PARALLEL PROGRAMMING
MANY-CORE COMPUTING:
CUDA INTRODUCTION (3/5)
Schedule

1. Introduction, performance metrics & analysis
2. Many-core hardware, low-level optimizations
3. GPU hardware and Cuda class 1: basics
4. Cuda class 2: advanced
5. Case study: LOFAR telescope with many-cores
GPU hardware introduction
It's all about the memory
Integration into host system

- Typically PCI Express 2.0 x16
- Theoretical speed 8 GB/s
  - protocol overhead → 6 GB/s
- In reality: 4 – 6 GB/s
- V3.0 is coming soon
  - Double bandwidth
  - Less protocol overhead
Lessons from Graphics Pipeline

- Throughput is paramount
  - must paint every pixel within frame time
  - scalability

- Create, run, & retire lots of threads very rapidly
  - measured 14.8 billion thread/s on increment() kernel

- Use multithreading to hide latency
  - 1 stalled thread is OK if 100 are ready to run
CPU vs GPU

- Movie
- The Mythbusters
  - Jamie Hyneman & Adam Savage
  - Discovery Channel
- Appearance at NVIDIA’s NVISION 2008
Why is this different from a CPU?

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, etc.)
  - multithreading can hide latency => skip the big caches
  - share control logic across many threads
Chip area CPU vs GPU

CPU

GPU
Flynn’s taxonomy revisited

<table>
<thead>
<tr>
<th></th>
<th>Single Data</th>
<th>Multiple Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single instruction</td>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>Multiple instruction</td>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

GPUs don’t fit!
Key architectural Ideas

- Data parallel, like a vector machine
  - There, 1 thread issues parallel vector instructions

- SIMT (Single Instruction Multiple Thread) execution
  - Many threads work on a vector, each on a different element
  - They all execute the same instruction
  - HW automatically handles divergence

- Hardware multithreading
  - HW resource allocation & thread scheduling
  - HW relies on threads to hide latency
  - Context switching is (basically) free
ATI GPUs
Latest generation ATI

- Southern Islands
  - 1 chip: HD 7970
    - 2048 cores
    - 264 GB/sec memory bandwidth
    - 3.8 tflops single, 947 gflops double precision
    - Maximum power: 250 Watts
    - 399 euros!
  - 2 chips: HD 7990
    - 4096 cores, 7.6 tflops
- Comparison: entire 72-node DAS-4 VU cluster has 4.4 tflops
ATI programming models

- Low-level: CAL (assembly)
- High-level: Brook+
  - Originally developed at Stanford University
  - Streaming language
  - Performance is not great
- Now: OpenCL
GPU Hardware: NVIDIA

The way it's meant to be played™
Reading material

- Reader:
  - NVIDIA’s Next Generation CUDA Compute Architecture: Fermi

- Recommended further reading:
  - CUDA: Compute Unified Device Architecture
Fermi

- Consumer: GTX 480, 580
- GPGPU: Tesla C2050
  - More memory, ECC
  - 1.0 teraflop single
  - 515 megaflop double
- 16 streaming multiprocessors (SM)
  - GTX 580: 16
  - GTX 480: 15
  - C2050: 14
- SMs are independent
Fermi Streaming Multiprocessor (SM)

- 32 cores per SM (512 cores total)
- 64KB configurable L1 cache / shared memory
- 32,768 32-bit registers
CUDA Core Architecture

- Decoupled floating point and integer data paths
- Double precision throughput is 50% of single precision
- Integer operations optimized for extended precision
  - 64 bit and wider data element size
- Predication field for all instructions
- Fused-multiply-add
Memory Hierarchy

- Configurable L1 cache per SM
  - 16KB L1 cache / 48KB Shared
  - 48KB L1 cache / 16KB Shared

- Shared 768KB L2 cache
Multiple Memory Scopes

- **Per-thread private memory**
  - Each thread has its own local memory
  - Stacks, other private data
  - registers

- **Per-SM shared memory**
  - Small memory close to the processor, low latency

- **Device memory**
  - GPU frame buffer
  - Can be accessed by any thread in any SM
Atomic Operations

- Device memory is not coherent!

- Share data between streaming multiprocessors

- Read / Modify / Write

- Fermi increases atomic performance by 5x to 20x
  - Still, much slower than non-atomic access
ECC (Error-Correcting Code)

- All major internal memories are ECC protected
  - Register file, L1 cache, L2 cache

- DRAM protected by ECC (on Tesla only)

- ECC is a must have for many computing applications
Programming NVIDIA GPUs
NVIDIA GPUs become more generic

- Expand performance sweet spot of the GPU
  - Caching
  - Concurrent kernels
  - Double precision floating point
  - C++
- Full integration in modern software development environment
  - Debugging
  - Profiling
- Bring more users, more applications to the GPU
CUDA

- CUDA: Scalable parallel programming
  - C/C++ extensions
- Provide straightforward mapping onto hardware
  - Good fit to GPU architecture
  - Maps well to multi-core CPUs too
- Scale to 1000s of cores & 100,000s of threads
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization
Parallel Abstractions in CUDA

- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads
Hierarchy of concurrent threads

- Parallel kernels composed of many threads
  - All threads execute the same sequential program
  - Called the Kernel

- Threads are grouped into thread blocks
  - Threads in the same block can cooperate
  - Threads in different blocks cannot!

- All thread blocks are organized in a Grid

- Threads/blocks have unique IDs
Grids, Thread Blocks and Threads

<table>
<thead>
<tr>
<th>Thread Block 0, 0</th>
<th>Thread Block 0, 1</th>
<th>Thread Block 0, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
</tr>
<tr>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>0,2</td>
<td>0,2</td>
<td>0,2</td>
</tr>
<tr>
<td>0,3</td>
<td>0,3</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>1,0</td>
<td>1,0</td>
</tr>
<tr>
<td>1,1</td>
<td>1,1</td>
<td>1,1</td>
</tr>
<tr>
<td>1,2</td>
<td>1,2</td>
<td>1,2</td>
</tr>
<tr>
<td>2,3</td>
<td>2,3</td>
<td>2,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,0</td>
<td>2,0</td>
</tr>
<tr>
<td>2,1</td>
<td>2,1</td>
<td>2,1</td>
</tr>
<tr>
<td>2,2</td>
<td>2,2</td>
<td>2,2</td>
</tr>
<tr>
<td>2,3</td>
<td>2,3</td>
<td>2,3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread Block 1, 0</th>
<th>Thread Block 1, 1</th>
<th>Thread Block 1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
</tr>
<tr>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>0,2</td>
<td>0,2</td>
<td>0,2</td>
</tr>
<tr>
<td>0,3</td>
<td>0,3</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>1,0</td>
<td>1,0</td>
</tr>
<tr>
<td>1,1</td>
<td>1,1</td>
<td>1,1</td>
</tr>
<tr>
<td>1,2</td>
<td>1,2</td>
<td>1,2</td>
</tr>
<tr>
<td>2,3</td>
<td>2,3</td>
<td>2,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,0</td>
<td>2,0</td>
</tr>
<tr>
<td>2,1</td>
<td>2,1</td>
<td>2,1</td>
</tr>
<tr>
<td>2,2</td>
<td>2,2</td>
<td>2,2</td>
</tr>
<tr>
<td>2,3</td>
<td>2,3</td>
<td>2,3</td>
</tr>
</tbody>
</table>
CUDA Model of Parallelism

- CUDA virtualizes the physical hardware
  - Devices have
    - Different numbers of SMs
    - Different compute capabilities (Fermi = 2.0)
  - block is a virtualized streaming multiprocessor (threads, shared memory)
  - thread is a virtualized scalar processor (registers, PC, state)
- Scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks should be independent
Memory Spaces in CUDA

Host

Grid

Block (0, 0)
- Shared Memory
- Registers
- Thread (0, 0)
- Thread (1, 0)

Block (1, 0)
- Shared Memory
- Registers
- Thread (0, 0)
- Thread (1, 0)

Device Memory

Constant Memory
Device Memory

- CPU and GPU have separate memory spaces
  - Data is moved across PCI-e bus
  - Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions

- Pointers are just addresses
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa
Additional memories

- **Textures**
  - Read-only
  - Data resides in device memory
  - Different read path, includes specialized caches

- **Constant memory**
  - Data resides in device memory
  - Manually managed
  - Small (e.g., 64KB)
  - Use when all threads in a block read the same address
    - Serializes otherwise
Host (CPU) manages device (GPU) memory:

- `cudaMalloc(void **pointer, size_t nbytes)`
- `cudaMemset(void *pointer, int val, size_t count)`
- `cudaFree(void* pointer)`

```c
int n = 1024;
int nbytes = n * sizeof(int);
int* data = 0;
cudaMalloc(&data, nbytes);
cudaMemset(data, 0, nbytes);
cudaFree(data);
```
Data Copies

- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn’t start copying until previous CUDA calls complete

- `enum cudaMemcpyKind`
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

- Non-blocking copies are also available
  - DMA transfers, overlap computation and communication
## CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int var;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td>int array_var[10];</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>shared</strong> int shared_var;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int global_var;</td>
<td>device</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>constant</strong> int constant_var;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>
Philosophy: provide minimal set of extensions necessary

Function qualifiers:

```c
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

Execution configuration:
```
dim3 gridDim(100, 50); // 5000 thread blocks
dim3 blockDim(4, 8, 8); // 256 threads per block (1.3M total)
my_kernel <<< gridDim, blockDim >>> (...); // 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
Calculating the global thread index

“global” thread index:

\[
\text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x};
\]
Calculating the global thread index

```
\[ \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x} \]
```

\[
4 \times 2 + 1 = 9
\]
void vector_add(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    // initialization code here ... 

    // launch N/256 blocks of 256 threads each
    vector_add<<<<< N/256, 256 >>>>(deviceA, deviceB, deviceC);

    // cleanup code here ...
}

(can be in the same file)
int main(int argc, char** argv) {
    int size = N * sizeof(float);

    // allocate host memory
    hostA = malloc(size);
    hostB = malloc(size);
    hostC = malloc(size);

    // initialize A, B arrays here...

    // allocate device memory
    cudaMalloc(&deviceA, size);
    cudaMalloc(&deviceB, size);
    cudaMalloc(&deviceC, size);
Vector addition host code

```c
// transfer the data from the host to the device
cudaMemcpy(deviceA, hostA, size, cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, hostB, size, cudaMemcpyHostToDevice);

// launch N/256 blocks of 256 threads each
vector_add<<<N/256, 256>>>(deviceA, deviceB, deviceC);

// transfer the result back from the GPU to the host
cudaMemcpy(hostC, deviceC, size, cudaMemcpyDeviceToHost);
```