1. Introduction and Programming Basics (2-10-2014)
2. More programming and performance analysis (6-10-2014)
3. Advanced programming (9-10-2014)
4. Case study: LOFAR telescope with many-cores by Rob van Nieuwpoort (??)
Today

- Understand many-cores and their impact on HPC.
  - Top500

- Understand the hardware alternatives.
  - CPUs
  - GPUs
  - Xeon Phi
  (+ more – check backup slides)

- Basic programming
  - Case-study
4

Introduction to many-cores

Moore’s law
Many-cores in real-life
What are many-cores?

- From Wikipedia: “A many-core processor is a multi-core processor in which the number of cores is **large enough** that traditional multi-processor techniques are no longer efficient — largely because of issues with **congestion** in supplying **instructions** and **data** to the many processors.”

- In this course:
  - **Multi-core/many-core CPUs**
    - Regular CPUs are “multi-cores” (4-12 cores)
    - Xeon Phi is a “many-core” CPU/accelerator (60-70 cores)
  - **Graphics Processing Units (GPUs)**
    - Different kind of cores (100x cores)
Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase....” Electronics Magazine 1965
Transistor Counts

![Graph showing the increase in transistor counts over the years. The x-axis represents the year from 1970 to 2015, and the y-axis represents the number of transistors from 1000 to 10,000,000,000.](image-url)
Revolution in Processors

- Chip density is continuing to increase about 2x every 2 years
- BUT
  - Clock speed is not
  - Performance per cycle is not
  - Power is not
New ways to use transistors

- Parallelism on-chip: multi-core processors
  - Transformed in many-core processors.

- “Multicore revolution”
  - Every machine is a parallel machine.
  - Accelerators start playing an important role.
    - Specialized
    - Energy efficient
    - Used on demand

- Can all applications use this parallelism?
- Can we program all these architectures efficiently?
  - Performance? Productivity?
## Top500 – June 2014

- State of the art in HPC (top500.org)
- Trial for all new HPC architectures

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National University of Defense Technology, China</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3120000</td>
<td>33862.7</td>
<td>54902.4</td>
<td>17808</td>
</tr>
<tr>
<td></td>
<td></td>
<td>195 cores/node!</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DOE/SC/Oak Ridge National Laboratory, United States</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>560640</td>
<td>17590.0</td>
<td>27112.5</td>
<td>8209</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accelerated!</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DOE/NNSA/LLNL, United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM</td>
<td>1572864</td>
<td>17173.2</td>
<td>20132.7</td>
<td>7890</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science (AICS), Japan</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu</td>
<td>705024</td>
<td>10510.0</td>
<td>11280.4</td>
<td>12660</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Argonne National Laboratory, United States</td>
<td>Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM</td>
<td>786432</td>
<td>8586.6</td>
<td>10066.3</td>
<td>3945</td>
</tr>
</tbody>
</table>
Top500 – accelerators

- Relatively low numbers
- High performance impact

![Pie charts showing Accelerator/CP Family System Share and Performance Share.](image-url)
China's Tianhe-2

#1 in Top500 – June 2014

54.902 PFLOPs peak
33.862 PFLOPs max

16,000 nodes = 16,000 x (2 x Xeon IvyBridge + 3 x Xeon Phi)
= 3,120,000 cores (=> 195 cores/node)
GPU vs. CPU performance

Theoretical GFLOP/s

1 GFLOPs = 10^9 ops / second
GPU vs. CPU performance

1 GB/s = 8 \times 10^9 \text{ bits / second}
Why do we use many-cores?

- Performance
  - Large scale parallelism

- Power Efficiency
  - Use transistors more efficiently

- Price (GPUs)
  - Game market is huge, bigger than Hollywood
    - Gaming pays for our HPC needs!
  - Mass production, economy of scale

- Prestige
  - Reach ExaFLOP by 2019/2022 …
Multi-core CPUs
Multi-core CPUs

- **Architecture**
  - Few large cores
  - (Integrated GPUs)
  - Vector units
    - Streaming SIMD Extensions (SSE)
    - Advanced Vector Extensions (AVX)
  - Stand-alone

- **Memory**
  - Shared, multi-layered
  - Per-core caches + shared caches

- **Programming**
  - Multi-threading
  - OS Scheduler
Parallelism

- Core-level parallelism \sim task/data parallelism (coarse)
  - 4-12 of powerful cores
    - Hardware hyperthreading (2x)
  - Local caches
  - Symmetrical or asymmetrical threading model
  - Implemented by programmer

- SIMD parallelism = data parallelism (fine)
  - 4-SP/2-DP floating point operations per second
    - 256-bit vectors
  - Run same instruction on different data
  - Sensitive to divergence
    - NOT the same instruction \Rightarrow performance loss
  - Implemented by programmer OR compiler
Programming models

- Pthreads + intrinsics
- TBB – Thread building blocks
  - Threading library
- OpenCL
  - To be discussed …
- OpenMP
  - Traditional parallel library
  - High-level, pragma-based
- Cilk
  - Simple divide-and-conquer model
Xeon Phi
Larrabee

- GPU based on x86 architecture
  - Hardware multithreading
  - Wide SIMD
- Achieved 1 tflop sustained application performance (SC09)
- Canceled in Dec 2009, re-targeted to HPC market
Intel Xeon Phi (=MIC)

- First product: “Knights corner”
  - Accelerator or stand-alone
  - ±60 Pentium1-like cores
  - 512-bit SIMD

- Memory
  - L1-cache per core (32KB I$ + 32KB D$)
  - Very large unified L2-cache (512KB/core, ~30MB/chip)
  - At least 8GB of GDDR5

- Programming
  - Multi-threading
    - Traditional models: OpenMP, MPI, Cilk, TBB, parallel libraries + OpenCL
  - OS thread scheduler
    - User-control via affinity
Architecture of Xeon Phi
Parallelism

- 3 operation modes
  - Stand-alone
  - Offload
  - Hybrid

- Core-level parallelism \(\sim\) task parallelism (coarse)
  - \(\sim60\) cores, 4x hyperthreading

- SIMD parallelism = data parallelism (fine)
  - Fine-grained parallelism
  - 16 SP-Flop, 16 int-ops, 8 DP-Flop / cycle
  - AVX-512 extensions
    - No support for MMX, SSE \(\Rightarrow\) not backward compatible
GPUs
Current generation: NVIDIA Kepler
- 7.1B transistors
- More cores, more parallelism, more performance
GPGPU History

- Use Graphics primitives for HPC
  - Ikonas [England 1978]
  - Pixel Machine [Potmesil & Hoffert 1989]
  - Pixel-Planes 5 [Rhoades, et al. 1992]

- Programmable shaders, around 1998
  - DirectX / OpenGL
  - Map application onto graphics domain!

- GPGPU
  - Brook (2004), Cuda (2007), OpenCL (Dec 2008), ...
GPGPU History

Use graphics primitives for HPC

Ikonas [England 1978]

Pixel-planes 5 [Rhoades, et al. 1992]

1978

1989

1992

1998

Graphics Programming on GPU (GPGPU)

Brook (2004)

OpenCL (December 2008)

2004

2007

2008

Pixel machine [Potmesil & Hoffert 1989]

DirectX/OpenGL Map application onto graphics domain

Programmable shaders, around 1998

CUDA (2007)
Another GPGPU history

Growth of GPU Computing

- **100M**: CUDA-Capable GPUs
- **150K**: CUDA Downloads
- **1**: Supercomputer
- **60**: University Courses
- **4,000**: Academic Papers
- **430M**: CUDA-Capable GPUs
- **1.6M**: CUDA Downloads
- **50**: Supercomputers
- **640**: University Courses
- **37,000**: Academic Papers

2008  2013
GPGPU @ NVIDIA

GPU Roadmap

- **Fermi** (FP64)
- **Maxwell** (Unified Virtual Memory)
- **Kepler** (Dynamic Parallelism)
- **Volta** (Stacked DRAM)

- **Tesla** (CUDA)

Horizontal axis: 2008 to 2014

Vertical axis: DP GFLOPS per Watt

Values: 32, 16, 8, 4, 2, 1, 0.5
GPUs @ AMD

AMD Radeon Graphics Roadmap

<table>
<thead>
<tr>
<th>Performance</th>
<th>Graphics</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>7200, $449</td>
<td>HD 7970 GHz Edition</td>
<td></td>
</tr>
<tr>
<td>6700, $399</td>
<td>HD 7970</td>
<td></td>
</tr>
<tr>
<td>6150, $299</td>
<td>HD 7950</td>
<td></td>
</tr>
<tr>
<td>5000, $229</td>
<td>HD 7870 GHz Edition</td>
<td></td>
</tr>
<tr>
<td>4200, $189</td>
<td>HD 7850</td>
<td></td>
</tr>
<tr>
<td>2750, $119</td>
<td>HD 7770 GHz Edition</td>
<td></td>
</tr>
<tr>
<td>2050, $99</td>
<td>HD 7750</td>
<td></td>
</tr>
<tr>
<td>6300, $499</td>
<td>GTX 680</td>
<td></td>
</tr>
<tr>
<td>5650, $399</td>
<td>GTX 670</td>
<td></td>
</tr>
<tr>
<td>5000, $299</td>
<td>GTX 660 Ti</td>
<td></td>
</tr>
<tr>
<td>4350, $229</td>
<td>GTX 660</td>
<td></td>
</tr>
<tr>
<td>2900, $149</td>
<td>GTX 650 Ti</td>
<td></td>
</tr>
<tr>
<td>2000, $109</td>
<td>GTX 650</td>
<td></td>
</tr>
<tr>
<td>1200, $89</td>
<td>GT 640</td>
<td></td>
</tr>
</tbody>
</table>

3DMark Fire Strike
Performance, Price

30 | Never Settle | January 2013 | UNDER EMBARGO UNTIL FEB 4, 2013 @ 12:01AM EST
GPUs @ ARM

ARM Mali
“Graphics Performance Leadership”

- Innovative Tri-pipe architecture for performance and flexibility
- GPGPU computing with OpenCL 1.1 up to 68GFLOPS
- State of the art bandwidth reduction
- DirectX11 and next generation Khronos graphics standards up to 2GPix/s

Mali-400 MP
- World’s first multicore embedded GPU
- High-performance graphics to beyond 1080p
- Leading power efficiency and reduced bandwidth

Mali-T604

Mali-300
- Ideal configuration for mid-range use-cases
- Efficient energy and bandwidth usage

Mali-200
- Entry level
- Leading anti-aliasing for superior image quality

All Mali GPUs support the Khronos APIs OpenVG 1.1 and OpenGL ES 2.0 plus roadmap
(NVIDIA) GPUs

- **Architecture**
  - Many (100s) slim cores
  - Sets of (32 or 192) cores grouped into “multiprocessors” with shared memory
    - $SM(X) =$ stream multiprocessors
  - Work as accelerators

- **Memory**
  - Shared L2 cache
  - Per-core caches + shared caches
  - Off-chip global memory

- **Programming**
  - Symmetric multi-threading
  - Hardware scheduler
NVIDIA’s Fermi GPU Architecture
Parallelism

- **Data parallelism (fine-grain)**
  - Restricted forms of task parallelism possible with newest generation of NVIDIA GPUs

- **SIMT (Single Instruction Multiple Thread) execution**
  - Many threads execute concurrently
    - Same instruction
    - Different data elements
    - HW automatically handles divergence
  - Not same as SIMD because of multiple register sets, addresses, and flow paths*

- **Hardware multithreading**
  - HW resource allocation & thread scheduling
    - Excess of threads to hide latency
    - Context switching is (basically) free

Integration into host system

- Typically PCI Express 2.0
- Theoretical speed 8 GB/s
  - Effective $\leq$ 6 GB/s
  - In reality: 4 – 6 GB/s
- V3.0 recently available
  - Double bandwidth
  - Less protocol overhead
CPU vs. GPU

CPU

GPU
Why so different?

- Different goals produce different designs!
  - CPU must be good at everything
  - GPUs focus on massive parallelism
    - Less flexible, more specialized

- 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 $\Rightarrow$ lots of resources (registers, etc.)
  - multithreading can hide latency $\Rightarrow$ no big caches
  - share control logic across many threads
CPU vs. GPU

- Movie
- The Mythbusters
  - Jamie Hyneman & Adam Savage
  - Discovery Channel
- Appearance at NVIDIA’s NVISION 2008
Programming many-cores
Programming many-cores

= parallel programming:
  - Choose/design algorithm
  - Parallelize algorithm
    - Expose enough layers of parallelism
    - Minimize communication, synchronization, dependencies
    - Overlap computation and communication
  - Implement parallel algorithm
    - Choose parallel programming model
    - (?) Choose many-core platform
  - Tune/optimize application
    - Understand performance bottlenecks & expectations
    - Apply platform specific optimizations
    - (?) Apply application & data specific optimizations
Programming GPUs in CUDA

Practical aspects
A first example
CUDA

- CUDA: Scalable parallel programming
  - C/C++ extensions
    - Other wrappers exist

- Straightforward mapping onto hardware
  - Hierarchy of threads (to map to cores)
    - Configurable at logical level
  - Various memory spaces (to map to physical spaces)
    - Usable via variable scopes

- Scale to 1000s of cores & 100,000s of threads
  - GPU threads are lightweight
  - GPUs need 1000s of threads for full utilization
Hierarchy of threads

- Each thread executes the kernel code
  - One thread runs on one CUDA core
- Threads are logically grouped into thread blocks
  - Threads in the same block can cooperate
  - Threads in different blocks cannot cooperate
- All thread blocks are logically organized in a Grid
  - 1D or 2D or 3D
  - Threads and blocks have unique IDs
- A grid specifies in how many instances a kernel is being run
Hierarchy of threads
Grids, Thread Blocks and Threads

Grid

Thread Block 0, 0

Thread Block 0, 1

Thread Block 0, 2

Thread Block 1, 0

Thread Block 1, 1

Thread Block 1, 2
Launch kernel \((12 \times 6 = 72 \text{ instances})\)

```cpp
myKernel<<<numBlocks,threadsPerBlock>>>(...);
```

- `dim3 threadsPerBlock(3,4);`
  - `threadsPerBlock.x = 3`
  - `threadsPerBlock.y = 4`
  - Each thread:
    - `(threadIdx.x, threadIdx.y)`

- `dim3 numBlocks(2,3);`
  - `blockDim.x = 2`
  - `blockDim.y = 3`
  - Each block:
    - `(blockIdx.x,blockIdx.y)`
Multiple Device 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
All 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
Using CUDA

- Two parts of the code:
  - Device code = GPU code = kernel(s)
    - Sequential program
    - Write for 1 thread, execute for all
  - Host code = CPU code
    - Instantiate grid + run the kernel
    - Memory allocation, management, deallocation
    - C/C++/Java/Python/…

- Host-device communication
  - Explicit / implicit via PCI/e
  - Minimum: data input/output
Processing flow

All this happens from the host code.

Kernel runs here

Processing flow on CUDA

Image courtesy of Wikipedia
### 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:
```c
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:
```c
dim3 gridDim;   // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index
dim3 threadIdx; // Thread index
void syncthreads();  // Thread synchronization
First CUDA program

- Determine mapping of operations and data to threads
- Write kernel(s)
  - Sequential code
  - Written per-thread
- Determine block geometry
  - Threads per block, blocks per grid
  - Number of grids (≥ number of kernels)
- Write host code
  - Memory initialization and copying to device
  - Kernel(s) launch(es)
  - Results copying to host
- Optimize the kernels
void vector_add(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}
How do we parallelize this?

- What does each thread compute?
  - One addition per thread
  - Each thread deals with *different* elements
  - How do we know which element?
    - Compute a mapping of the grid to the data
      - Any mapping will do!
Processing flow

All this happens from the host code.

Image courtesy of Wikipedia
// 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 = ?
    C[i] = A[i] + B[i];
}
Calculating the global thread index

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

- “global” thread index:
  
  \[ \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x} \; ; \]

- Example:
  
  \[ 4 \times 2 + 1 = 9 \]
// 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];
}
All this happens from the host code.
// 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 ...
   N = 5120;
   // 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)
// 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];
}

What if \( N = 5000 \)?

int main() {
    // initialization code here ...
    N = 5000;
    // launch \( N/256 \) blocks of 256 threads each
    vector_add<<<N/256, 256>>>(deviceA, deviceB, deviceC);
    // cleanup code here ...
}
// 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;
    if (i<N) C[i] = A[i] + B[i];
}

What if N = 5000?

int main() {
    // initialization code here ...
    N = 5000;
    // launch N/256 blocks of 256 threads each
    vector_add<<< N/256+1,256 >>>(deviceA, deviceB, deviceC);
    // cleanup code here ...
}
Host (CPU) manages device (GPU) memory:

- `cudaMalloc(void **pointer, size_t nbytes)`
- `cudaMemcpy(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);
cudaMemcpy(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`

- Non-blocking copies are also available
  - DMA transfers, overlap computation and communication
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 add: Host
// 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);
}
Summary

- Determine mapping of operations and data to threads
- Write kernel(s)
  - Sequential code
  - Written per-thread
- Determine block geometry
  - Threads per block, blocks per grid
  - Number of grids (>= number of kernels)
- Write host code
  - Memory initialization and copying to device
  - Kernel(s) launch(es)
  - Results copying to host
- Optimize the kernels
Remarks, comments, ideas are all welcome!

A.L.Varbanescu@uva.nl
Extra Slides: Intel SCC
Intel Single-chip Cloud Computer

- **Architecture**
  - Tile-based many-core (48 cores)
  - A tile is a dual-core
  - Stand-alone

- **Memory**
  - Per-core and per-tile
  - Shared off-chip

- **Programming**
  - Multi-processing with message passing
  - User-controlled mapping/scheduling

- **Gain performance ...**
  - Coarse-grain parallelism
  - Multi-application workloads (cluster-like)
Intel Single-chip Cloud Computer
Intel SCC Tile

- 2 cores
- 16K L1 cache per core
- 256K L2 per core
- 8K Message passing buffer
- On-chip network router
Extra slides: 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 5870 architecture overview
Each of the 20 SIMD engines has:
- 16 thread processors x 5 stream cores = 80 scalar stream processing units
- 20 * 16 * 5 = 1600 cores total
- 32KB Local Data Share
- its own control logic and runs from a shared set of threads
- a dedicated fetch unit with 8KB L1 cache
- a 64KB global data share to communicate with other SIMD engines
Each thread processor includes:
- 4 stream cores + 1 special function stream core
- general purpose registers
- FMA in a single clock
EDC (Error Detection Code)
- CRC Checks on Data Transfers for Improved Reliability at High Clock Speeds

Bandwidths
- Up to 1 TB/sec L1 texture fetch bandwidth
- Up to 435 GB/sec between L1 & L2
- 153.6 GB/s to device memory
- PCI-e 2.0, 16x: 8GB/s to main memory
Unified Load/Store Addressing

Non-unified Address Space

- Local
  - \*p_local

- Shared
  - \*p_shared

- Device
  - 0
  - \*p_device
  - 32-bit

Unified Address Space

- Local
- Shared
- Device
  - 0
  - \*p
  - 40-bit
Two 6-core processors on a single chip

Up to four of these chips in a single compute node
- 48 cores in total

Non-uniform memory access (NUMA)
- Per-core cache
- Per-chip cache
- Local memory
- Remote memory (hypertransport)
AMD Magny-Cours

Magny-Cours MCM (2 nodes)

- 2.3 GHz Istanbul Core 0
  - 64KB L1D Cache
  - 512KB L2 Cache
- 2.3 GHz Istanbul Core 5
  - 64KB L1D Cache
  - 512KB L2 Cache
- 2.3 GHz Istanbul Core 6
  - 64KB L1D Cache
  - 512KB L2 Cache
- 2.3 GHz Istanbul Core 11
  - 64KB L1D Cache
  - 512KB L2 Cache

- 6MB L3 Cache
- Hypertransport 3.1
- DDR3 Memory Controllers

- 3x1B @ 6.4GT/s
- 2x8B @ 1.33GT/s
- 5x1B @ 6.4GT/s
- 3x1B @ 6.4GT/s
- 2x8B @ 1.33GT/s
AMD Magny-Cours
AWARI on the Magny-Cours

- **DAS-2 (1999)**
  - 51 hours
  - 72 machines / 144 cores
  - 72 GB RAM in total
  - 1.4 TB disk in total

- **Magny-Cours (2011)**
  - 45 hours
  - 1 machine, 48 cores
  - 128 GB RAM in 1 machine
  - 4.5 TB disk in 1 machine
  - Less than 12 hours with new algorithm (needs more RAM)
Extra Slides: Cell Broadband Engine
Cell/B.E.

- **Architecture**
  - Heterogeneous
  - 1 PowerPC (PPE)
  - 8 vector-processors (SPEs)

- **Programming**
  - User-controlled scheduling
  - 6 levels of parallelism, all under user control
  - Fine- and coarse-grain parallelism
Cell/B.E. memory

- “Normal” main memory
  - PPE: normal read / write
  - SPEs: Asynchronous manual transfers
    - Direct Memory Access (DMA)

- Per-core fast memory: the Local Store (LS)
  - Application-managed cache
  - 256 KB

- 128 x 128 bit vector registers
Roadrunner (IBM)

- Los Alamos National Laboratory
- #1 of top500 June 2008 – November 2009
- Now #19
- 122,400 cores, 1.4 petaflops
- First petaflop system
- PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz
The Cell’s vector instructions

- Differences with SSE
  - SPEs execute *only* vector instructions
  - More advanced shuffling
  - Not 16, but 128 registers!
  - Fused Multiply Add support
FMA instruction

Multiply-Add (MAD): $D = A \times B + C$

$$A \times B$$

= Product \hspace{10px} (truncate digits)

+ C

= D

Fused Multiply-Add (FMA): $D = A \times B + C$

$$A \times B$$

= Product \hspace{10px} (retain all digits)

+ C

= D \hspace{10px} (no loss of precision)
Cell Programming models

- IBM Cell SDK
- C + MPI
- OpenCL
- Many models from academia...
Cell SDK

- Threads, but only on the PPE
- Distributed memory
- Local stores = application-managed cache!
- DMA transfers
- Signaling and mailboxes
- Vectorization
Direct Memory Access (DMA)

- **Start asynchronous DMA**
  - `mfc_get (local store space, main mem address, #bytes, tag);`

- **Wait for DMA to finish**
  - `mfc_write_tag_mask(tag);`
  - `mfc_read_tag_status_all();`

- **DMA lists**

- **Overlap communication with useful work**
  - **Double buffering**
float vectorSum(int size, float* vector) {
    float result = 0.0;
    for(int i=0; i<size; i++) {
        result += vector[i];
    }
    return result;
}
Parallelization strategy

- Partition problem into 8 pieces
  - (Assuming a chunk fits in the Local Store)

- PPE starts 8 SPE threads

- Each SPE processes 1 piece
  - Has to load data from PPE with DMA

- PPE adds the 8 sub-results
float vectorSum(int size, float* PPEVector) {
    float result = 0.0;

    int chunkSize = size / NR_SPES; // Partition the data.
    float localBuffer[chunkSize];   // Allocate a buffer in
                                    // my private local store.

    int tag = 42;

    // Points to my chunk in PPE memory.
    float* myRemoteChunk = PPEVector + chunkSize * MY_SPE_NUMBER;
// Copy the input data from the PPE.
mfc_get(localBuffer, myRemoteChunk, chunkSize*4, tag);
mfc_write_tag_mask(tag);
mfc_read_tag_status_all();

// The real work.
for (int i=0; i<chunkSize; i++) {
    result += localBuffer[i];
}
return result;
}
Can we optimize this strategy?
Can we optimize this strategy?

- Vectorization

- Overlap communication and computation
  - Double buffering
  - Strategy:
    - Split in more chunks than SPEs
    - Let each SPE download the next chunk while processing the current chunk
float vectorSum(float* PPEVector, int size, int nrChunks) {
    float result = 0.0;
    int chunkSize = size / nrChunks;
    int chunksPerSPE = nrChunks / NR_SPES;
    int firstChunk = MY_SPE_NUMBER * chunksPerSPE;
    int lastChunk = firstChunk + nrChunks;

    // Allocate two buffers in my private local store.
    float localBuffer[2][chunkSize];
    int currentBuffer = 0;

    // Start asynchronous DMA of first chunk.
    float* myRemoteChunk = PPEVector + firstChunk * chunkSize;
    mfc_get(localBuffer[currentBuffer], myRemoteChunk, chunkSize, currentBuffer);
for (int chunk = firstChunk; chunk < lastChunk; chunk++) {
    // Prefetch next chunk asynchronously.
    if (chunk != lastChunk - 1) {
        float* nextRemoteChunk = PPEVector + (chunk + 1) * chunkSize;
        mfc_get(localBuffer[!currentBuffer], nextRemoteChunk,
                chunkSize, !currentBuffer);
    }
    // Wait for of current buffer DMA to finish.
    mfc_write_tag_mask(currentBuffer); mfc_read_tag_status_all();
    // The real work.
    for (int i = 0; i < chunkSize; i++)
        result += localBuffer[currentBuffer][i];

    currentBuffer = !currentBuffer;
}
return result;
Double and triple buffering

- Read-only data
  - Double buffering

- Read-write data
  - Triple buffering!
    - Work buffer
    - Prefetch buffer, asynchronous download
    - Finished buffer, asynchronous upload

- General technique
  - On-chip networks
  - GPUs (PCI-e)
  - MPI (cluster)
  - …