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

Hongliang Gao, Martin Dimitrov, Jingfei Kong, Huiyang Zhou
School of EECS, University of Central Florida
{hgao, dimitrov, jfkong, zhou}@eecs.ucf.edu

Abstract

With the future workloads predicted to feature petascale data sets, we developed a course on exploring various massively parallel architectures and programming models for data-intensive applications. The course covers the architectures of three processors, AMD/ATI RV670 Streaming graphics processors, Nvidia G80 graphics processors, and Cell Broadband Engines, and their latest programming support. The course starts with the single-program multiple-data (SPMD) programming model on ATI/Nvidia graphics processors and extends to the multiple-program multiple-data (MPMD) model using Cell processors. In this paper, we report our experience in developing & teaching this course and present some interesting insights learnt from exploring these different architectures. The results from the students' program experiences are promising: the SPMD model is easy to grasp and with a sound understanding of architectural features, significant performance gains can be achieved through various program optimizations. For example, one optimized matrix multiplication algorithm has been developed, which outperforms the carefully tuned library code (Nvidia CUBLAS) by 49%.

1. Introduction

Data volumes for future applications in many domains, ranging from science to business information processing, are predicted to grow exponentially. Such large data sets (or data streams) are often associated with intensive computation to extract useful information to end users. To address this data-intensive and computation-intensive requirement, various high-performance computing platforms have been proposed or developed. Among them, massively parallel processors like graphics processors (GPU) are promising due to their programmability and high energy efficiency. To facilitate general-purpose computing on graphic processor units (GPGPU), hardware features such as IEEE single and double precision, integer arithmetic, bitwise operations, shared memories, and scatter write capabilities have been added to the latest GPUs. Furthermore, new programming models like CUDA [3] and Brook+ [4] have been developed recently to make the GPU resources more accessible to application developers. In both CUDA and Brooks+, GPUs are used as an accelerator for the CPU to offload data-intensive computations. The GPU programs simply use extended C/C++. In contrast, the traditional GPGPU approach involves the daunting task of mapping general-purpose computation into the graphics domain and using OpenGL/DirectX APIs to access the GPU hardware.

Current GPUs feature high degrees of thread-level parallelism by supporting a large number of light-weight threads. These threads are grouped into warps (a term used by Nvidia) or wavefronts (a term used by AMD/ATI), which execute instructions in the single-instruction multiple-data (SIMD) mode. Different warps/wavefronts follow the single-program multiple data (SPMD) model. In other words, all threads in the same warp/wavefront share one program counter (PC) while each warp/wavefront has its own PC.

Compared to GPUs, Cell Broadband Engine (CellBE) architecture is also designed for data-intensive applications and it supports more flexible programming models. In Cell processors, both the synergetic processor units (SPU) and the PowerPC processor unit (PPU) have SIMD-style data processing hardware. Each of those processor units can execute a different program, thereby being capable of supporting multiple-program multiple-data (MPMD) programming model.

The advent of these massively parallel processors, however, presents a challenge to traditional parallel computing curriculum. Classical parallel architecture courses focus on low-level hardware designs for cache coherence, memory consistency, interconnect, etc., many of which are either not necessary or too costly in processors designed for data-intensive applications. Parallel programming courses, on the other hand, mainly deal with high-level programming algorithms and concepts, which are not sufficient to take full advantages of these massively parallel processors. Each of these massively parallel processors has its unique architectural features, which can have a tremendous impact on performance. As a result, the

programmer needs to understand both the architectural features and the programming models in order to choose the right processors for their target applications, develop the code, reason about the performance, and perform program optimizations. In this paper, we report our experience in developing a course for this purpose at the University of Central Florida and present some interesting results from the students' programming experiences.

The remainder of the paper is organized as follows. Section 2 presents the course, including the topics, the programming assignments, and the term projects. Section 3 discusses the interesting results from students' programming experiences. Section 4 concludes the paper.

2. The Course: Multi-Core/Many-Core Architecture and Programming

The course was developed through close collaboration with leading industry, including guest lectures and lab equipment (i.e., graphics cards) donations from AMD/ATI. The course also benefits significantly from the lecture material of "ECE 498 AL1: Programming Massively Parallel Processors" developed at UIUC [1], which focuses on Nvidia G80 processors and the CUDA programming model. We also draw on materials from the IBM Cell programming workshop [2]. In comparison to those courses, our course offers a broader coverage of various architectures and their programming support so that the students can become familiar with multiple platforms and evaluate their strengths and weaknesses to select the right one for their target applications.

The audience of this course is mainly graduate students with some background on computer architecture and/or parallel programming. This paper is based on our experience in offering the course [7] for the first time in spring 2008.

2.1 Course Description

In this course, we discuss three processor models and their current programming environments and tools. For each processor model, we follow the same trilogy: *architecture*, *programming models and supporting tools*, and *performance optimization techniques*. We start with the graphics processors (GPU) and then focus on the more flexible CellBE.

2.1.1. GPU architectures and programming

The GPU part of the course includes both the Nvidia G80 architectures and the AMD/ATI R670 architectures. For either architecture, we discuss detailed architectural features that are related to general purpose computing and we omit those features that are mainly used for graphics processing. Next, the high-level programming support, CUDA [3] for G80 architectures and Brook+ [4] for AMD/ATI RV670 architectures, and the related code development/ analysis tool chains are discussed. Then, we address performance optimization through detailed code examples and architectural impact analysis. To better understand the strengths and weaknesses of the two leading GPU architectures and their support for general purpose computing, we performed a cross-platform comparison and the results are summarized in Table 1.

Both Brook+ and CUDA use the GPU as an accelerator for the CPU program to offload the data-intensive computations and both employ the SPMD programming model. In either GPU, each of the threads works on its own data partition using its thread ID (CUDA) or index (Brook+). From the programmer's perspective, CUDA supports more flexible gather and scatter operations to access the data set while Brook+ employs a relatively more restrictive streaming model.

As shown in Table 1, both types of GPUs offer very high single-precision (SP) floating-point (FP) number computation power and memory bandwidth. AMD/ATI RV670 processors also support double-precision (DP) FP number computation at a reduced throughput as DP FP computation is achieved by combining multiple SP FP ALUs. Nvidia G80 GPUs provide on-chip shared memory structures, which enable efficient data sharing among the threads in the same thread block. In comparison, AMD/ATI RV670 architecture does not have shared memory. Instead, it opts for a larger register file than G80, which can be used for either a higher number of threads or a higher number of registers per thread. AMD/ATI RV670 also exploits instruction-level parallelism using 5–way VLIW instructions compared to scalar operations in G80 processors. Understanding these different architectural features is important to select the appropriate computing hardware for the target applications. For example, if an application involves data communication (or sharing) among neighboring threads (i.e., local communication), G80 processors may be a good fit due to their shared memory structure. If there is little data sharing among threads, the large number of registers in RV670 may enable more aggressive code optimizations such as loop unrolling or instruction scheduling, which usually result in an increase of register usage. Both GPU architectures may not be a good fit for applications

Table 1. A comparison of architectural features and programming support for AMD/ATI RV670

	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) 320 units @ released frequency of 775Mhz * 2ops for muladd = 496 GFLOPS peak; Double precision 256 units used @ released frequency of 775 Mhz; Adds→256/2*775Mhz = 99.2 GFLOPS; MulAdd→256/4*775Mhz *2 ops for MulAdd = 99.2 GFLOPS
System bus support	PCIe 2.0x16	PCIe 2.0x16
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, 4 SMs share 1 Texture subsystem	4 clusters, 16 x 5 cores per cluster, each cluster time-multiplex 1 Texture subsystem
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
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) (no L2)
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

featuring high global communication (i.e., data communication can be among any threads), for which the classical cache-coherent parallel processors could be a better choice.

2.1.2. Cell broadband engine architecture and programming

CellBE is a heterogeneous multi-core architecture. Compared to GPUs, Cell processors have much less degree of thread-level parallelism and rely on asynchronous direct memory accesses (DMA) to hide memory access latency. Also, unlike GPUs, each core in Cell processors can execute a different program, thereby being capable of supporting the MPMD programming model. The high computation power is achieved through SIMD/vector processing in either the synergetic processor unit (SPU) or the PowerPC processor unit (PPU). Given their architectural features, the key to achieve high performance on Cell processors mainly includes parallelization strategy (i.e., task or data decomposition), explicit control/data transmission using mailboxes/DMA, and vector programming. Typical programming techniques on those aspects are discussed using detailed code examples from the Cell SDK examples and the Cell programming tutorial.

2.2 Programming Assignments

For each processor model covered in the course, there are two programming assignments: matrix multiplication and 2-D convolution. In each assignment, the initial step is to get familiar with the programming environment and the syntax of the extended programming language support and to write the code, which produces equivalent results to the CPU version. The next step is to optimize the code for the target processor model. This step involves extensive uses of performance analysis tools and careful reasoning of architectural impact on the performance. The report of the assignments includes execution times for various matrix sizes and the number of lines of the code, which is used to roughly estimate the programming complexity.

2.3 Term Projects

Term projects are a key component of the course. Students are first asked to select an application with rich data-level parallelism and to choose a target hardware platform. Then, a project proposal is submitted to finalize the selection of applications and the target processor models. Students also make a short (5-10 minutes) presentation to discuss the core algorithm and the strategy to map the application to the target processor. The instructor and the experienced developers from AMD/ATI offer feedback on the proposed work. The last step of the project includes a project presentation (~30 minutes) and a technical report, which discuss how the code is ported to the target processors, how the performance optimization is performed, what the important lessons are learnt from the process, and how much performance gains are achieved compared to the CPU code.

3. Results

3.1 GPGPU Programming

In this section, we report some interesting results based on the students' programming exercises. For each programming assignment, we collect both the number of lines in the code (only the kernel function for GPUs) and the achieved throughput, which is measured as the ratio of the number of FP operations in the CPU algorithm over the average GPU execution time among 10 runs. The GPU execution time also includes the data transmission latency between the GPUs and the CPU. From all the results collected from 11 students, we present the minimum, the median, and the maximum value for each metric. In order to further separate the actual kernel execution time from any other overhead resulting from data transmission, staging/startup of deep pipeline, and CPU/GPU communication, an additional experiment was performed by repeating the kernel execution 100 times per setup on the fastest solutions for both processors. The results are shown in Table 2 and Table 3 for matrix multiplication (a product of two 2k x 2k matrices) and 2D convolution (a convolution of a 2k x 2k matrix with a 5x5 convolution kernel), respectively. All the computations use SP FP numbers. DP FP number computation is supported in Brook+ 1.0 Beta but it requires slight code changes if the program uses vector variables of 4 SP FP numbers, which need to be replaced with two vector variables of 2 DP FP numbers. The achieved throughput for DP FP numbers is around 50% of the throughput for SP FP numbers.

Table 2. Matrix multiplication (a product of two 2k x 2k matrices) on GPUs.

	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
Max. Iterative Kernel Throughput	193 GFLOPS	74.7 GFLOPS

Here, note that, although we present the results for both Nvidia 8800 GTX processors and AMD/ATI HD3870 processors, a conclusion should *not* be made in terms of which ones deliver higher performance. There are two main reasons. First, the results in Table 2 and Table 3 are based on initial effort of inexperienced students. Experienced programmers can find ways to achieve higher performance, like the GPU math library functions provided by either Nvidia or AMD/ATI (both can deliver over 100 GFLOPS on our test machine). Second, not all GPU hardware resources are exposed at Brook+/CUDA and continuing development on the programming models will likely bring higher performance. For example, the matrix multiplication program included in the AMD/ATI CAL SDK (intermediate level) has much higher throughput (213 GFLOPS) than the Brook+ version and there is current ongoing work to port the optimizations from CAL to Brook+.

Table 3. Image convolution (a 2k x 2k matrix convoluted with a 5 x 5 kernel) on GPUs

	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
Max. Iterative Kernel Throughput	112 GFLOPS	21.7 GFLOPS

Several interesting observations can be made from the results in Table 2 and Table 3. First, the SPMD programming model is easy for students to grasp, at least for simple algorithms like matrix multiplication and convolution. The students started with the un-

optimized CPU code, which has a throughput of 30 MFLOPS for matrix multiplication and 205 MFLOPS for convolution, and achieved reasonably high throughput using the two different types of GPUs. The relatively low throughput for convolution using GPUs is due to the low computational requirement, around 0.21 Giga floating-point number operations. The majority execution time (over 80%) is spent on data transmission between CPU and GPUs. However, even in this case, the optimized GPU code can achieve 6x (median) improvement using HD3870 processors or 30x (median) improvement using 8800 GTX processors over the un-optimized CPU version. The high performance gains (ranging 10X to 1000X) seem to justify the effort of porting the CPU code to the SPMD code for massively parallel processors. The same effort spent on optimizing CPU code is unlikely to produce improvements in a similar order of magnitude.

Second, if we use the number of lines in the code to estimate the coding complexity, it seems to be very promising that even inexperienced programmers can achieve decent performance gains without too much complexity or coding effort.

Third, compared the median and the maximum performance shown in Table 2 and Table 3, we can see the best designs have 2X to 3X speedups over the median performance. In fact, the best throughput for matrix multiplication achieved on Nvidia 8800 GTX processors is 49% higher than the carefully tuned matrix multiplication algorithm in the Nvidia CUBLAS library, which has a throughput of 100 GFLOPS on our test machine when the latency for data transmission between CPU and GPU is included.

Fourth, in both matrix multiplication and convolution, the tiled algorithm makes good use of the shared memory in G80 architectures to leverage the global memory accesses from different threads in the same thread block. For HD3870 processors, the large register file enables aggressive unroll & jam optimizations, but not as effective as tiling for matrix multiplication and convolution.

Fifth, we noticed during experimentation on the AMD HD 3870, that when larger kernals (4kx4k matrix multiply) or multiple kernel executions of smaller task were used, the ratio of computational workload relative to the time required for data transmission between the CPU and GPU is altered and can result in visibly significantly higher computation rates. In other words, there are large changes (1.7x-9.9x) between maximum iterative kernel throughput (the last entry of Tables 2 and 3) and the maximum throughput. Although this change is relatively smaller on the G80 (1.3x-6.2x), it does display the same effect suggesting that overlapping the data moves and kernel executions can result in higher results. Again, we are eager to see the conclusion of the ongoing CAL level optimizations being ported to Brook+.

Sixth, the performance analysis tools play a critical role in performance optimization, especially those tools with the machine assembly-level information. The reason is due to the non-linear performance effect of code optimizations [5], which are difficult to analyze at the source code level or the intermediate level. The ATI/AMD's shader analyzer provides the actual assembly instructions to be executed on hardware. The detailed VLIW scheduling information and the latency calculation are very helpful to reason about the performance. In comparison, Nvidia's PTX code is an intermediate representation, which will be further optimized and register allocated before executed in hardware. A useful resort is the deCUDA tool [6], which disassembles the CUDA binary to generate the unofficial Nvidia assembly code. During the optimization for matrix multiplication using CUDA, for example, after applying all the optimizations described in [5], a careful inspection of the assembly produced from deCUDA reveals that the multiplication-add (MAD) instruction can only have 1 source operand accessing the shared memory. This low-level instruction-set architecture (ISA) constraint is not present at the PTX level and it has significant impact on matrix multiplication, as illustrated in Figure 1.

```
…//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];
}
…
```

Figure 1. A code segment of tiled matrix multiplication (tile size: 16x16).

The code segment in Figure 1 is part of a tiled matrix multiplication (tile size 16x16, 256 threads per thread block). The arrays '*As*' and '*Bs*' are shared memory variables. The loop in Figure 1 is not translated into a sequence of MAD instructions since the MAD instruction would require two source operands from shared memory. Instead, each MAD instruction is accompanied with a load (from shared memory) instruction. Additionally, there are extra instructions to generate proper index for the load instructions (see Figure 4a). As a result, the instruction count of this completely unrolled loop is much more than 16. A good solution to this problem is to let each thread calculate *C* (e.g. 16) elements instead of 1

element and then perform loop interchange to eliminate one shared memory access in the loop body, as shown in Figure 2. Increasing the number of elements in the product matrix in each thread also substantially enlarges the tile size, 16x256 if we keep the same number of threads (256) in each thread block. The source code before and after loop interchange is illustrated in Figure 3.

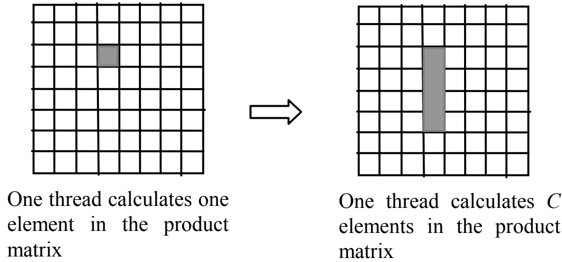

Figure 2. Increasing the tile size for loop interchange.

```
… //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];
  }
…
```

(a) code before loop interchange

```
…//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;
  }
}
…
```

(b) code after loop interchange

Figure 3. Code optimization (loop interchange) to reduce the dynamic instruction count.

As shown in Figure 3, after loop interchange, the variable '*B[k][j]*' is independent upon the inner loop index '*i*' and can be loaded into a register before the inner loop. This way, each MAD operation in the inner loop only has 1 shared memory access to '*As[i][k]*'. To eliminate offset calculation instruction, we also load a tile of the array *A* into *As* using the column-major (i.e., make *As[i][k]* and *As[i+1][k]* adjacent) and pad one row to *As* to remove bank conflicts in updating shared memories. The complete code is available at the course website [7]. Here, we show in Figure 4 the assembly code segments before and after the optimization, which are generated using deCUDA. As seen in the figure, before the optimization, there is one or two additional instructions (move or add) for every MAD instruction. After the optimization, all those extra instructions are removed. Since each core in G80 processors uses a scalar pipeline, such high reduction in the dynamic instruction count combined with the benefits of large tiles results in a significant improvement on the effective throughput over the basic tiled algorithm in Figure 1 (from 77GFLOPS to 149 GFLOPS), when computing the product of two 2kx2k matrices. The resulting matrix multiplication code also outperforms the carefully tuned library code (Nvidia CUBLAS) by 49%.

Finally, we examine whether there exists some correlation between the code size and the achieved throughput. From the results that we collected from 11 students, we observe the expected trend: larger code sizes typically imply more crafted code and therefore higher performance. However, there are many exceptions. The programs with the best performance have a high number of lines of code but usually not the longest ones. Similarly, the programs with the lowest throughput are short but may not be the shortest ones.

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

(a) Assembly code before the 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
…
```

(b) Assembly code after the optimization

Figure 4. The assembly code generated using deCUDA, before and after optimization (large tile + loop interchange).

Table 4. A summary of term projects

Project	Target SW/HW Platform	Code size (kernel) (lines of code)	Performance Results
Image Super-resolution	Nvidia CUDA	750	59x vs. Matlab code 40x vs. sequential C code
Line of Sight	Nvidia CUDA	55	For a 2kx2k height image, 3.2 million checks per second using the GPU vs. 79k checks per second using the CPU
Instant Radiosity	Nvidia CUDA	603	An 8x8x8 grid takes 0.1ms using the GPU compared 10s using the CPU
Parallel lossless data compression	Cell	231	6 SPUs achieves 3x faster than PPU. Not as good as a high performance CPU.
Ray Tracing	Nvidia CUDA & AMD/ATI Brook+	2834 in CUDA 3420 in Brook+	For a Buddha scene which has 175k triangles, it takes 0.66 sec for CPU while 0.067 sec for GPU
Parallel Sorting	Nvidia CUDA & AMD/ATI Brook+	48 in Brook+ 135 in CUDA	It takes 0.25 sec for GPUs (optimized Bitonic) to sort 4M elements while it takes 0.75 sec for the CPU (optimized Quicksort)
Triple DES	Nvidia CUDA & Cell	86 in CUDA 127 in Cell SPU code	It takes 89s for CPU, 35s for GPU, and 19s for Cell to encrypt the Bible (3953KB)
Accelerating large neural nets	Nvidia CUDA	300	47x speedup vs. CPU code when querying the a 3-layer 48x48 net; 4x speedup when evaluating the nets
Harr Wavelet	Nvidia CUDA and direct 3D9	57 in CUDA	8x speedup vs. CPU code; similar performance between CUDA and direct 3D9.
Lane Detection	Nvidia CUDA	323	9x speedup vs. CPU code in functions to detect lanes
DNA Sequencing	Nvidia CUDA	60	For DNA sequences with 1024 symbols, 3.6x speedup vs. CPU code.

3.2 Cell Programming

The programming assignment on CellBE was done in two parts. The first is to develop the code in the simulation environment and the second is to test the performance using a PS3 playstation running Fedora Linux.

Among the students (3) who finished the assignments in two weeks, the median throughput is 8.5GFLOPS for matrix multiplication (a product of two 2kx2k matrices) and 0.87GFLOS for convolution (a 2kx2k matrix convoluted with a 5x5 kernel) when 6 SPUs are used. Again, no conclusion should be drawn in comparing CellBE with GPUs due to the limited time and experiences on optimizing Cell programs. From the students' code, the median code size for matrix multiplication is 188 lines of code for the PowerPC Unit (PPU) and 95 lines for the Synergetic Processor Unit (SPU). For 2D convolution, the median code size is 227 and 99 lines of code for the PPU and SPU, respectively, implying higher coding effort than the SPMD model in GPUs. In comparison, the highly optimized matrix multiplication algorithm in the Cell SDK library has a throughput of 139 GFLOPS on the same PS3 playstation and the code size is 608 and 1362 lines of code for the PPU and SPU, respectively.

3.3 Projects

The term projects listed in Table 4 cover a wide range of applications, including ray tracing, computer vision, data compression, artificial intelligence, information security, etc.

The performance results shown in Table 4 are encouraging. The highly impressive speedups well justify the effort to port the CPU code to the massively parallel processors for relatively large applications. Again, although the CPU code is not highly optimized, speedups of similar magnitudes are not very likely to obtain by optimizing the CPU code. Additionally, the code sizes reported in Table 4 also indicate that the complexity of composing GPU codes is not overwhelming for those applications.

4. Conclusions

In this paper, we present a course we developed on exploring multi-core/many-core architectures and programming models. The course covers three

processor models, including AMD/ATI RV 670 graphics processors, Nvidia G80 graphics processors, Cell processors, and their programming models. The results from the students' programming assignments and term projects are promising: the SPMD programming model is easy to grasp; the students are able to identify data-intensive applications in their fields, whose characteristics match well with processor features; and by applying the optimization principles discussed in the class, decent speedups have been achieved with reasonably low coding complexity.

5. Acknowledgement

The authors would like to thank Mike Mantor, Justin Hensley, and Jason Yang from AMD/ATI for their guest lectures on ATI GPUs and Brook+, Mike Mantor and Vineet Goel from AMD/ATI for their help throughout the course development, and Raja Koduri, CTO AMD, for approving ATI support for this course. The authors would also like to thank the anonymous reviewers for their helpful suggestions to improve the paper and the students who took the course in spring 2008. The development of the course is supported by NSF CAREER award CCF-0747062.

6. References

[1] ECE 498AL: Programming Massively Parallel Processors, ECE Department, UIUC, http://courses.ece.uiuc.edu/ece498/al1/

[2] IBM Cell Programming Workshop, http://www.cc.gatech.edu/~bader/CellProgramming.html

[3] Nvidia Developing with CUDA, http://www.nvidia.com/object/cuda_develop.html

[4] AMD Stream Computing, http://ati.amd.com/technology/streamcomputing/index.html

[5] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk (NVIDIA), and Wen-mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA", 13th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP), 2008.

[6] decuda: disassembler for the NVIDIA CUDA binary, http://www.cs.rug.nl/~wladimir/decuda/

[7] ST: CDA6938 Multi-core/Many-core architecture and programming, School of Electrical Engineering and Computer Science, University of Central Florida http://csl.cs.ucf.edu/courses/CDA6938/