Parallel Computing with CUDA

Mark Harris

NVIDIA Developer Technology

NVIDIA.

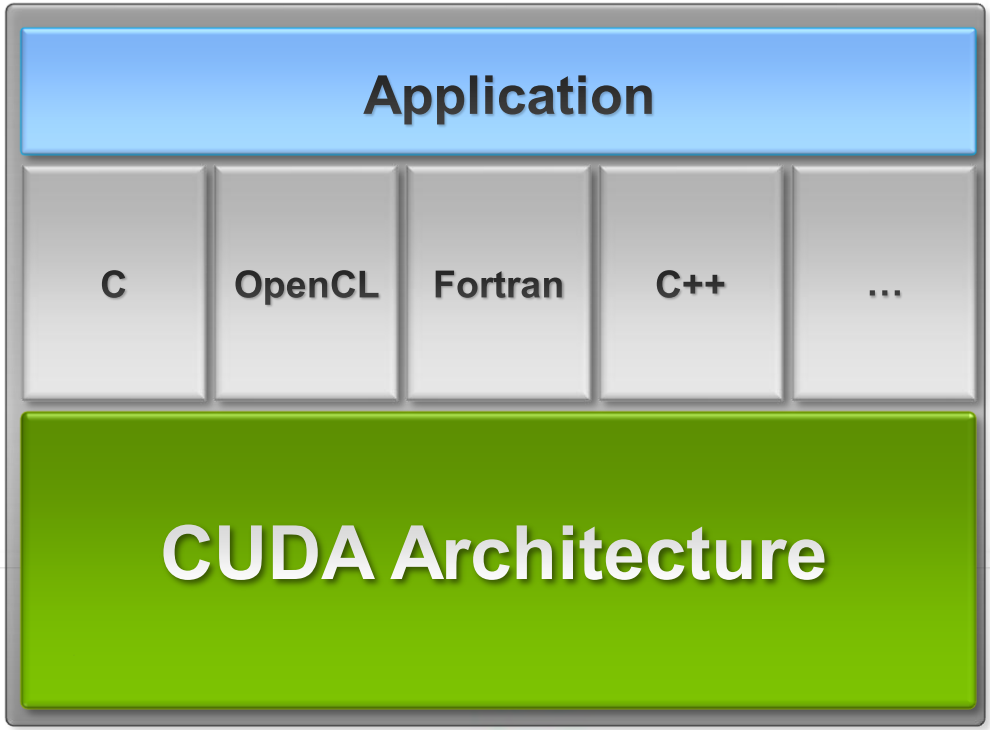
SIGGRAPHASIA2008

NEW HORIZONS

CUDA Parallel Computing Architecture

- ISA and hardware compute engine in NVIDIA GPUs
  - Exposes the computational horsepower of NVIDIA GPUs
  - Enables general-purpose GPU computing

- Architected to support any computational interface
  - Standard languages and APIs
  - C, OpenCL, etc.



C for CUDA

- C for CUDA provides a scalable parallel programming model and software environment for parallel computing
  - Minimal extensions to familiar C/C++ environment
  - Heterogeneous serial-parallel programming model

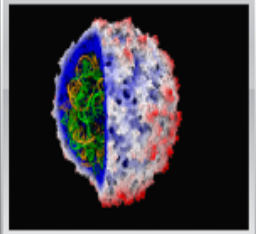
The Democratization of Parallel Computing

- CUDA brings parallel computing to the masses
  - Over 100M CUDA-capable GPUs sold
  - A “developer kit” costs ~$100 (for 500 GFLOPS!)
    - CUDA developer toolkit is free
- Data-parallel supercomputers are everywhere
  - Seeing innovations in data-parallel computing already
- Massively parallel computing is now a commodity technology
  - Enabling new science and new computer science

CUDA Application Examples

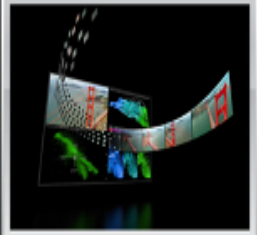
146X

Interactive visualization of volumetric white matter connectivity



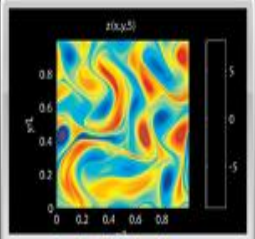
36X

Ion placement for molecular dynamics simulation



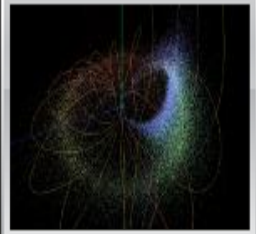
19X

Transcoding HD video stream to H.264



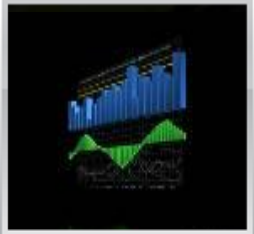
17X

Simulation in Matlab using .mex file CUDA function



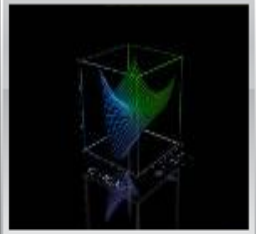
100X

Astrophysics N-body simulation



149X

Financial simulation of LIBOR model with swaptions



47X

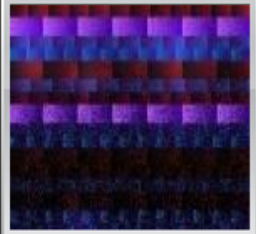
GLAME@lab: An M-script API for linear Algebra operations on GPU

20X

Ultrasound medical imaging for cancer diagnostics

24X

Highly optimized object oriented molecular dynamics



30X

Cmatch exact string matching to find similar proteins and gene sequences

CUDA Programming Model

Some Design Goals

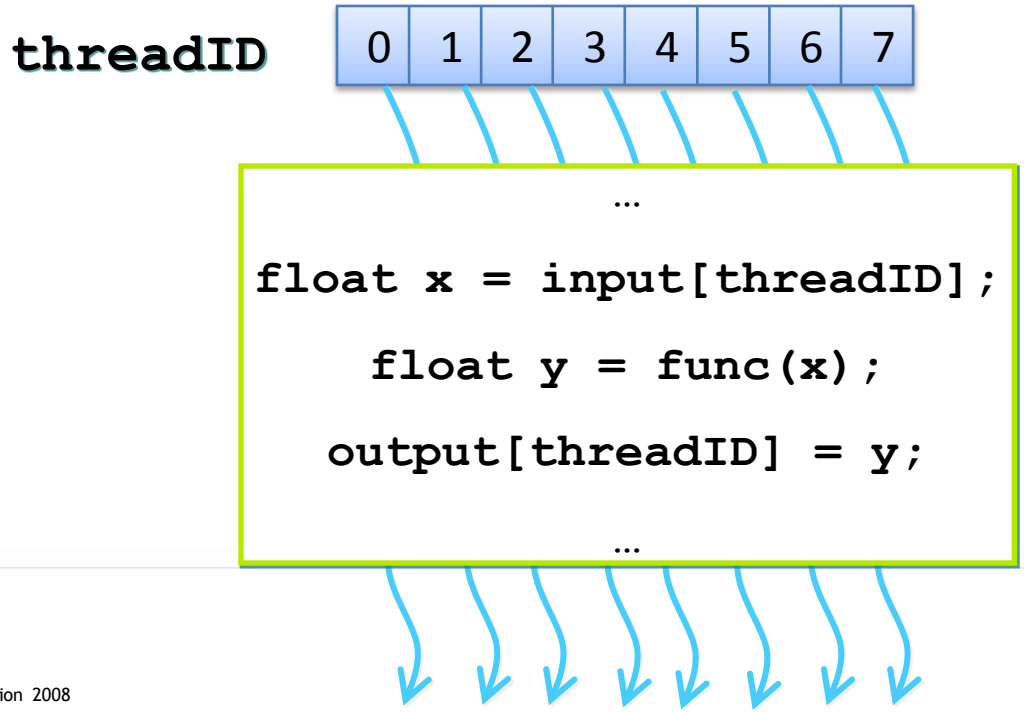
- Scale to 100's of cores, 1000's of parallel threads
- Let programmers focus on parallel algorithms
  - not mechanics of a parallel programming language
- Enable heterogeneous systems (i.e., CPU+GPU)
  - CPU good at serial computation, GPU at parallel

Key Parallel Abstractions in CUDA

- "Zillions" of lightweight threads
  - ➔ Simple decomposition model
- Hierarchy of concurrent threads
  - ➔ Simple execution model
- Lightweight synchronization primitives
  - ➔ Simple synchronization model
- Shared memory model for cooperating threads
  - ➔ Simple communication model

“Zillions” of Lightweight Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same program
  - Each thread uses its ID to compute addresses and make control decisions



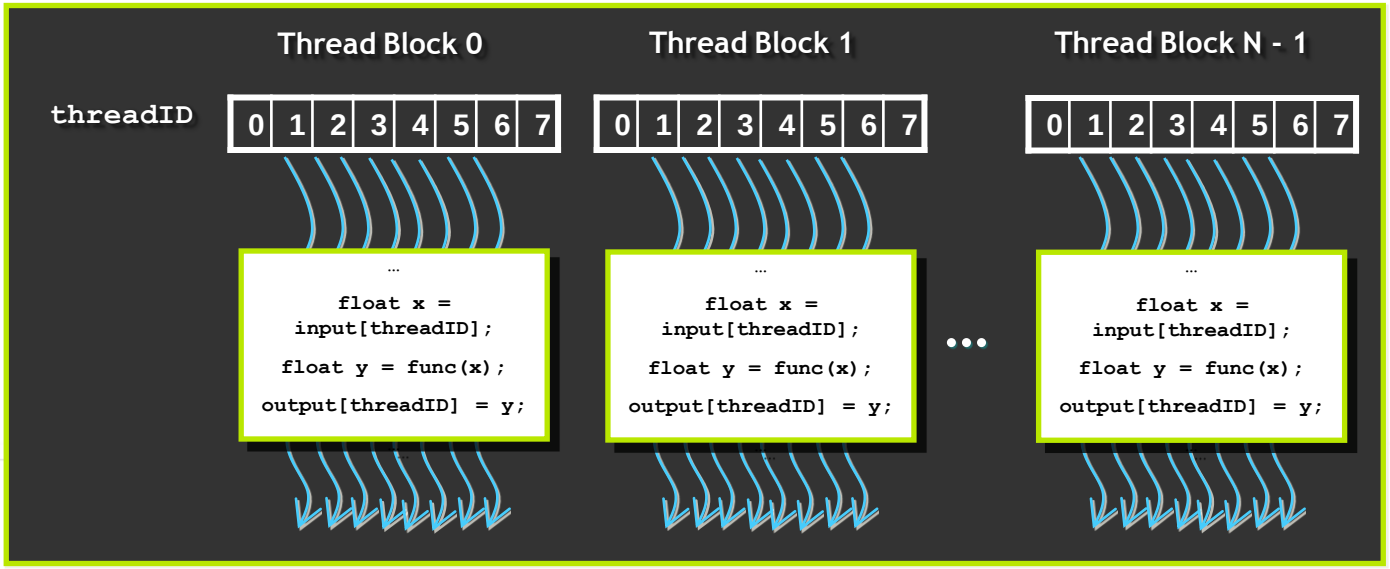
Thread Cooperation

- Threads might not be completely independent
  - Share results to save computation
  - Share memory accesses for drastic bandwidth reduction

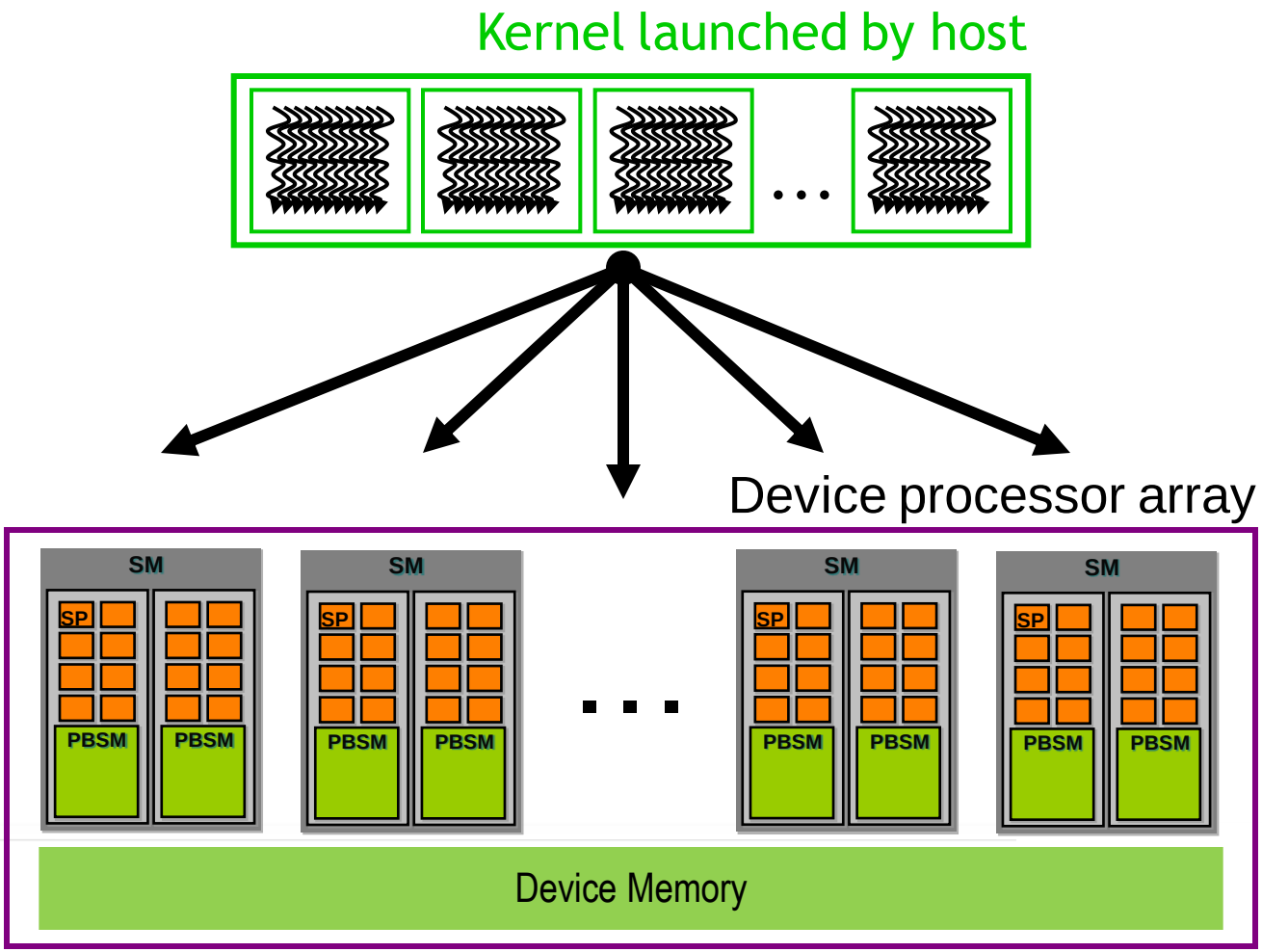
- Thread cooperation is a powerful feature of CUDA
  - Threads can cooperate via on-chip shared memory and lightweight synchronization primitives

Thread Blocks: Scalable Cooperation

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

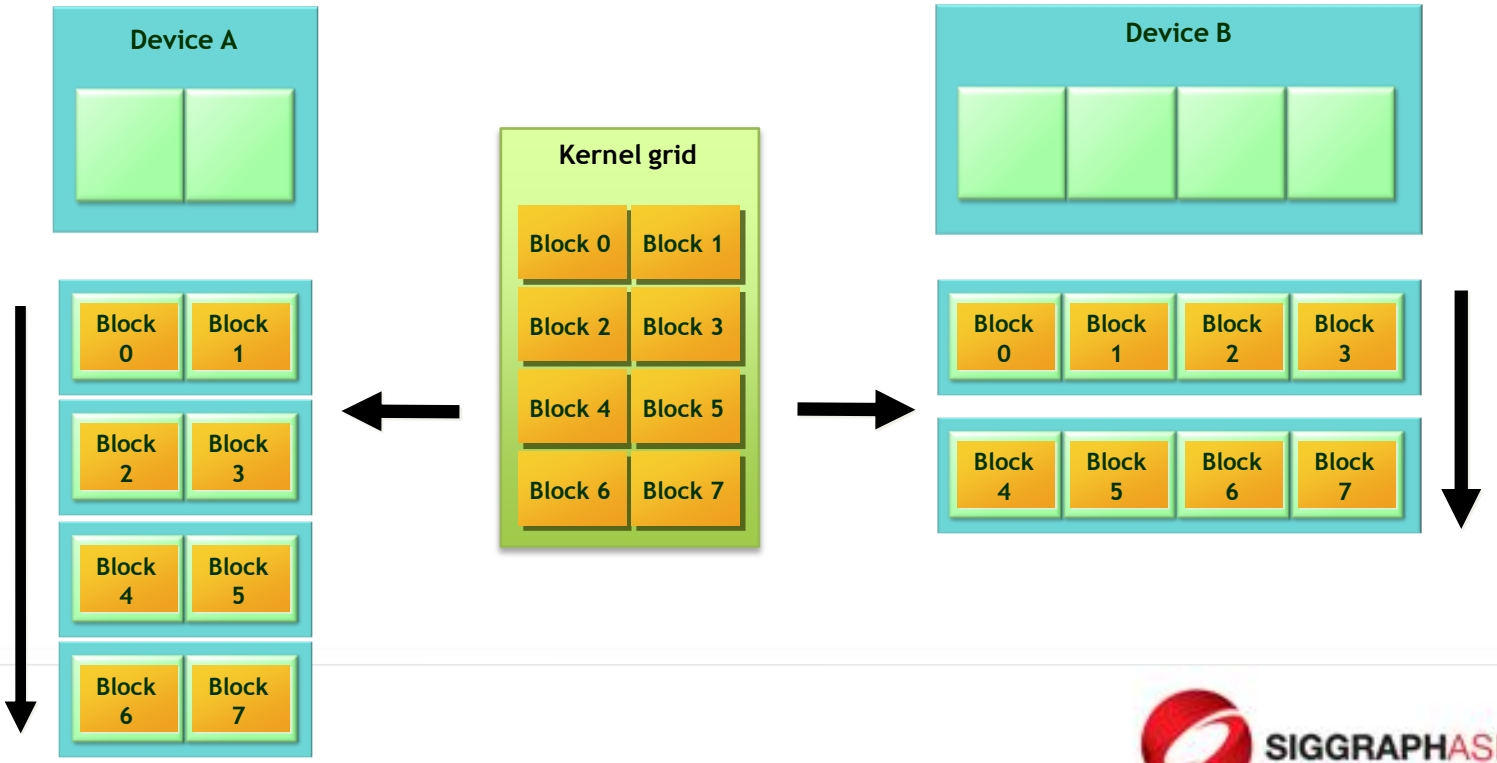


Blocks Run on Multiprocessors



Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
  - Kernels scale to any number of parallel multiprocessors



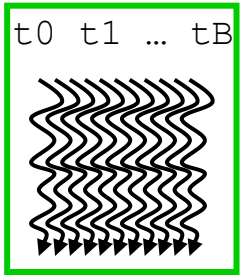
Hierarchy of concurrent threads

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

Thread *t*

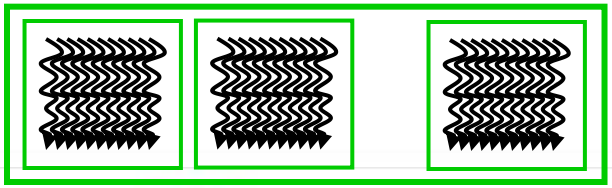
- Threads are grouped into thread blocks
  - threads in the same block can cooperate

Block *b*



- Kernel program executed by many thread blocks

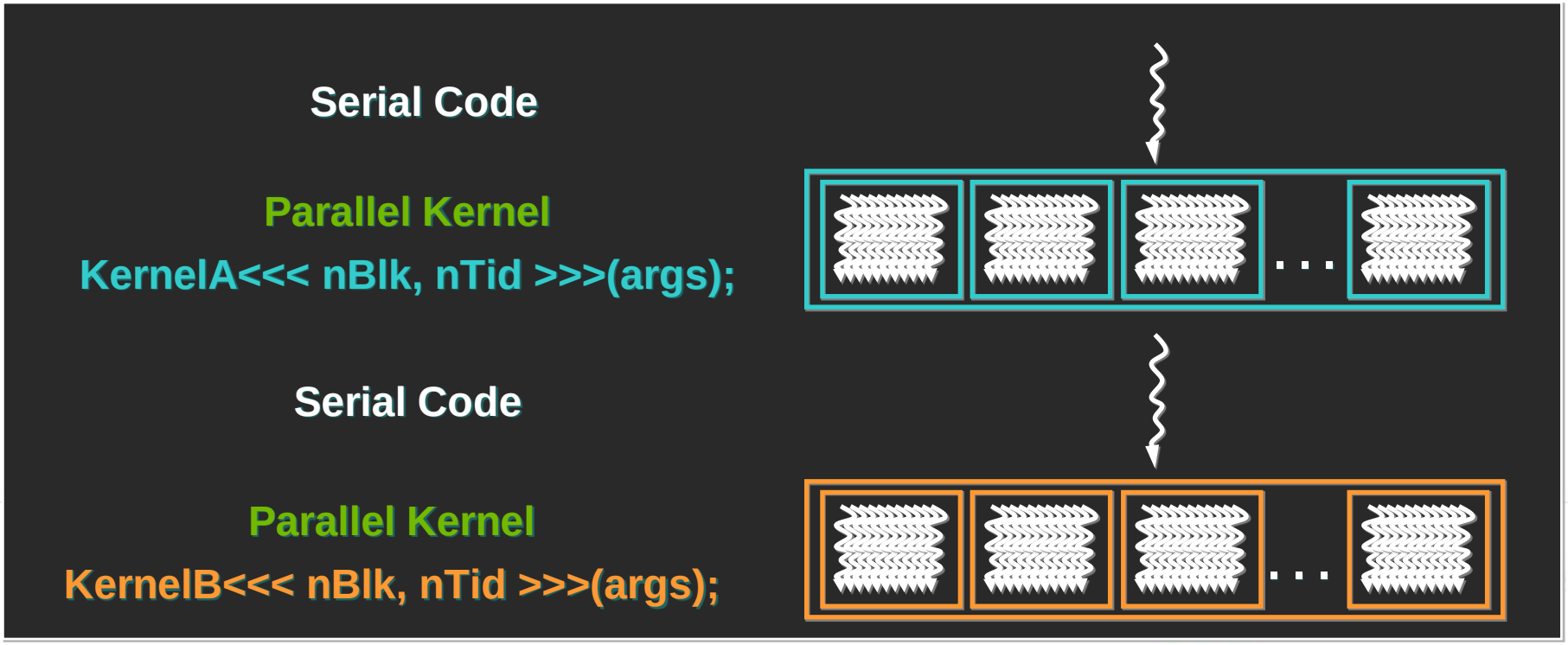
Kernel foo()



- Threads/blocks have unique IDs

Heterogeneous Programming

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



Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Device Code

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Host code for memory

```
// allocate host (CPU) memory
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
… initalize h_A and h_B …

// allocate device (GPU) memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),cudaMemcpyHostToDevice));
cudaMemcpy( d_B, h_B, N * sizeof(float),cudaMemcpyHostToDevice));
```

Synchronization & Coordination

- Threads within block may synchronize with **barriers**

  … Step 1 …

  __syncthreads();

  … Step 2 …

- Blocks **coordinate** via atomic memory operations
  - e.g., increment shared queue pointer with **atomicInc()**

- Implicit barrier between **dependent kernels**

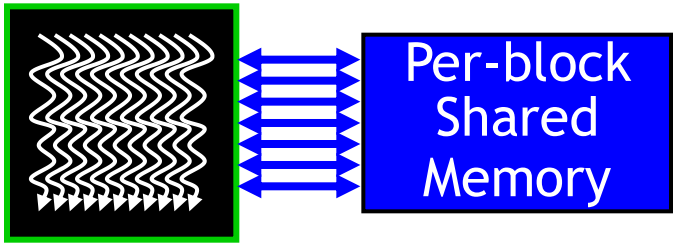
  vec_minus<<<nblocks, blksize>>>(a, b, c);

  vec_dot<<<nblocks, blksize>>>(c, c);

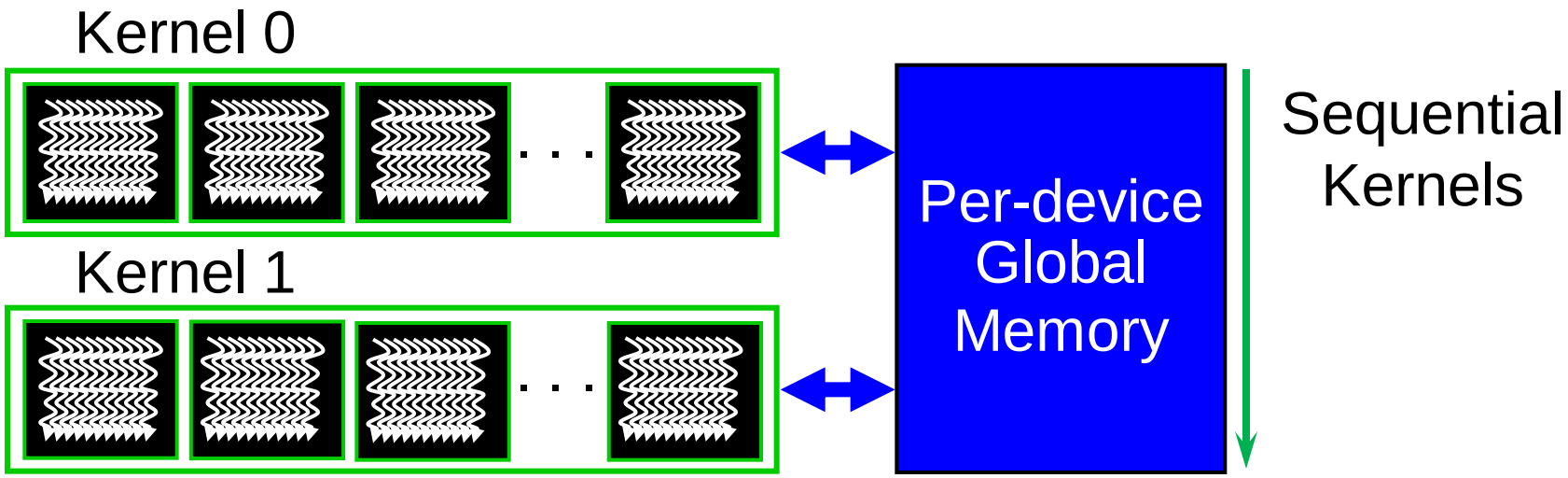
Levels of Parallelism

- Thread parallelism
  - each thread is an independent thread of execution
- Data parallelism
  - across threads in a block
  - across blocks in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels

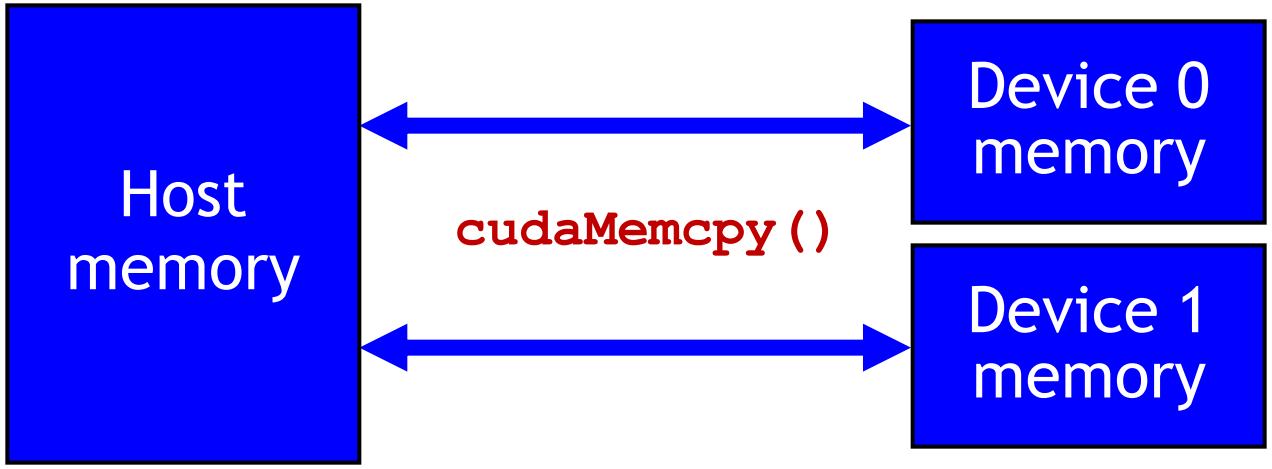
Memory model



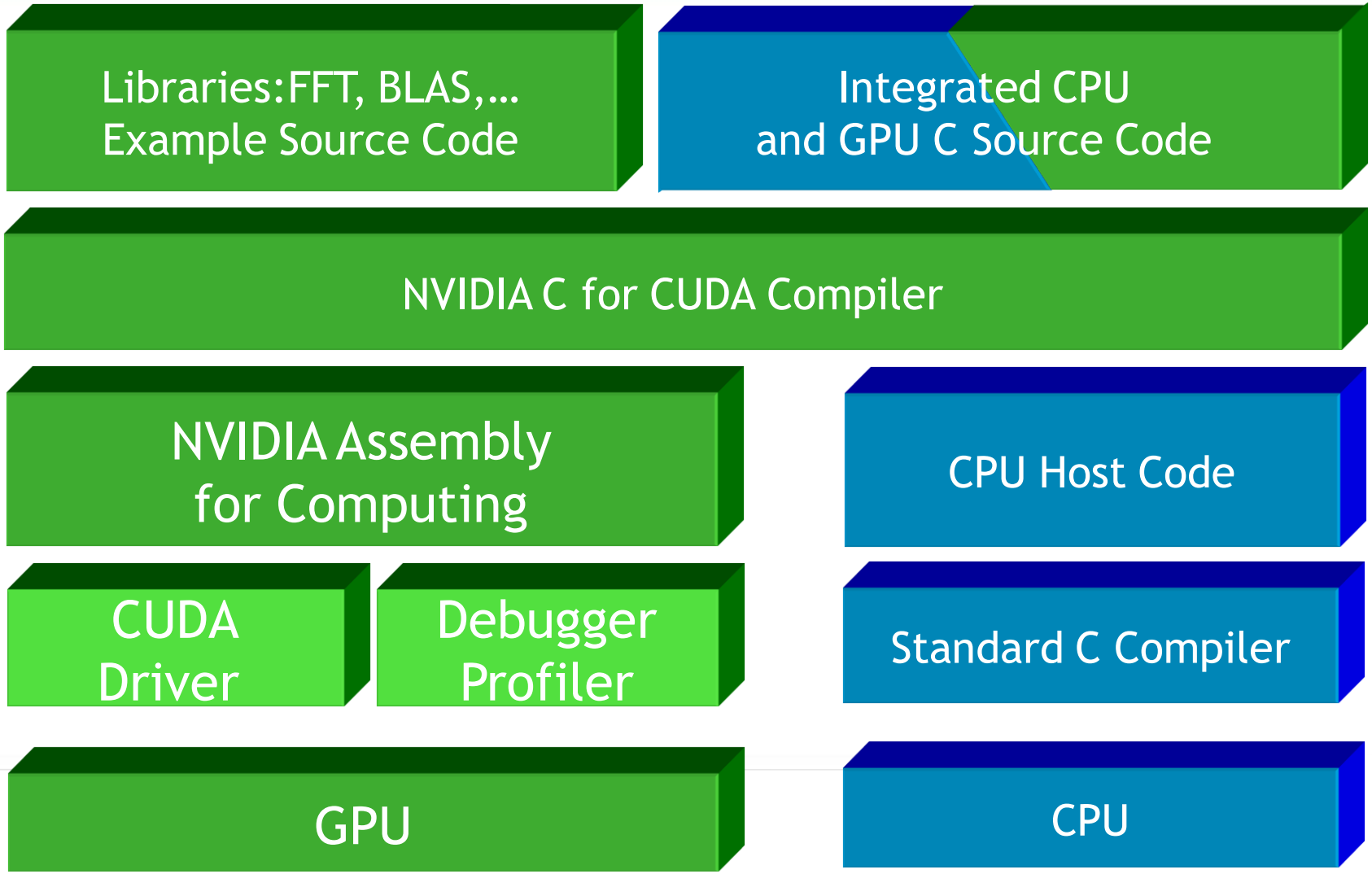
Memory model



Memory model



CUDA SDK


Libraries:FFT, BLAS,...
Example Source Code
Integrated CPU
and GPU C Source Code
NVIDIA C for CUDA Compiler
NVIDIA Assembly
for Computing
CPU Host Code
CUDA
Driver
Debugger
Profiler
Standard C Compiler
GPU
CPU

Graphics Interoperability

- Go “Beyond Programmable Shading”
- Access textures from within CUDA
  - Free filtering, wrapping, 2/3D addressing, spatial caching...
- Share resources (vertex, pixel, texture, surface)
  - Map resources into CUDA global memory, manipulate freely
  - Geometry: physics (collision, response, deformation, ...), lighting, geometry creation, non-traditional rasterization, acceleration structures, AI, ...
  - Imaging: image processing, (de)compression, procedural texture generation, postprocess effects, computer vision, ...

OpenGL Interop Steps

- Register a buffer object with CUDA
  - `cudaGLRegisterBufferObject(GLuint buffObj);`
  - OpenGL can use a registered buffer only as a source
  - Unregister the buffer prior to rendering to it by OpenGL
- Map the buffer object to CUDA memory
  - `cudaGLMapBufferObject(void **devPtr, GLuint buffObj);`
  - Returns an address in global memory
  - Buffer must registered prior to mapping
- Launch a CUDA kernel to process the buffer
- Unmap the buffer object prior to use by OpenGL
  - `cudaGLUnmapBufferObject(GLuint buffObj);`
- Unregister the buffer object
  - `cudaGLUnregisterBufferObject(GLuint buffObj);`
  - Optional: needed if the buffer is a render target
- Use the buffer object in OpenGL code

Summary

- CUDA provides a powerful parallel programming model
  - Heterogeneous - mixed serial-parallel programming
  - Scalable - hierarchical thread execution model
  - Accessible – C for CUDA uses minimal but expressive changes to C
  - Interoperable - simple graphics interop mechanisms

- CUDA is an attractive platform
  - Broad - OpenGL, DirectX, WinXP, Vista, Linux, MacOS
  - Widespread - over 100M CUDA GPUs, 30K CUDA developers

CUDA provides tremendous scope for innovative graphics research "beyond programmable shading"

http://www.nvidia.com/cuda

Questions?

Mark Harris
mharris@nvidia.com

CUDA
Design Choices

Goal: Scalability

- Scalable execution
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling
- Hierarchical execution model
  - Decompose problem into sequential steps (kernels)
  - Decompose kernel into computing parallel blocks
  - Decompose block into computing parallel threads
- Hardware distributes *independent* blocks to SMs as available

Goal: easy to program

- Strategies:
  - Familiar programming language mechanics
    - C/C++ with small extensions
  - Simple parallel abstractions
    - Simple barrier synchronization
    - Shared memory semantics
    - *Hardware-managed* hierarchy of threads

Hardware Multithreading

- Hardware allocates resources to blocks
  - blocks need: thread slots, registers, shared memory
  - blocks don't run until resources are available
- Hardware schedules threads
  - threads have their own registers
  - any thread not waiting for something can run
  - context switching is (basically) free – every cycle
- Hardware relies on threads to hide latency
  - i.e., parallelism is necessary for performance

Graphics Interop Details

Interop Scenario:
Dynamic CUDA-generated texture

- Register the texture PBO with CUDA
- For each frame:
  - Map the buffer
  - Generate the texture in a CUDA kernel
  - Unmap the buffer
  - Update the texture
  - Render the textured object

```
unsigned char *p_d=0;
cudaGLMapBufferObject((void**)&p_d, pbo);
prepTexture<<<height,width>>>(p_d, time);
cudaGLUnmapBufferObject(pbo);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
glBindTexture(GL_TEXTURE_2D, texID);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0,0, 256,256,
                GL_BGRA, GL_UNSIGNED_BYTE, 0);
```

Interop Scenario:
Frame Post-processing by CUDA

- For each frame:
  - Render to PBO with OpenGL
  - Register the PBO with CUDA
  - Map the buffer
  - Process the buffer with a CUDA kernel
  - Unmap the buffer
  - Unregister the PBO from CUDA

```
unsigned char *p_d=0;
cudaGLRegisterBufferObject(pbo);
cudaGLMapBufferObject((void**)&p_d, pbo);
postProcess<<<blocks,threads>>>(p_d);
cudaGLUnmapBufferObject(pbo);
cudaGLUnregisterBufferObject(pbo);
...
```

Texture Fetch in CUDA

- Advantages of using texture:
  - Texture fetches are cached
    - optimized for 2D locality (swizzled)
  - Textures are addressable in 2D
    - using integer or normalized coordinates
    - means fewer addressing calculations in code
  - Get filtering for free
    - bilinear interpolation, anisotropic filtering etc.
  - Free wrap modes (boundary conditions)
    - clamp to edge
    - repeat
- Disadvantages:
  - Textures are read-only
  - May not be a big performance advantage if global reads are already coalesced

Texture Fetch in CUDA

- Textures represent 1D, 2D or 3D arrays of memory
- CUDA texture type defines type of components and dimension:

```
texture<type, dim>
```

- texfetch function performs texture fetch:

```
texfetch(texture<> t,
         float x, float y=0, float z=0)
```

Channel Descriptors

- Channel descriptor is used in conjunction with texture and describes how reads from global memory are interpreted
- Predefined channel descriptors exist for built-in types, e.g.:

```
cudaChannelFormatDesc
cudaCreateChannelDesc<float>(void);
cudaChannelFormatDesc
cudaCreateChannelDesc<float4>(void);
cudaChannelFormatDesc
cudaCreateChannelDesc<uchar4>(void);
```

Arrays

- Arrays are the host representation of n-dimensional textures

- To allocate an array

```
cudaError_t
cudaMalloc( cudaArray_t **arrayPtr,
    struct cudaChannelFormatDesc &desc,
    size_t width,
    size_t height = 1);
```

- To free an array:

```
cudaError_t
cudaFree( struct cudaArray *array);
```

Loading Memory from Arrays

- Load memory to arrays using a version of `cudaMemcpy`:

```
cudaError_t cudaMemcpy(
        void                    *dst,
  const struct cudaArray        *src,
        size_t                   count,
        enum cudaMemcpyKind      kind
)
```

Binding Textures

- Textures are bound to arrays using:

```
template<class T, int dim, bool norm>
 cudaError_t cudaBindTexture(
    const struct texture<T, dim, norm> &tex,
    const struct cudaArray             *array
 )
```

- To unbind:

```
template<class T, int dim, bool norm>
cudaError_t cudaUnbindTexture(
        const struct texture<T, dim, norm> &tex
)
```

Example

```
cudaArray* cu_array;
texture<float, 2>  tex;

// Allocate array
cudaMalloc( &cu_array, cudaCreateChannelDesc<float>(),
         width, height );

// Copy image data to array
cudaMemcpy( cu_array, image, width*height, cudaMemcpyHostToDevice);

// Bind the array to the texture
cudaBindTexture( tex, cu_array);

// Run kernel
dim3 blockDim(16, 16, 1);
dim3 gridDim(width / blockDim.x, height / blockDim.y, 1);
kernel<<< gridDim, blockDim, 0 >>>(d_odata, width, height);

cudaUnbindTexture(tex);
```

Example (cont)

```
__global__ void
kernel(float* odata, int height, int width)
{
 unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
 unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
 float c = texfetch(tex, x, y);
 odata[y*width+x] = c;
}
```

CUDA Programming

NVIDIA

SIGGRAPHASIA2008
NEW HORIZONS

CUDA: Features available to kernels

- Standard mathematical functions

  `sinf`, `powf`, `atanf`, `ceil`, etc.

- Built-in vector types

  `float4`, `int4`, `uint4`, etc. for dimensions 1..4

- Texture accesses in kernels

```
texture<float,2> my_texture;  // declare texture reference

float4 texel = texfetch(my_texture, u, v);
```

C for CUDA = C with Extensions

- Philosophy: provide minimal set of extensions necessary to expose power

- Function qualifiers:

```
__global__ void MyKernel() { }
__device__ float MyDeviceFunc() { }
```

- Variable qualifiers:

```
__constant__ float MyConstantArray[32];
__shared__   float MySharedArray[32];
```

- Execution configuration:

```
dim3 dimGrid(100, 50);  // 5000 thread blocks
dim3 dimBlock(4, 8, 8); // 256 threads per block
MyKernel <<< dimGrid, dimBlock >>> (...); // Launch kernel
```

- Built-in variables and functions valid in device code:

```
dim3 gridDim;    // Grid dimension
dim3 blockDim;   // Block dimension
dim3 blockIdx;   // Block index
dim3 threadIdx;  // Thread index
void   syncthreads(); // Thread synchronization
```

CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory

  `cudaMalloc()`, `cudaFree()`

- Explicit memory copy for host ↔ device, device ↔ device

  `cudaMemcpy()`, `cudaMemcpy2D()`, ...

- Texture management

  `cudaBindTexture()`, `cudaBindTextureToArray()`, ...

- OpenGL & DirectX interoperability

  `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...

Myths of GPU Computing

NVIDIA.

SIGGRAPHASIA2008
NEW HORIZONS

Myths of GPU Computing

- **GPUs layer normal programs on top of graphics**
  - NO: CUDA compiles directly to the hardware
- **GPUs 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**

Myths of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPUs 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

Myths of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPUs 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

Myths of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPUs 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

Myths of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPUs 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

Myths of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPUs architectures are:
  - ~~Very wide (1000s) SIMD machines…~~
  - ~~…on which branching is impossible or prohibitive…~~
  - …with 4-wide vector registers. NO: scalar thread processors
- GPUs are power-inefficient
- GPUs don't do real floating point

Myths of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPUs 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

Myths of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPUs architectures are:
  - ~~Very wide (1000s) SIMD machines…~~
  - ~~…on which branching is impossible or prohibitive…~~
  - ~~…with 4-wide vector registers.~~
- GPUs are power-inefficient:

  No – 4-10x perf/W advantage, up to 89x reported for some studies
- GPUs don't do real floating point

Myths of GPU Computing

- ~~GPUs layer normal programs on top of graphics~~
- GPUs 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

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**
log2(x) and 2^x estimates accuracy	**23 bit**	**No**	**No**

GPU 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 FP?

- G8x GPU FP is IEEE 754
  - Comparable to other processors / accelerators
  - More precise / usable in some ways
  - Less precise in other ways

- GPU FP getting better every generation
  - Double precision support in GTX 20
  - Goal: best of class by 2009