The “New” Moore’s Law

• Computers no longer get faster, just wider
  - Many people have not yet gotten this memo

• You **must** re-think your algorithms to be parallel!
  - Not just a good idea - the only way to gain performance
  - Otherwise: if it’s not fast enough now, it never will be

• Data-parallel computing is most scalable solution
  - Otherwise: refactor code for 2 cores  4 cores  8 cores  16 cores...
  - You will always have more data than cores - build the computation around the data
Enter the GPU

- Massive economies of scale
- Massively parallel
• **CUDA** is a scalable parallel programming model and a software environment for parallel computing
  - Minimal extensions to familiar C/C++ environment
  - Heterogeneous serial-parallel programming model

• NVIDIA’s GPU architecture accelerates CUDA
  - Expose the computational horsepower of NVIDIA GPUs
  - Enable general-purpose *GPU computing*

(CUDA also maps well to multicore CPUs...)
Beyond Programmable Shading: In Action

The Democratization of Parallel Computing

• GPU Computing with CUDA brings parallel computing to the masses
  - Over 85,000,000 CUDA-capable GPUs sold
  - A “developer kit” costs ~$100 (for 500 GFLOPS!)

• Data-parallel supercomputers are everywhere
  - CUDA makes this power accessible
  - We’re already seeing innovations in data-parallel computing

Massively parallel computing has become a commodity technology
Beyond Programmable Shading: In Action

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

0. Zillions of lightweight threads
   ➔ Simple decomposition model

1. Hierarchy of concurrent threads
   ➔ Simple execution model

2. Lightweight synchronization primitives
   ➔ Simple synchronization model

3. Shared memory model for cooperating threads
   ➔ Simple communication model
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
Heterogeneous Programming

- 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

Serial Code

Parallel Kernel

```
foo<<< nBlk, nTid >>>(args);
```

Serial Code

Parallel Kernel

```
bar<<< nBlk, nTid >>>(args);
```
Example: Vector Addition Kernel

```c
// 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];
}

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

Beyond Programmable Shading: In Action
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];
}

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

• What is a thread?
  - An independent thread of execution
  - Has its own PC, variables (registers), processor state, etc.

• What is a thread block?
  - A virtualized multiprocessor
  - Fit processors to data & computation, each kernel launch

  - A (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want
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
Blocks Run on Multiprocessors

Kernel launched by host

Device processor array

Device Memory

Beyond Programmable Shading: In Action
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
Beyond Programmable Shading: In Action

Memory model

Thread

Per-thread Local Memory

Block

Per-block Shared Memory
Memory model

Kernel 0

Kernel 1

Per-device Global Memory

Sequential Kernels
Memory model

- Host memory
- Device 0 memory
- Device 1 memory

`cudaMemcpy()`
CUDA SDK

- Libraries: FFT, BLAS, ...
- Example Source Code

- Integrated CPU and GPU C Source Code

- NVIDIA C Compiler

- NVIDIA Assembly for Computing

- CUDA Driver
- Debugger Profiler

- CPU Host Code
- Standard C Compiler
- CPU
Beyond Programmable Shading: In Action

• **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, ...
Summary

- **CUDA is a powerful parallel programming model**
  - Heterogeneous - mixed serial-parallel programming
  - Scalable - hierarchical thread execution model
  - Accessible - 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 85M CUDA GPUs, 60K CUDA developers

*CUDA provides tremendous scope for innovative graphics research “beyond programmable shading”*
Questions?
David Luebke
dluebke@nvidia.com

SIGGRAPH 2008
Beyond Programmable Shading: In Action
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
Beyond Programmable Shading: In Action

Blocks Run on Multiprocessors

Kernel launched by host

Device processor array

Device Memory
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
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
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

```c
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

```c
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:

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

To free an array:

```c
cudaError_t cudaFree( struct cudaArray *array);
```
Loading Memory from Arrays

• Load memory to arrays using a version of cudaMemcpy:

```c
cudaError_t cudaMemcpy(
    void *dst,
    const struct cudaArray *src,
    size_t count,
    enum cudaMemcpyKind kind
)
```
• Textures are bound to arrays using:

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

• To unbind:

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

```c
__global__ void
kernelfloat* 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;
}
```
GPU Computing with CUDA

- **CUDA = Compute Unified Driver Architecture**
  - Co-designed hardware & software for direct GPU computing

- **Hardware: fully general data-parallel architecture**
  - General thread launch
  - Global load-store
  - Parallel data cache
  - Scalable data-parallel execution/memory model
  - Scalar architecture
  - Integers, bit operations
  - Double precision (shortly)
  - C with minimal yet powerful extensions

- **Software: program the GPU in C**
Beyond Programmable Shading: In Action

CUDA SDK

Libraries: FFT, BLAS, ...
Example Source Code

Integrated CPU
and GPU C Source Code

NVIDIA C Compiler

NVIDIA Assembly
for Computing

CUDA Driver

Debugger
Profiler

GPU

CPU Host Code

Standard C Compiler

CPU
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);
  ```
Beyond Programmable Shading: In Action

G8x CUDA = C with Extensions

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

- **Function qualifiers:**
  ```c
  __global__ void MyKernel() { }
  __device__ float MyDeviceFunc() { }
  ```

- **Variable qualifiers:**
  ```c
  __constant__ float MyConstantArray[32];
  __shared__ float MySharedArray[32];
  ```

- **Execution configuration:**
  ```c
  dim3 dimGrid(100, 50);
  dim3 dimBlock(4, 8, 8);
  MyKernel <<< dimGrid, dimBlock >>> (...);
  ```

- **Built-in variables and functions valid in device code:**
  ```c
  dim3 gridDim;
  dim3 blockDim;
  dim3 blockIdx;
  dim3 threadIdx;
  void __syncthreads();
  ```
CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  \texttt{cudaMalloc()}, \texttt{cudaFree()}

- Explicit memory copy for host ↔ device, device ↔ device
  \texttt{cudaMemcpy()}, \texttt{cudaMemcpy2D()}, \ldots

- Texture management
  \texttt{cudaBindTexture()}, \texttt{cudaBindTextureToArray()}, \ldots

- OpenGL & DirectX interoperability
  \texttt{cudaGLMapBufferObject()}, \texttt{cudaD3D9MapVertexBuffer()}, \ldots
Example: Vector Addition Kernel

```c
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
Example: Invoking the Kernel

__global__ void vecAdd(float* A, float* B, float* C);

void main()
{
    // Execute on N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}

Beyond Programmable Shading: In Action
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));
... initialize 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);

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
A quick review

- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = group of SIMD threads that execute a kernel and can communicate via shared memory

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A - resident</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Precision</th>
<th>NVIDIA Tesla T10</th>
<th>IEEE 754</th>
<th>IEEE 754</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rounding modes for FADD and FMUL</td>
<td>All 4 IEEE, round to nearest, zero, Inf, -Inf</td>
<td>All 4 IEEE, round to nearest, zero, Inf, -Inf</td>
<td>All 4 IEEE, round to nearest, zero, Inf, -Inf</td>
</tr>
<tr>
<td>Denormal handling</td>
<td>Full speed</td>
<td>Supported, costs 1000’s of cycles</td>
<td>Supported only for results, not input operands (input denormals flushed-to-zero)</td>
</tr>
<tr>
<td>NaN support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Overflow and Infinity support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Flags</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>FMA</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Square root</td>
<td>Software with low-latency FMA-based convergence</td>
<td>Hardware</td>
<td>Software only</td>
</tr>
<tr>
<td>Division</td>
<td>Software with low-latency FMA-based convergence</td>
<td>Hardware</td>
<td>Software only</td>
</tr>
<tr>
<td>Reciprocal estimate accuracy</td>
<td>24 bit</td>
<td>12 bit</td>
<td>12 bit + step</td>
</tr>
<tr>
<td>Reciprocal sqrt estimate accuracy</td>
<td>23 bit</td>
<td>12 bit</td>
<td>12 bit + step</td>
</tr>
<tr>
<td>log2(x) and 2^x estimates accuracy</td>
<td>23 bit</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Feature</td>
<td>G80</td>
<td>SSE</td>
<td>IBM Altivec</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Precision</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
</tr>
<tr>
<td>Rounding modes for FADD and FMUL</td>
<td>Round to nearest and round to zero</td>
<td>All 4 IEEE, round to nearest, zero, inf, -inf</td>
<td>Round to nearest only</td>
</tr>
<tr>
<td>Denormal handling</td>
<td>Flush to zero</td>
<td>Supported, 1000’s of cycles</td>
<td>Supported, 1000’s of cycles</td>
</tr>
<tr>
<td>NaN support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Overflow and Infinity support</td>
<td>Yes, only clamps to max norm</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Flags</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Square root</td>
<td>Software only</td>
<td>Hardware</td>
<td>Software only</td>
</tr>
<tr>
<td>Division</td>
<td>Software only</td>
<td>Hardware</td>
<td>Software only</td>
</tr>
<tr>
<td>Reciprocal estimate accuracy</td>
<td>24 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td>Reciprocal sqrt estimate accuracy</td>
<td>23 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td>log2(x) and 2^x estimates accuracy</td>
<td>23 bit</td>
<td>No</td>
<td>12 bit</td>
</tr>
</tbody>
</table>
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
Applications & Sweet Spots

SIGGRAPH 2008

Beyond Programmable Shading: In Action
GPU Computing: Motivation

Beyond Programmable Shading: In Action
Beyond Programmable Shading: In Action

GPUs Are Fast & Getting Faster

Peak GFLOP/s

- NVIDIA GPU
- Intel CPU

Sep-02 | Jan-04 | May-05 | Oct-06 | Feb-08
GPU Computing Sweet Spots

Applications:

- High arithmetic intensity:
  Dense linear algebra, PDEs, n-body, finite difference, ...

- High bandwidth:
  Sequencing (virus scanning, genomics), sorting, database...

- Visual computing:
  Graphics, image processing, tomography, machine vision...
Beyond Programmable Shading: In Action

GPU Computing Example Markets

- Computational Geoscience
- Computational Chemistry
- Computational Medicine
- Computational Modeling
- Computational Science
- Computational Biology
- Computational Finance
- Image Processing
GPU Computing Sweet Spots

• From cluster to workstation
  - The “personal supercomputing” phase change
    • From lab to clinic
    • From machine room to engineer, grad student desks
    • From batch processing to interactive
    • From interactive to real-time

• GPU-enabled clusters
  - A 100x or better speedup changes the science
    • Solve at different scales
    • Direct brute-force methods may outperform cleverness
    • New bottlenecks may emerge
    • Approaches once inconceivable may become practical
### Applications - Condensed

- 3D image analysis
- Adaptive radiation therapy
- Acoustics
- Astronomy
- Audio
- Automobile vision
- Bioinformatics
- Biological simulation
- Broadcast
- Cellular automata
- Computational Fluid Dynamics
- Computer Vision
- Cryptography
- CT reconstruction
- Data Mining
- Digital cinema/projections
- Electromagnetic simulation
- Equity training
- Film
- Financial - lots of areas
- Languages
- GIS
- Holographics cinema
- Imaging (lots)
- Mathematics research
- Military (lots)
- Mine planning
- Molecular dynamics
- MRI reconstruction
- Multispectral imaging
- nbody
- Network processing
- Neural network
- Oceanographic research
- Optical inspection
- Particle physics
- Protein folding
- Quantum chemistry
- Ray tracing
- Radar
- Reservoir simulation
- Robotic vision/Al
- Robotic surgery
- Satellite data analysis
- Seismic imaging
- Surgery simulation
- Surveillance
- Ultrasound
- Video conferencing
- Telescope
- Video
- Visualization
- Wireless
- X-ray
Tesla HPC Platform

• HPC-oriented product line
  - C870: board (1 GPU)
  - D870: deskside unit (2 GPUs)
  - S870: 1u server unit (4 GPUs)
Beyond Programmable Shading: In Action

NVIDIA Decoder Ring

Architecture: TESLA
Chips: G80, G84, G86

SIGGRAPH 2008
New Applications

Real-time options implied volatility engine

Ultrasound imaging

Swaption volatility cube calculator

HOOMD Molecular Dynamics

Manifold 8 GIS

SDK: Mandelbrot, computer vision

Seismic migration

Also...

Image rotation/classification

Graphics processing toolbox

Microarray data analysis

Data parallel primitives

Astrophysics simulations
Computed Tomography:


X. Xue, A. Cheryauka, and D. Tubbs. Acceleration of fluoro-CT reconstruction for a mobile C-Arm on GPU and FPGA hardware: A simulation study. SPIE Medical Imaging 2006.


Magnetic Resonance Imaging:


Registration:

The Future of GPUs

• GPU Computing drives new applications
  - Reducing “Time to Discovery”
  - 100x Speedup changes science and research methods

• New applications drive the future of GPUs and GPU Computing
  - Drives new GPU capabilities
  - Drives hunger for more performance

• Some exciting new domains:
  - Vision, acoustic, and embedded applications
  - Large-scale simulation & physics
Parallel Computing’s Golden Age

• 1980s, early `90s: a golden age for parallel computing
  - Particularly data-parallel computing

• Architectures
  - Connection Machine, MasPar, Cray
  - True supercomputers: incredibly exotic, powerful, expensive

• Algorithms, languages, & programming models
  - Solved a wide variety of problems
  - Various parallel algorithmic models developed
  - P-RAM, V-RAM, circuit, hypercube, etc.
Parallel Computing’s Dark Age

• But...impact of data-parallel computing limited
  - Thinking Machines sold 7 CM-1s (100s of systems total)
  - MasPar sold ~200 systems

• Commercial and research activity subsided
  - Massively-parallel machines replaced by clusters of ever-more powerful commodity microprocessors
  - Beowulf, Legion, grid computing, ...

Massively parallel computing lost momentum to the inexorable advance of commodity technology
Beyond Programmable Shading: In Action
CPU/GPU Parallelism

- **Moore’s Law** gives you more and more transistors
  - What do you want to do with them?
  - **CPU strategy:** make the workload (one compute thread) run as fast as possible
    - Tactics:
      - Cache (area limiting)
      - Instruction/Data prefetch
      - Speculative execution
      - limited by “perimeter” - communication bandwidth
    ...then add task parallelism...multi-core
  - **GPU strategy:** make the workload (as many threads as possible) run as fast as possible
    - Tactics:
      - Parallelism (1000s of threads)
      - Pipelining
      - limited by “area” - compute capability
Beyond Programmable Shading: In Action

Hardware Implementation:
A Set of SIMD Multiprocessors

- The device is a set of multiprocessors
- Each multiprocessor is a set of 32-bit processors with a Single Instruction Multiple Data architecture
- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a warp
- The number of threads in a warp is the warp size
Streaming Multiprocessor (SM)

- **Processing elements**
  - 8 scalar thread processors (SP)
  - 32 GFLOPS peak at 1.35 GHz
  - 8192 32-bit registers (32KB)
    - ½ MB total register file space!
  - usual ops: float, int, branch, ...

- **Hardware multithreading**
  - up to 8 blocks resident at once
  - up to 768 active threads in total

- **16KB on-chip memory**
  - low latency storage
  - shared amongst threads of a block
  - supports thread communication
Hardware Implementation:
Memory Architecture

- The device has local device memory
  - Can be read and written by the host and by the multiprocessors

- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - on-chip shared memory
  - A read-only constant cache
  - A read-only texture cache
Each thread can:
- Read/write per-block on-chip shared memory
- Read per-grid cached constant memory
- Read/write non-cached device memory:
  - Per-grid global memory
  - Per-thread local memory
- Read cached texture memory