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

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

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

G80 Architecture Overview

Streaming Processor Array

TPC TPC TPC . . . TPC TPC TPC

8 TPCs

Texture Processor Cluster

TEX

SM

SM

Streaming Multiprocessor

Instruction L1

Data L1

Instruction Fetch/Dispatch

Shared Memory

SP SP SP SP SP SP SP SP

SFU SFU

ATI Streaming Processors

Command Processor
Vertex Index Fetch
Setup Unit
Tessellator
Rasterizer
Interpolators
Geometry Assembler
Vertex Assembler
Hierarchical Z
Stream Out
Shader Caches Instruction & Constant
Ultra-Threaded Dispatch Processor
Memory Read/Write Cache
Unified Shader Processors
Texture Units
L1 Texture Cache
L2 Texture Cache
Z/Stencil Cache
Shader Export
Decompress
Compress
Render Back-Ends
Color Cache

Parallelism

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

Memory Hierarchy

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

Programming Support

	G80 (Geforce 8800)	RV670 (AMD Radeon HD 3870)
Programming model	SPMD	SPMD
Programming Language	C/C++	C
Intermediate Language	PTX	AMD/ATI IL
Assembly-level analysis	Decuda	GPU ShaderAnalyzer
Thread management	Thread hierarchy	Streaming model

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)

	Nvidia 8800 GTX	AMD/ATI HD3870
Min. # of lines in the code	18	12
Median. # of lines in the code	37	21
Max. # of lines in the code	140	35
Min. Throughput	13.8 GFLOPS	8.3 GFLOPS
Median. Throughput	67 GFLOPS	18.3 GFLOPS
Max. Throughput	149 GFLOPS	43 GFLOPS

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)

	Nvidia 8800 GTX	AMD/ATI HD3870
Min. # of lines in the code	12	15
Median. # of lines in the code	43	34
Max. # of lines in the code	115	88
Min. Throughput	0.27 GFLOPS	0.42GFLOPS
Median. Throughput	6.08 GFLOPS	1.2 GFLOPS
Max. Throughput	18 GFLOPS	2.2 GFLOPS

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).

```
…//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];
}
…
```

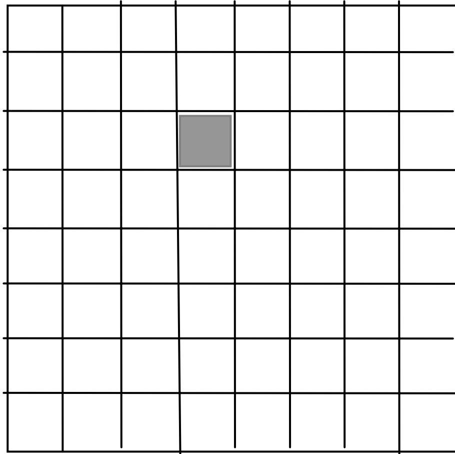
Assembly from DECUDA

```
...
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
...
```

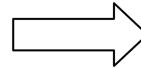
Multiply-Add only allows 1 source operand from the shared memory

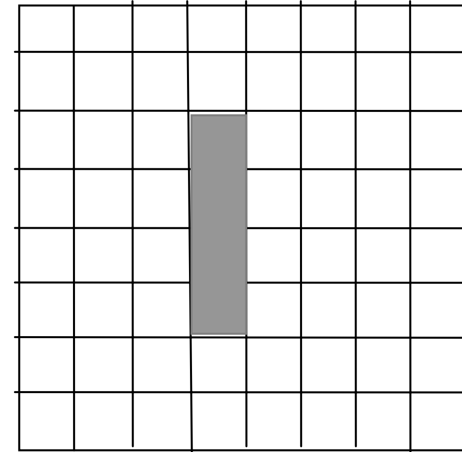
Performance Optimization

- Enlarge the tile size (16 x256): each thread calculates 16 elements



One thread calculates one element in the product matrix

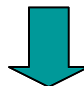




One thread calculates *C* elements in the product matrix

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?

School of Electrical Engineering and Computer Science
University of Central Florida