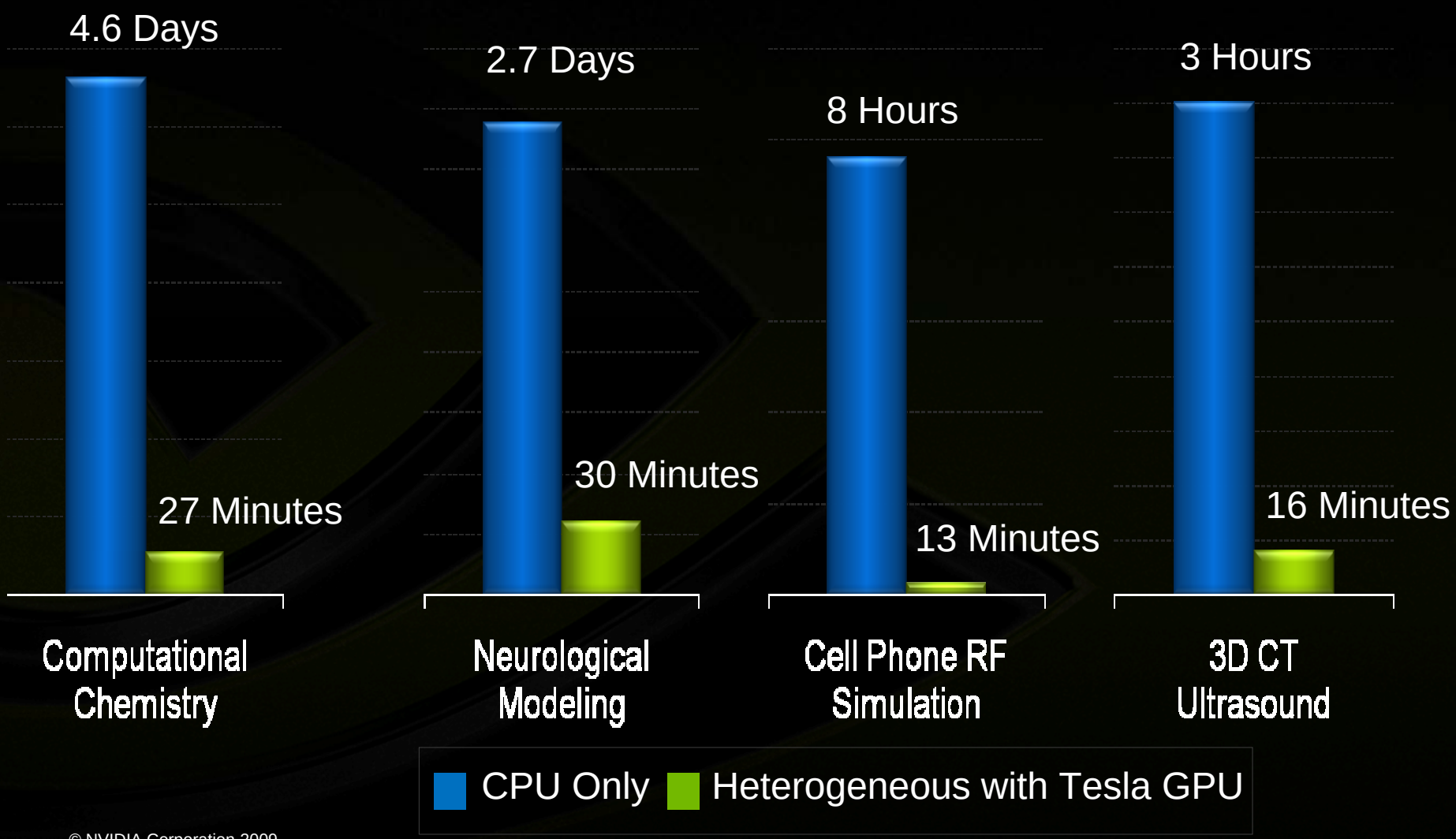
Workloads

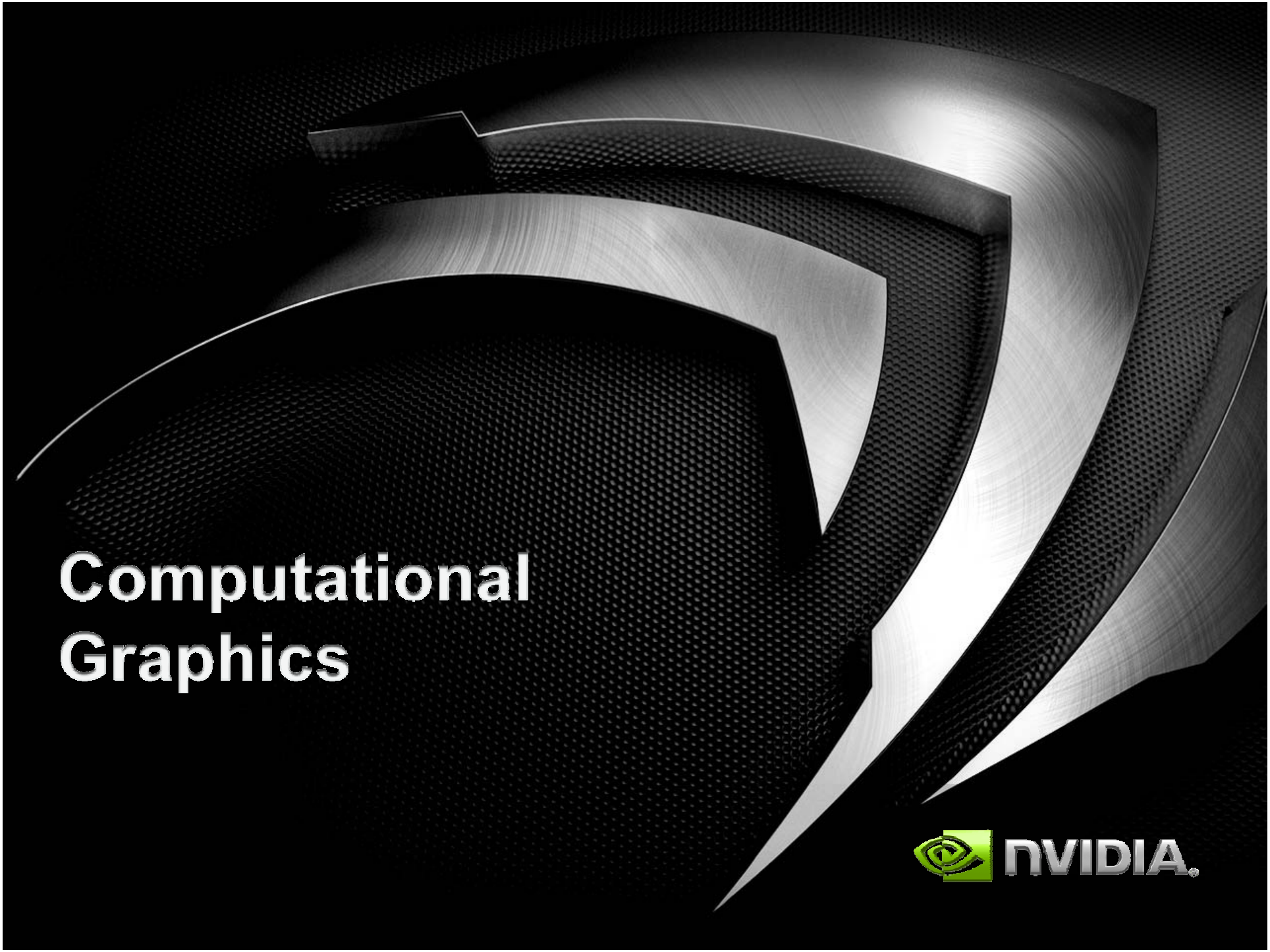
- Each GPU is designed to target a mix of known and speculative workloads
- The art of GPU design is choosing these workloads
  (and shipping on schedule!)

What workloads will drive future GPUs?

- DXn-1 performance
- High performance computing
- *Computational graphics*

HPC: Accelerating Time to Insight




Computational
Graphics
NVIDIA.

Histogram

- Distribution of colors in an image
- Image analysis for HDR tone mapping

Reinhard HDR Tonemapping operator

HDR in Valve's source engine

Separable Filters

- Depth of field
- Bloom filter

Halo 3 © Bungie Studios

Crysis © Crytek GmbH

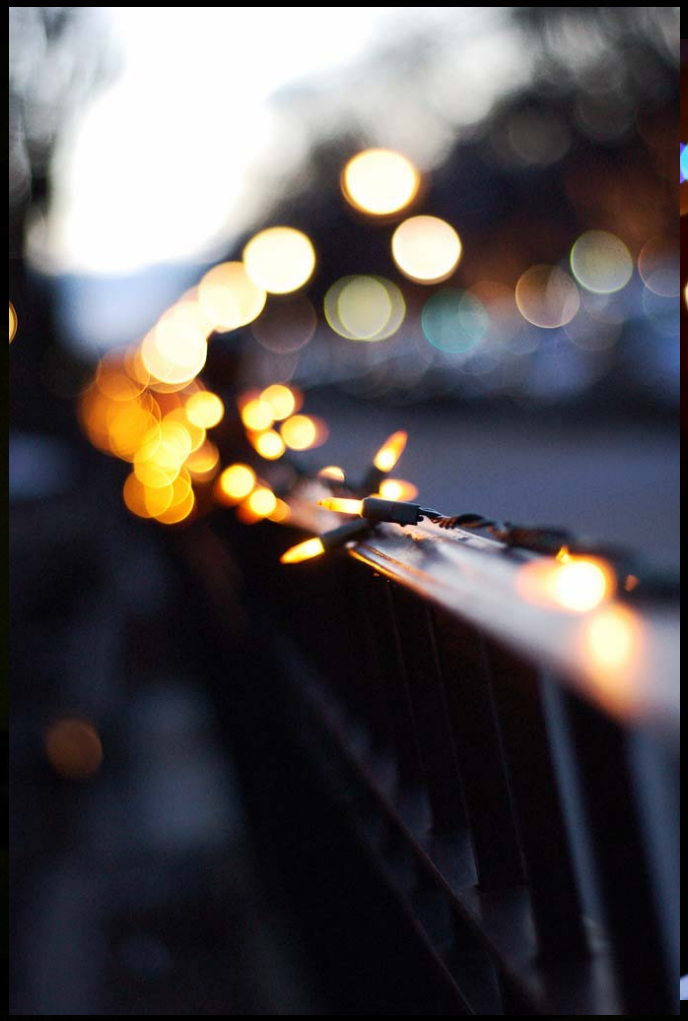
Separable Filters

- Subsurface scattering via texture space diffusion
- 2x faster using CUDA

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

Non Separable Filters

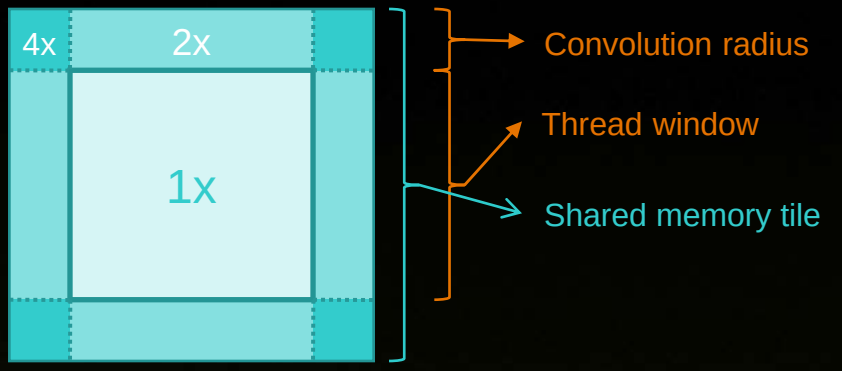
- High fidelity depth of field - bokeh



Images & argument from Vincent Scheib, Emergent

Non Separable Filters

- CUDA reduces bandwidth requirements dramatically
- Simple approach:
  - Load one tile per thread block
  - Most samples loaded only once
  - Apron loaded multiple times
- Optimized approach:
  - Sliding window



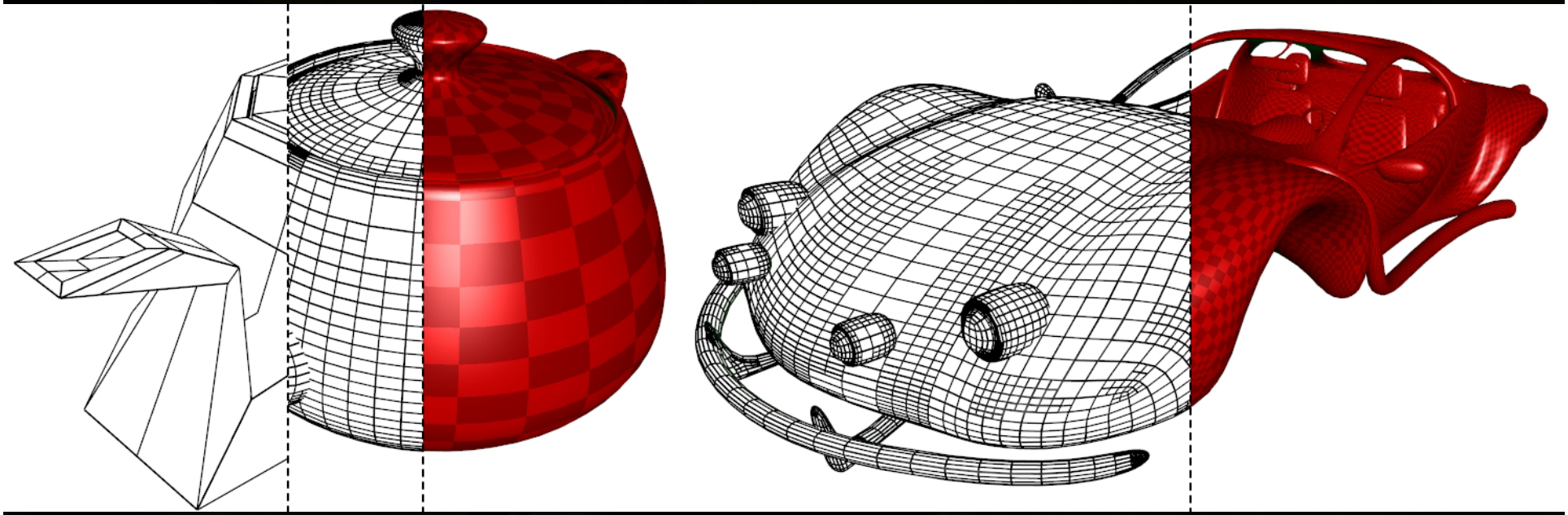
Post-Processing Effects

- Large convolutions
  - 2D FFT transform
  - Summed area tables
- Prevent edge bleeding
  - Bilateral filter
  - Simulated heat diffusion

© Per Lonroth, Mattias Unger, DICE

CUDA Tessellation

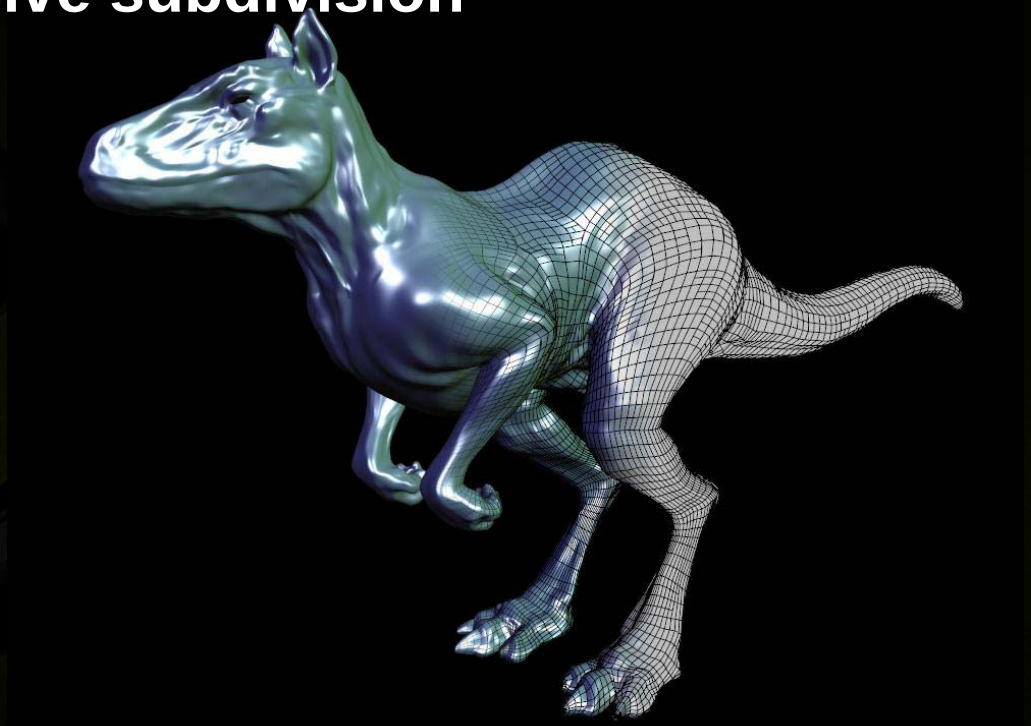
- Flexible adaptive geometry generation
- Recursive subdivision



Real-Time View-Dependent Rendering of Parametric Surfaces
Eisenacher, Meyer, Loop 2009

CUDA Tessellation

- **Flexible adaptive geometry generation**
- **Recursive subdivision**



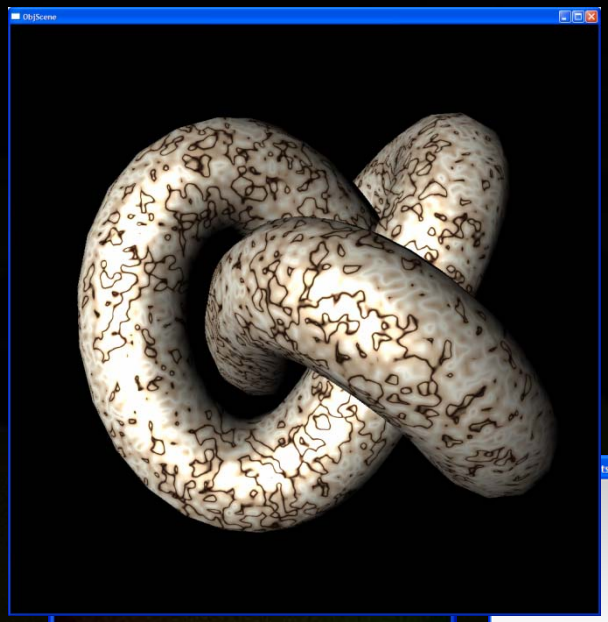
Real-Time Reyes-Style Adaptive Surface Subdivision
Patney & Owens 2009

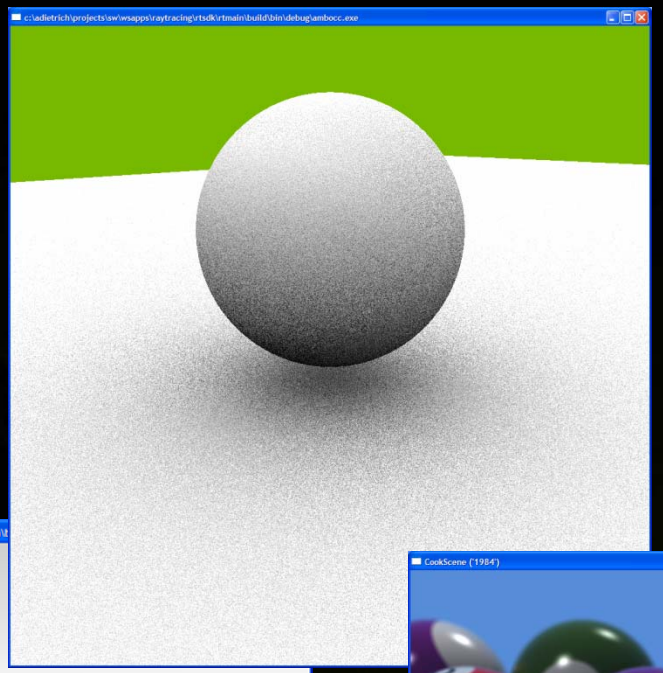
CUDA-Enabled Rendering Algorithms

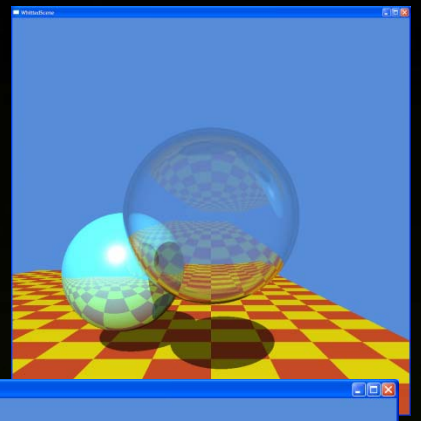
- Creation of irregular data structures
  - Alias Free Shadow Maps
- Traversal of complex data structures
  - Ray Tracing

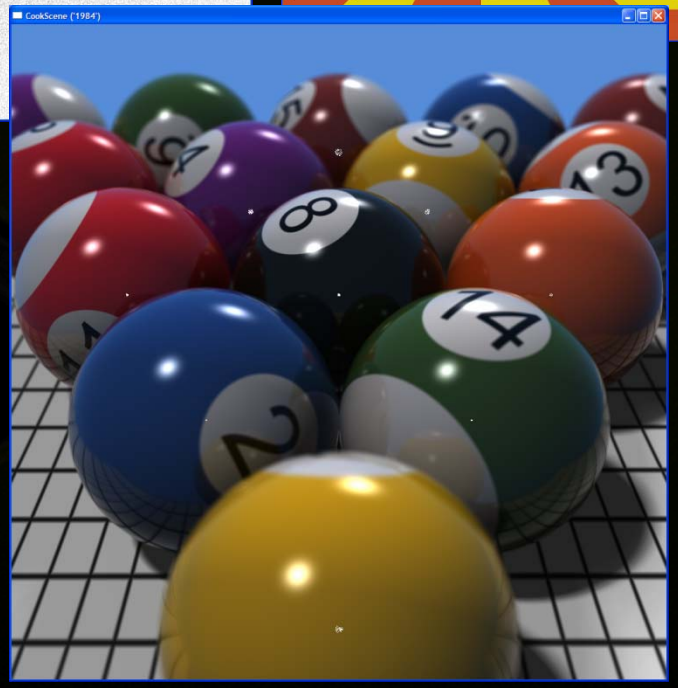
Raytracing

Upcoming CUDA Ray Tracing API










GPU Computing 4.0?
NVIDIA.

Key CUDA Challenges

- Express other programming models elegantly
  - Persistent thread blocks reading and writing work queues
  - Thread block or warp as a (parallel) task
  - Both common “power-user” CUDA patterns
- Foster more high-level languages & platforms
- Improve & mature development environment

Key GPU Workloads

- Computational graphics (but don’t forget DirectXn-1)
- Scientific and numeric computing
- Image processing – video & images
- Computer vision
- Speech & natural language
- Machine learning

Final Thoughts – Education

- We should teach parallel computing in CS 1 or CS 2
  - Computers don't get faster, just wider
  - Manycore is the ~~future~~ now of computing
- Heapsort and mergesort
  - Both O(n lg n)
  - One parallel-friendly, one not
  - Students need to understand this early


Questions?
dluebke@nvidia.com
NVIDIA.

Extra Slides

Motivation: NVIDIA

Supercomputing Performance

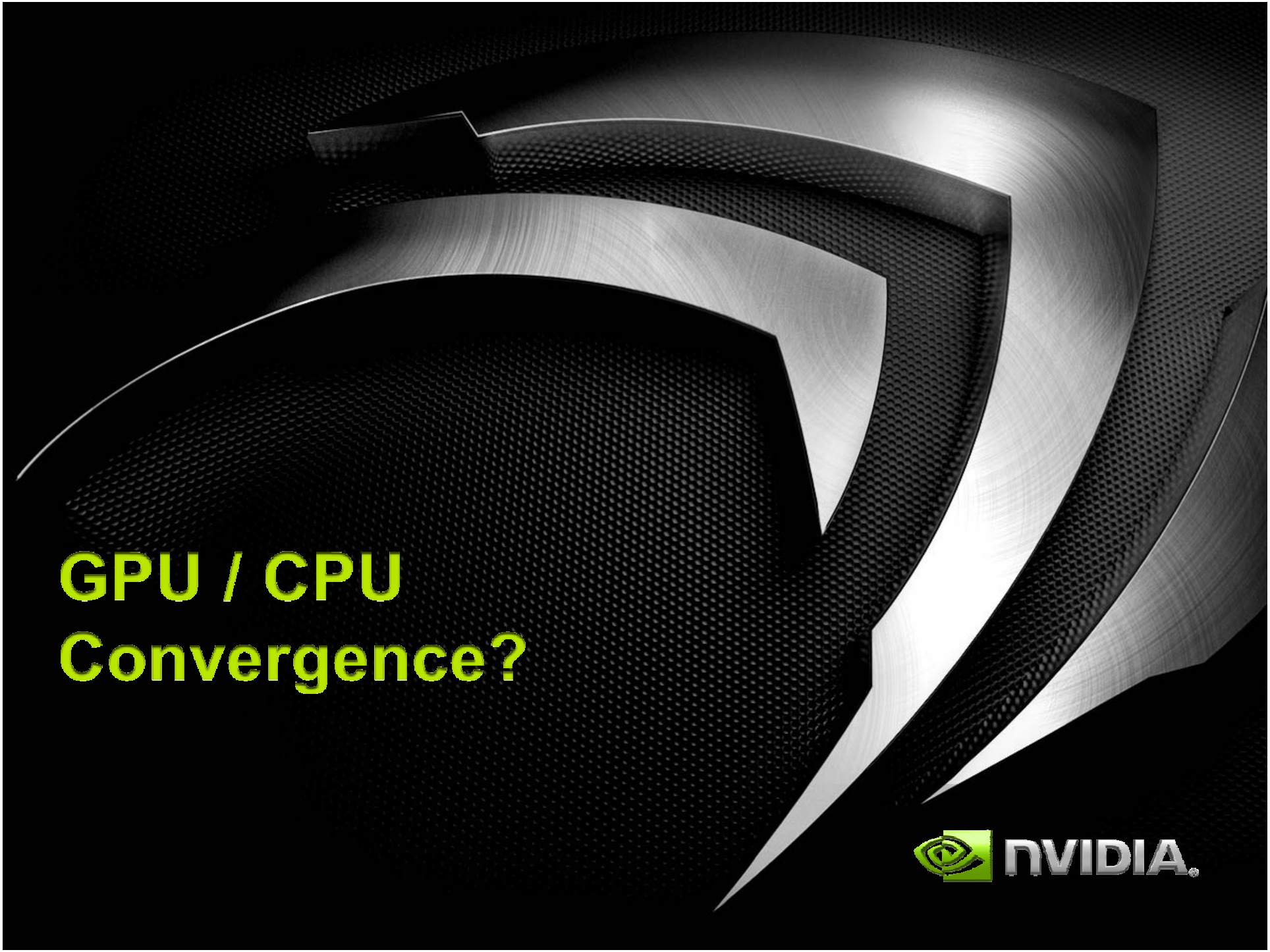
- 960 cores. 4 TeraFLOPS
- 250x the performance of a desktop

Personal

- One researcher, one supercomputer
- Plugs into standard power strip

Accessible

- Program in C for Windows, Linux
- Available now under $10,000


GPU / CPU
Convergence?
NVIDIA.

GPU / CPU Convergence?

- Yes
  - Actually, depends on the meaning of the word CPU
    A. Increasingly irrelevant serial processor OR
    B. The thing AMD & INTEL make to capture application cycles
  - A becomes a little dot on a chip
  - B converges with the GPU

What is a Game?

1. AI
   - Tera-scale GPU
2. Physics
   - Tera-scale GPU
3. Graphics
   - Tera-scale GPU
4. Perl Script
   - 5 sq mm^2 (1% of die) serial CPU
     - Important to have, along with
       - Video processing, dedicated display, DMA engines, etc.

The Future of Graphics Hardware?

- **Forward-looking statements:**

  All future interesting problems are *throughput* problems.

  GPUs will evolve to be *the* general-purpose throughput processors.

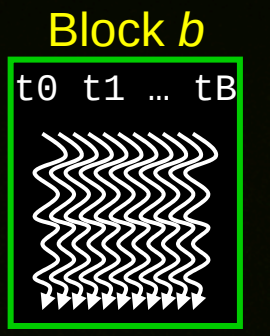
  CPUs will become (already are?) “good enough”, and shrink to a corner of the die.


CUDA
(in 3 slides)
NVIDIA.

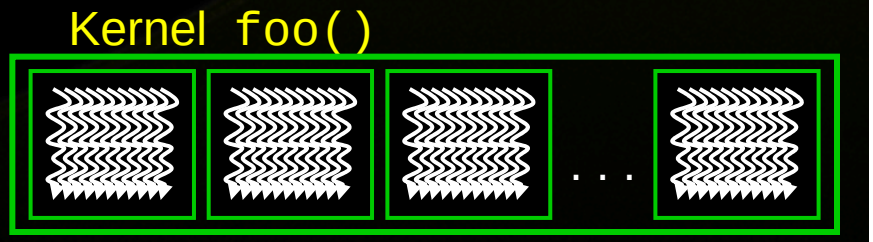
Hierarchy of concurrent threads

- Parallel **kernels** composed of many threads
  - all threads execute the same sequential program

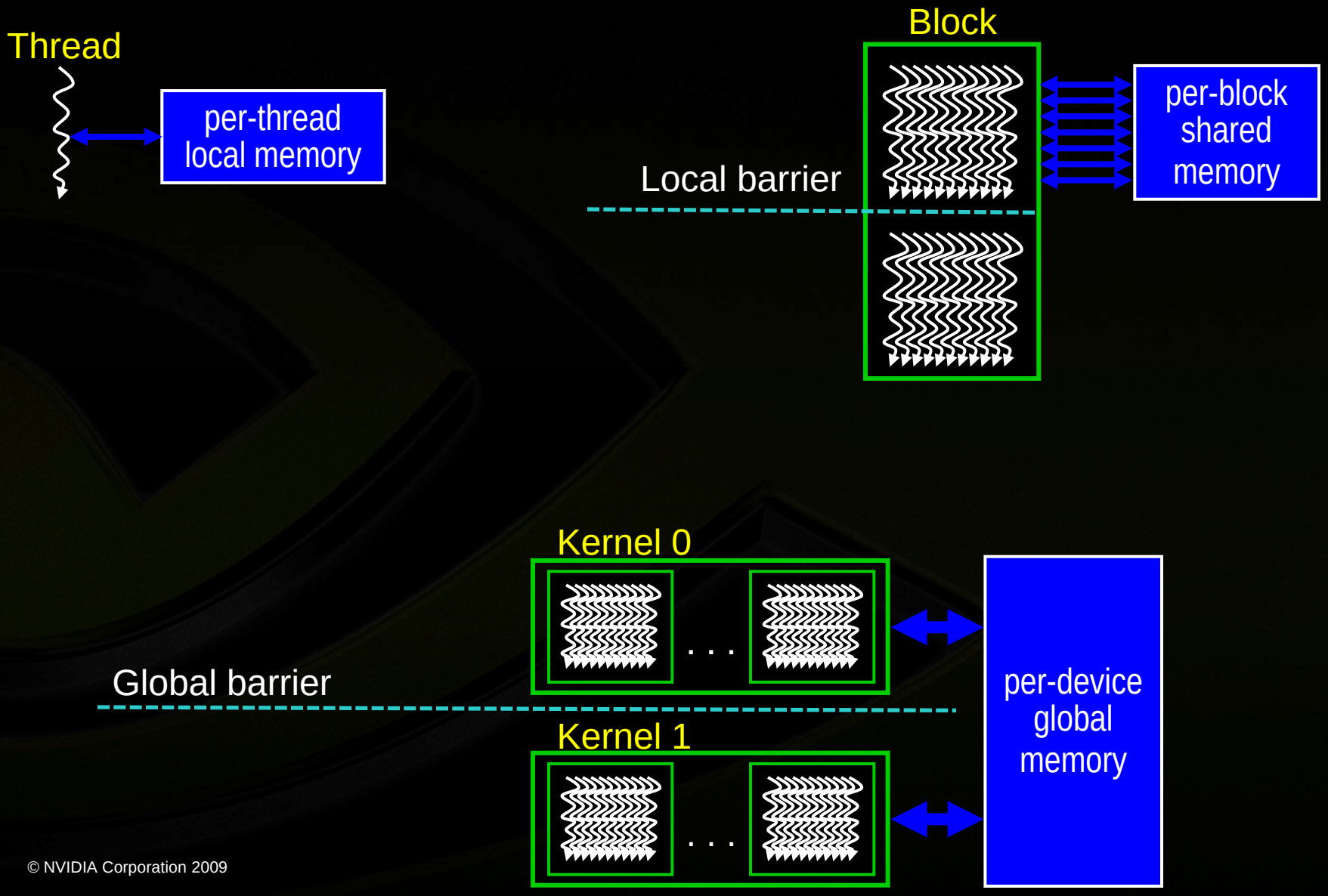
- Threads are grouped into **thread blocks**
  - threads in the same block can cooperate



- Threads/blocks have unique IDs



Hierarchical organization



Heterogeneous Programming

- C for CUDA = serial program with parallel kernels
  - Serial C code executes in a CPU thread
  - Parallel kernel C code executes in thread blocks across multiple processing elements