Experiencing Various Massively Parallel Architectures and Programming Models for Data-Intensive Applications

Hongliang Gao, Martin Dimitrov, Jingfei Kong, Huiyang Zhou

School of Electrical Engineering and Computer Science
University of Central Florida
Background

• Future applications are expected to be highly data-intensive
  – Scientific computing
  – Business information processing
  – Entertainment computing
  – Etc.

• Common requirements: Intensive computation on very large data sets

• Massively parallel processors like graphics processors are promising
  – High floating-point computation capability and memory bandwidth
  – Programming support for general-purpose computing
A Challenge to the Existing Curriculum

- **Parallel Architecture Course**
  - Hardware features for general-purpose multiprocessor such as cache coherence, memory consistency, interconnect, etc., are either not necessary or too costly in processors designed for data-intensive applications

- **Parallel Algorithm & Programming Course**
  - High-level programming algorithms and concepts, which are not sufficient to take full advantages of these massively parallel processors

- **Needs to understand both the architectural features and the programming models**
  - Select right target processors
  - Reason about the performance
  - Perform program optimization
Multi-core/Many-core Architecture and Programming

• Three different processor models and their programming support
  – Nvidia G80 and CUDA
  – AMD/ATI RV670 and Brook+
  – Cell processors and their SPE/PPE code development

• Target audience: graduate students with some background on computer architecture and/or parallel programming

• Co-developed with AMD/ATI researchers

• [http://csl.cs.ucf.edu/courses/CDA6938](http://csl.cs.ucf.edu/courses/CDA6938)
Outline

• Introduction
• The course: Multi-core/Many-core Architecture and Programming
  – Course description
  – Programming Assignments
  – Term Projects
• Results
  – Interesting observations from programming assignments
  – Findings on performance optimization
  – Project results
GPU Architectures and Programming Support

- AMD/ATI streaming processors and Nvidia G80 processors
- Both are massively parallel processors
- Both use the Single-Program Multiple-Data (SPMD) programming model
- The threads in the same wavefront/warp are executed in the Single-Instruction Multiple-Data (SIMD) mode
## Architectural Features

<table>
<thead>
<tr>
<th></th>
<th>G80 (Geforce 8800)</th>
<th>RV670 (AMD Radeon HD 3870)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stream Processors</strong></td>
<td>128</td>
<td>320</td>
</tr>
<tr>
<td><strong>StreamProc Clk</strong></td>
<td>1350 Mhz</td>
<td>775Mhz</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>345.6 GFLOPS (no double precision FP support) 128 units @ 1350Mhz *2 for muladd</td>
<td>496 GFLOPS (99.2 GFLOPS for double-precision FP numbers)</td>
</tr>
<tr>
<td><strong>Memory BW</strong></td>
<td>384 bit GDDR3 @ 900MHz, 86.4 GB/s</td>
<td>256 bit <a href="mailto:GDDR4@1.13GHz">GDDR4@1.13GHz</a>/pin, 72 GB/s</td>
</tr>
<tr>
<td><strong>Memory size</strong></td>
<td>768 MB</td>
<td>512MB</td>
</tr>
<tr>
<td><strong>Thread Hierarchy</strong></td>
<td>16 Stream multi-processors (SM), 8 streaming processors (SP) per SM, 2 SMs share 1 Texture subsystem</td>
<td>4 clusters, 16 x 5 cores per cluster, each cluster time-multiplex 1 Texture subsystem</td>
</tr>
</tbody>
</table>
G80 Architecture Overview

8 TPCs

Streaming Processor Array

Texture Processor Cluster

 Streaming Multiprocessor

Instruction Fetch/Dispatch

Shared Memory

Instruction L1

Data L1

TEX

SM

SM

SP

SP

SP

SFU

SFU
ATI Streaming Processors
**Parallelism**

<table>
<thead>
<tr>
<th></th>
<th>G80 (Geforce 8800)</th>
<th>RV670 (AMD Radeon HD 3870)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread Hierarchy</strong></td>
<td>16 Stream multiprocessors (SM), 8 streaming processors (SP) per SM,</td>
<td>4 clusters, 16 x 5 cores per cluster</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Max. # threads</strong></td>
<td>768 per SM * 16 SM</td>
<td>64 per wavefront * 192 wave fronts</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Max. # active threads in execution</strong></td>
<td>32 (warp size) per SM * 16 SM. Each warp takes 4 cycles to issue</td>
<td>64 (wavefront size) per cluster * 4 clusters. Each wavefront takes 4 cycles to issue</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction-level parallelism</strong></td>
<td>Scalar operations for each thread</td>
<td>5-way VLIW for each thread</td>
</tr>
</tbody>
</table>


## Memory Hierarchy

<table>
<thead>
<tr>
<th></th>
<th>G80 (Geforce 8800)</th>
<th>RV670 (AMD Radeon HD 3870)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register File (32-bit registers)</td>
<td>512 kB = 32kB per SM * 16 SM; 8K registers per SM; 1K register per SP</td>
<td>1MB = 256kB per cluster * 4 cluster; 64K registers per cluster; 1K register per core</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>256 kB = 16kB per SM * 16 SM</td>
<td>N/A</td>
</tr>
<tr>
<td>R/W cache</td>
<td>N/A</td>
<td>A cache (size not disclosed)</td>
</tr>
<tr>
<td>Local/Global/Texture memory</td>
<td>Device Mem size</td>
<td>Device Mem Size</td>
</tr>
<tr>
<td>Constant Cache</td>
<td>8KB per SM, 128KB in total</td>
<td>L1 (size not disclosed)</td>
</tr>
</tbody>
</table>
# Programming Support

<table>
<thead>
<tr>
<th></th>
<th>G80 (Geforce 8800)</th>
<th>RV670 (AMD Radeon HD 3870)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming model</td>
<td>SPMD</td>
<td>SPMD</td>
</tr>
<tr>
<td>Programming Language</td>
<td>C/C++</td>
<td>C</td>
</tr>
<tr>
<td>Intermediate Language</td>
<td>PTX</td>
<td>AMD/ATI IL</td>
</tr>
<tr>
<td>Assembly-level analysis</td>
<td>Decuda</td>
<td>GPU ShaderAnalyzer</td>
</tr>
<tr>
<td>Thread management</td>
<td>Thread hierarchy</td>
<td>Streaming model</td>
</tr>
</tbody>
</table>
Cell Broadband Engine Architecture and Programming

• Heterogeneous multi-core architecture

• Much less thread-level parallelism than GPUs

• Significantly different programming focus from GPUs:
  – Task/data decomposition
  – Explicit control/data transmission using mailboxes/DMA
  – Vector programming
Programming Assignment

• matrix multiplication and 2-D image convolution

• Code development process
  - CPU code first
  - GPU/Cell code development
  - Performance analysis & optimization
Term Projects

• Select applications with rich data-level parallelism
• Select target processor platform
• Code development
• Presentation and technical report
Results: Programming Assignments

• Started with *un-optimized* CPU code
  – Matrix multiplication (a product of two 2kx2k matrices): \(30M\ FLOPS\)
  – Convolution (a 2kx2k matrix convolved with a 5x5 kernel): \(205MFLOPS\)

• Single-precision is used

• Double-precision is supported in AMD/ATI GPUs and Brook+ (1.0 beta)
  – ~50% of the throughput of single-precision FP numbers
Results: Matrix multiplication (2k x 2k matrices)

<table>
<thead>
<tr>
<th></th>
<th>Nvidia 8800 GTX</th>
<th>AMD/ATI HD3870</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. # of lines in the code</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>Median. # of lines in the code</td>
<td>37</td>
<td>21</td>
</tr>
<tr>
<td>Max. # of lines in the code</td>
<td>140</td>
<td>35</td>
</tr>
<tr>
<td>Min. Throughput</td>
<td>13.8 GFLOPS</td>
<td>8.3 GFLOPS</td>
</tr>
<tr>
<td>Median. Throughput</td>
<td>67 GFLOPS</td>
<td>18.3 GFLOPS</td>
</tr>
<tr>
<td>Max. Throughput</td>
<td>149 GFLOPS</td>
<td>43 GFLOPS</td>
</tr>
</tbody>
</table>

Note: results should *not* be used to compare the performance of the two GPUs. Reasons: (1) initial effort of inexperienced students. Both math libraries from Nvidia and AMD/ATI achieve over 100 GFLOPS. (2) Not all architectural features are exposed. AMD/ATI CAL SDK (intermediate level) has much higher throughput (213 GFLOPS) than the Brook+ version.
Results: Image Convolution (a 2k x 2k matrix with a 5x5 kernel)

<table>
<thead>
<tr>
<th></th>
<th>Nvidia 8800 GTX</th>
<th>AMD/ATI HD3870</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. # of lines in the code</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>Median. # of lines in the code</td>
<td>43</td>
<td>34</td>
</tr>
<tr>
<td>Max. # of lines in the code</td>
<td>115</td>
<td>88</td>
</tr>
<tr>
<td>Min. Throughput</td>
<td>0.27 GFLOPS</td>
<td>0.42 GFLOPS</td>
</tr>
<tr>
<td>Median. Throughput</td>
<td>6.08 GFLOPS</td>
<td>1.2 GFLOPS</td>
</tr>
<tr>
<td>Max. Throughput</td>
<td>18 GFLOPS</td>
<td>2.2 GFLOPS</td>
</tr>
</tbody>
</table>
Observations

• SPMD programming model is easy to grasp. High performance gains from GPU code
  – Matrix multiplication: 226x (median) using GTX8800 and 40x (median) using HD3870
  – Convolution: 30x (median) using GTX8800 and 6x (median) using HD3870
    • (CPU-GPU transmission latency dominates the overall performance)

• The performance gains justify the effort to port the code to GPUs
  – The GPU performance is compared to un-optimized CPU code.
  – Same effort spent on optimizing CPU code is unlikely to produce improvements in a similar order of magnitude.
Observations

• Low coding complexity if measured by number of lines of code

• Best designs have 2x to 3x speedups over the median performance
  – Our matrix multiplication on Nvidia 8800 GTX processors (149 GFLOPS) vs. Nvidia CUBLAS library (100 GFLOPS)
An Important Lesson on Performance Optimization

• Performance analysis tools are critical, especially those with machine assembly-level information.
  - AMD Shader analyzer
  - DeCUDA

• Example: Tiled version of matrix multiplication on CUDA tile size: 16x16 (throughput 77GFLOPS).

```c
...//load a tile of array A and B into shared memory As and Bs

for(k = 0; k < 16; k++)  //completely unrolled
{
    Temp += As[i][k] * Bs[k][j];
}
...
```
Assembly from DECUDA

...  
mov.b32 $r12, s[$ofs4+0x0000]  
mov.b32 $r7, s[$ofs4+0x0040]  
mad.rn.f32 $r11, s[$ofs1+0x000c], $r11, $r13  
add.b32 $ofs4, $ofs3, 0x0000019c  
mad.rn.f32 $r13, s[$ofs1+0x0010], $r12, $r11  
mov.b32 $r12, s[$ofs4+0x0000]  
mov.b32 $r11, s[$ofs4+0x0040]  
mad.rn.f32 $r7, s[$ofs1+0x0014], $r7, $r13  
add.b32 $ofs4, $ofs3, 0x0000021c  
mad.rn.f32 $r13, s[$ofs1+0x0018], $r12, $r7  
...  

Multiply-Add only allows 1 source operand from the shared memory
Performance Optimization

- Enlarge the tile size (16 x 256): each thread calculates 16 elements

One thread calculates one element in the product matrix

One thread calculates \( C \) elements in the product matrix
Optimization: Loop Interchange

... //load a tile of array A and B into shared memory As and Bs
for(i = 0; i < C; i++) //completely unrolled
    for(k = 0; k < 16; k++) //completely unrolled
    {
        Temp[i] += As[i][k] * Bs[k][j];
    }
...

... //load a tile of array A into shared memory As
for(k = 0; k < 16; k++) //completely unrolled
{
    b = B[k][j];
    for(i = 0; i < C; i++) //completely unrolled
    {
        Temp[i] += As[i][k] * b;
    }
}
...

Assembly from DECUDA (after optimization)

... 
mov.u32 $r15, g[$r21]  //loading b
mad.rn.f32 $r0, s[0x001c], $r15, $r0
mad.rn.f32 $r1, s[0x0020], $r15, $r1
mad.rn.f32 $r2, s[0x0024], $r15, $r2
mad.rn.f32 $r3, s[0x0028], $r15, $r3
mad.rn.f32 $r4, s[0x002c], $r15, $r4
mad.rn.f32 $r5, s[0x0030], $r15, $r5
mad.rn.f32 $r6, s[0x0034], $r15, $r6
mad.rn.f32 $r7, s[0x0038], $r15, $r7
...

Throughput: 149 GFLOPS
Projects

• A wide range of applications

• More realistic workloads

• Much higher numbers of lines of code than programming assignments

• Similar findings to the programming assignments
  – Significant speedups from the GPU processors
  – Well justify the effort to port the code
Summary

• A course on various massively parallel architectures and the programming support

• SPMD programming model is easy to grasp; relatively low coding complexity

• The effort to port the code seems to be well justified by the performance gains for data intensive applications
Thank you and Questions?