2011

Optimization of Scientific Computation for Multicore Systems

Andrew B. McClain
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all
Part of the Computer Sciences Commons

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/1041

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu, library-corescholar@wright.edu.
OPTIMIZATION OF SCIENTIFIC COMPUTATION
FOR MULTICORE SYSTEMS

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

ANDREW BLAIR MCCLAIN
B.S., Wright State University, 2008

2011
Wright State University

Meilin Liu, Ph.D.
Thesis Director

Mateen Rizki, Ph.D., Chair
Department of Computer Science and Engineering

Committee on Final Examination

Meilin Liu, Ph.D.

Jack Jean, Ph.D.

Keke Chen, Ph.D.

Andrew Hsu, Ph.D.
Dean, School of Graduate Studies
ABSTRACT


With the rapid growth of semiconductor technology, chip density has increased significantly. As the power exponent is setting hard limits to frequency increases, multi-core and chip level multi-processors have become prevalent in recent years to take advantage of the increasing chip density. In the new generation of processors, multi-core architecture design is becoming the major trend: IBM/SONY/Toshiba's Cell Broadband Engine processors contain nine cores; NVIDIA graphics processors contain more than 30 cores.

One of the biggest challenges is to efficiently utilize the computational power provided by multi-core systems. The second challenge to achieving high performance in a computer system is the growing disparity between processor and memory speeds. This thesis examines the problems of sorting, matrix multiplication, and ordinary differential equation initial value problems on two target architectures, the Cell Broadband Engine, and the Nvidia CUDA enabled graphics processor. This thesis first studies how to exploit various levels of parallelism for these application programs. At the same time, the author also tries to explore the use of memory hierarchies and other architecture features to further improve the performance.
## Contents

1 Introduction 1
   1.1 Background ............................................. 1
   1.2 Problem Statement ....................................... 5
   1.3 Organization ............................................ 5

2 Parallel Computing Concepts 7
   2.1 Introduction ............................................. 7
   2.2 Data Dependencies ........................................ 7
   2.3 Granularity .............................................. 9
      2.3.1 Instruction Level Parallelism ....................... 9
      2.3.2 Data Level Parallelism ............................... 11
      2.3.3 Task Level Parallelism ............................... 11
   2.4 Data Locality ........................................... 12
   2.5 Memory .................................................. 14
      2.5.1 Shared Memory ..................................... 14
      2.5.2 Distributed Memory ................................. 15

3 Architecture and Programming Interface 16
   3.1 Introduction ............................................. 16
## 3.2 Cell Broadband Engine

- 3.2.1 Architecture .............................................. 17
- 3.2.2 Direct Memory Access ................................. 18
- 3.2.3 Single Instruction Multiple Data .................. 22

## 3.3 Nvidia Fermi Graphics Processors and CUDA

- 3.3.1 Architecture .............................................. 24
- 3.3.2 Compute Unified Device Architecture (CUDA) ... 25
- 3.3.3 Kernels and Threads ................................. 26
- 3.3.4 Memory Coalescing ................................ 27

## 4 Bitonic Sort

- 4.1 Introduction ............................................... 29
- 4.2 Bitonic Sequence ....................................... 30
- 4.3 Bitonic Merge and Sort ................................ 30
- 4.4 Bitonic Sort on the Cell B.E. ......................... 34
- 4.5 Bitonic Sort using CUDA .............................. 37
- 4.6 Results and Conclusion ................................. 42

## 5 Matrix Multiplication

- 5.1 Introduction ............................................... 46
- 5.2 Matrix Multiplication Algorithm .................... 46
  - 5.2.1 Simple Matrix Multiplication .................... 46
  - 5.2.2 Data Dependencies of Matrix Multiplication . 47
  - 5.2.3 Matrix Tiling ........................................ 48
  - 5.2.4 Data Locality of Tiled Matrix Multiplication 49
- 5.3 Matrix Multiplication on the Cell B.E. ............ 49
  - 5.3.1 Algorithm ............................................ 49
CONTENTS

5.3.2 Buffering Scheme ................................. 51
5.3.3 Single Instruction Multiple Data ................. 53

5.4 Matrix Multiplication using CUDA ................. 55
  5.4.1 Algorithm ........................................ 55
  5.4.2 Shared Memory .................................... 55
  5.4.3 Memory Coalescing ............................... 58

5.5 Results and Conclusion .............................. 58

6 Ordinary Differential Equation Solver ............... 61
  6.1 Introduction ........................................ 61
  6.1.1 LU Decomposition and Cholesky Factorization .... 62
  6.1.2 Backward Differentiation Formula ............... 62

6.2 Cell Broadband Engine Implementation ............... 65

6.3 Results and Conclusions .............................. 66

7 Conclusions and Future Work ............................ 70

Bibliography ................................................. 74
# List of Figures

2.1 Architecture of a shared memory system . . . . . . . . . . . . . 14

2.2 Architecture of a distributed memory system . . . . . . . . . . 15

3.1 Playstation 3 Cell architecture . . . . . . . . . . . . . . . . . . 17

3.2 Impact of DMA single buffering on runtime of Cell program . . 21

3.3 Comparison of SISD and SIMD . . . . . . . . . . . . . . . . . . 22

3.4 Example of an SIMD add operation as outlined in Algorithm 4 . 23

3.5 Diagram of CUDA device architecture . . . . . . . . . . . . . 25

4.1 Bitonic merge operation [14] . . . . . . . . . . . . . . . . . . . 32

4.2 Bitonic sort operation [14] . . . . . . . . . . . . . . . . . . . . 33

4.3 Concrete bitonic merge example (three steps for eight elements) 33

4.4 Graph Comparison of Bitonic Sort Running Times . . . . . . . 43

5.1 Example of tiled matrix multiplication . . . . . . . . . . . . . 48

5.2 Tiled matrix multiplication scheme for Cell . . . . . . . . . . 51

5.3 CUDA matrix multiplication diagram not exploiting shared mem-

5.4 Graph of running timse of matrix multiplication implementations 59
6.1 The solution of each equation of an OREGO initial value problem over the range of time units from 0 to 1000 . . . . . . . . . 68
List of Tables

4.1 The indices of the array to be sorted with their binary representation .......................... 39
4.2 First iteration of outer loop for the CUDA bitonic sort implementation .......................... 39
4.3 Second iteration of outer loop for the CUDA bitonic sort implementation ....................... 40
4.4 Third iteration of outer loop for the CUDA bitonic sort implementation ....................... 41
4.5 Complete concrete example of the bitonic sort algorithm ........................................... 41
4.6 Running Times of the Bitonic Sorting Algorithm ..................................................... 42
5.1 Table of running times (in seconds) of the matrix multiplication implementations ............... 59
6.1 Running time (seconds) of various initial value ODE problems on Cell with variable step size h .......... 67
Chapter 1

Introduction

1.1 Background

Mathematician John von Neumann, in 1945, wrote a report for the United States Army Ordnance Department outlining the concept of an EDVAC, or Electronic Discrete Variable Automatic Computer. The EDVAC used a stored program model that executed statements sequentially in the computer’s memory [17].

The ideas proposed by von Neumann were expanded in the 1960s with the creation of time-sharing operating systems. Multiple users could submit jobs to run on mainframe computers. These systems exploited concurrency at the process level and the handling of jobs was the responsibility of the systems programmer rather than the applications programmer.

Early personal computers were similar to those proposed by von Neumann allowing a single user the ability to run a program in sequential steps. The users of early personal computers interacted with simple text based interfaces. As these computers became more popular, and the technology
more advanced, more sophisticated environments were created that allowed
users to run multiple applications in the same environment.

The ability to run multiple applications and threads in the same envi-
ronment was achieved through an implementation of simultaneous multi-
threading (SMT) [4]. SMT employs an approach that utilizes a time-sliced,
multitasking operating system that interleaves execution of multiple threads
by sharing computing resources. A thread is defined as a basic unit of com-
putation and is defined by the state of the system, including a program
counter that points to the current statement in the execution of the pro-
gram, as well as the stack and other state information. Because the only
information needed to define a thread is the state of the architecture, the
state information may be saved to define the behavior of a logical proces-
sor. A single processor may schedule computation from multiple threads
by loading and storing the state information of the thread. The sharing
of resources and scheduling of threads is handled by the architecture and
corresponding operating system, although only one thread may be actively
processed at any given time [4].

In order to have truly parallel execution, the architecture must support
simultaneous execution on multiple cores rather than merely interleaving
the threads in a single time-shared resource. In 1966, Michael J. Flynn
proposed a means of describing and categorizing parallel systems known as
Flynn’s taxonomy [10]. Parallel systems are defined along two dimensions,
the number of data streams and the number of instruction streams. Parallel
systems may have a single or multiple data streams as well as a single or
multiple instruction streams. Thus, there are four types of systems defined
by Flynn’s taxonomy. Most modern computers fall into the categories of
single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD) [4].

In 1965, Gordon Moore made the observation that as semi-conductor technology advances, the amount of transistors available to manufacturers would approximately double every 18 to 24 months [15]. The increase in transistors led to an increase the clock speed and throughput of single processors, but more intelligent architecture design utilizes the available transistors to form multiple processing cores. Multicore processing uses chip multiprocessing (CMP) to schedule thread execution on multiple execution cores within a single chip. Each of the execution cores has its own set of hardware resources to enable truly parallel computation that can be exploited to increase the performance of certain applications.

Parallel computation involves performing multiple calculations simultaneously. A large problem is divided into several smaller problems which can be solved at the same time. Parallel computation in the form of multicore processors is becoming widely used. This is because of several physical constraints of single core systems. There is a limit to the scaling of the clock speed, or frequency, that traditional processors operate at. There is a measurable propagation delay for electric charge to travel through media [9]. Increasing the frequency of the processor also involves increasing voltage to ensure that the proper signal arrives within the given clock cycle. This results in increased energy consumption and heat generation. Multicore processors have thus become a trend to increase computational power and provide an easily scalable architecture [8].

Amdahl’s Law is used for determining the maximum speed up of a parallelized program compared to that of a sequential program. In the 1960’s,
Gene Amdahl derived the following law:

\[ SU = \frac{1}{\frac{1}{S} + \frac{1}{1-S}} \]

where SU is the maximum potential speed up of a parallelizable program and S is a fraction representing the time a program spends on non-parallelizable portions of code [5]. For example, if a program spends \( \frac{1}{2} \) of its time on code that must be executed sequentially, the maximum theoretical speed-up for a parallelized version of the code running on two cores is:

\[ SU = \frac{1}{\frac{1}{2} + \frac{1}{1-\frac{1}{2}}} \]
\[ SU = \frac{4}{3} \]

Overheads also exist for the communication and synchronization between threads. Amdahl’s law may be adapted to compensate for overhead as follows:

\[ SU = \frac{1}{\frac{1}{S} + \frac{1}{1-S} + \frac{1}{O(n)}} \]

where O(n) describes the fraction of execution time spent on dealing with the overhead of utilizing n cores. It is possible to achieve a speedup of less than one given a large overhead. That is, the overhead dominates the parallel computation and causes a loss of performance.

The overall performance of single-core processors is limited due to the fact that instruction streams may only be interleaved, and not executed simultaneously. These processors also share resources among threads that are required to wait until the resource is available. In contrast, multicore system programmers may divide workloads amongst cores that contain their own set of resources. The effective, architecture conscious, decomposition and division of a problem over multiple cores is essential to parallel performance.
1.2 Problem Statement

This thesis presents several parallelizable applications that were manually tuned for the Cell Broadband Architecture as well as the Nvidia CUDA enabled graphics processors. The manual tuning of these algorithms takes into consideration the use of memory hierarchies, algorithm design, and architecture features to reduce the overall running time of the applications.

The target applications for manual tuning are bitonic sort, matrix multiplication, and ordinary differential equation initial value problems. Sorting is a fundamental problem in the realm of computer science and has applications in database systems as well as other algorithms such as binary searches. The matrix multiplication kernel has been studied extensively due to its appearance in numerous algorithms related to linear algebra and the physical sciences. The solving of initial value problems from coupled ordinary differential equations is important for chemical kinetics, electronics, and modelling other interactions. The performance gains of these computation kernels may be utilized in many applications.

1.3 Organization

The organization of the paper is as follows:

Chapter 1 is an introduction to parallel programming as well as the outlining the problems that the thesis will address.

Chapter 2 is a review of basic parallel concepts such as data dependencies, granularity, data locality, and memory hierarchies.

Chapter 3 is an exploration of the architecture specific features of the Nvidia graphics processors and the Cell Broadband Engine, as well as the
corresponding interface, that will be utilized in the parallel implementations of the problem set.

Chapter 4 outlines the bitonic sorting algorithm as well as the implementations on both target architectures.

Chapter 5 outlines a tiled matrix multiplication algorithm as well as the implementations on both target architectures.

Chapter 6 is reserved for the ordinary differential equation initial value problem solver implementation on the Cell Broadband Engine using backward differentiation.

Chapter 7 is a summary of the results presented in previous chapters as well as the direction of future work.
Chapter 2

Parallel Computing Concepts

2.1 Introduction

Multicore systems feature several processing elements and functional units with one machine. Often, there exist multiple cores along with a more traditional master processor which controls the division of the subtasks and communication between cores. The issues of communication and synchronization between different tasks running on different processing cores introduce a new level of complexity to writing parallel programs.

In this chapter, an overview of the basic concepts and principles related to the techniques presented in this thesis, including data dependencies, granularity, data locality, and memory hierarchy, is provided.

2.2 Data Dependencies

When a statement in a program accesses the same data as a previous statement, there exists a data dependency between the two statements. The
CHAPTER 2. PARALLEL COMPUTING CONCEPTS

Logical decomposition of a problem into parallel subtasks must take into consideration the data dependencies present in the code. Statements with no data dependency between them can be directly parallelized without affecting the result of the program execution. When dependencies exist, they must be maintained in the parallelized version of the program in order to ensure that the program behaves as expected. There are three types of data dependence that must be analyzed and maintained: true dependency, anti-dependency, and output dependency.

\[ S1 : A[1] \leftarrow C[1] \]

True data dependence involves two sequential statements, \( S_1 \) and \( S_2 \), such that \( S_1 \) writes to a memory location that the next statement, \( S_2 \), reads from. In the example, \( A[1] \) is written to in \( S_1 \) and then later read in \( S_2 \). The only case in which \( B[1] \) will have the same value is if the statements were switched is upon the precondition that \( A[1] = C[1] \).

Anti-dependence involves two sequential statements, \( S_1 \) and \( S_2 \), such that \( S_1 \) reads from a memory location that the next statement, \( S_2 \), writes to. The above statements show an example of an anti-dependency. In the example, \( A[1] \) is read from in \( S_2 \) and then later written to in \( S_3 \). The only case in which \( A[1] \) will have the same value is if the statements were switched is upon the precondition that \( A[1] = D[1] \).

Output dependence between two statements, \( S_1 \) and \( S_2 \), occurs when \( S_1 \) writes to a memory location that a proceeding statement, \( S_2 \), also writes to. An output dependency is illustrated in the statements above. In the example, \( A[1] \) is written to in \( S_1 \) and then later written to in \( S_3 \). The
only case in which $A[1]$ will have the same value is if the statements were
switched is upon the precondition that $C[1] = D[1]$.

Modifying the order in which statements exhibiting a data dependence
are executed may impact the behavior of the program. Therefore, to retain
the general program behavior, all dependencies between statements must
be preserved.

2.3 Granularity

The term granularity, in relation to parallel computation, is a measurement
of the ratio of computation to communication overhead. Fine-grained solu-
tions involve relatively small tasks performed in parallel. Finer granularities
allow for more parallelism in code, but also may incur higher penalties in
the form of communication overhead and synchronization. Coarse-grained
solutions involve larger computation loads for individual tasks, but data is
normally transferred less frequently. It is important to find the correct load
balance and level of parallelism to exploit in a given parallelizable problem.

Three levels of parallelism, from fine to coarse, are examined: instruction
level parallelism, data level parallelism, and task level parallelism.

2.3.1 Instruction Level Parallelism

Instruction level parallelism deals with the overlapping of operations. If two
statements have no data dependencies between them, it is valid to execute
them in parallel. This level of parallelism is often handled by the hardware
or compiler.


An example of statements that may utilize instruction level parallelism is shown above. Note that there exists an output dependence between \( S_1 \) and \( S_2 \). These two statements must be executed sequentially. However, \( S_2 \) and \( S_3 \) are independent statements that may be executed simultaneously or even out of order.

Most of the details of instruction level parallelism are abstracted away from the programmer and controlled by the underlying hardware and the compiler. There exist many techniques for the hardware and compiler designers to utilize in order to achieve high levels of instruction parallelism.

Often, processors contain multiple function units. A processor may have several arithmetic floating point function units along with several integral arithmetic units and logic units. Instructions without data dependencies may be scheduled on separate function units to overlap computation.

Instruction pipelining allows for the partial overlapping of instructions. Instruction pipelining is the division of the execution of an instruction into independent stages in order to increase the throughput of a processor. For example, \( S_1 \) and \( S_2 \) may be partially overlapped. First, the processor must fetch the instruction for \( S_1 \) (the first stage in the pipeline). While \( S_1 \) is being decoded (the second stage in the pipeline), the \( S_2 \) instruction can be fetched. Different hardware within the processor corresponding to each stage present in the pipeline may operate simultaneously, partially overlapping instructions.
2.3.2 Data Level Parallelism

Data level parallelism, or loop level parallelism, involves decomposing a problem and distributing the data to be processed across multiple computation nodes.

```c
for ( i = 0 to N )
{
    A[i]++;
}
```

Single instruction multiple data (SIMD) is an example of data level parallelism. Assume that the same instruction is performed on multiple pieces of data. The data may be divided evenly among processing cores running the same code. In the algorithm above, if N=8, and there exist two cores in which to perform the calculations associated with each loop iteration, the computation can be divided amongst them. Processor 1 can perform loop iterations \{0,4\} and processor 2 can perform loop iterations \{5,8\}.

2.3.3 Task Level Parallelism

Task level parallelism, also known as thread level parallelism, is more coarse than instruction level parallelism. It distributes tasks, or threads, across several computation nodes. Each processor is responsible for executing a different thread, although the threads may contain the same code and operate on different data. It is often necessary for the parallel threads to communicate data.
The above algorithm shows an example of how tasks can be split between
processors. Assume that a program requires two tasks, A() and B(), to
be performed and no dependencies exist between the two tasks. Each of
the two independent tasks may be scheduled on separate processors. For
example, cpu1 can perform task A() and cpu2 can perform task B() in
parallel, reducing the overall running time of the program.

2.4 Data Locality

Data locality is defined through the reuse of memory locations. A memory
location is defined as being reused if it is loaded or stored by more than
one reference in a loop, or throughout multiple iterations in a loop [18]. A
program with strong locality may be optimized through several techniques
depending on the memory hierarchy. There are two types of locality of
concern: temporal and spatial.

```c
for ( i = 0 to N )
{
}
```
2.4. DATA LOCALITY

Temporal locality occurs when there are two or more uses of the same memory reference at different times [18]. The above algorithm is an example that shows a strong temporal locality with regards to B[1]. The value B[1] is reused in each of the loop iterations.

Spatial locality refers to the reuse of nearby memory locations [18]. The algorithm above also demonstrates strong spatial locality with regard to A[i]. Note that the array A is accessed sequentially. Element A[0] is accessed in the first iteration of the loop, followed by A[1], A[2], and so on. Because A is allocated as a contiguous space in memory, after the first iteration of the loop, subsequent iterations access an element of A that is adjacent to the previous iteration.

Cached systems store frequently used data in a small capacity bank close to the functional units of a processor. The access times from cache are smaller than that of main memory access, thus the more a program accesses the same memory location consecutively, the more benefit it will gain from cache. Modern cache systems do not read a single word from main memory, but rather several contiguous words that make up a cache line, resulting in quick access to neighboring values.

Prefetching is another technique used to exploit data locality. When memory is accessed, contiguous blocks of memory surrounding the reference may also be requested at the same time, hiding the memory latency that would be involved in waiting for the current computation to finish and accessing the next memory location.
2.5 Memory

2.5.1 Shared Memory

Figure 2.1: Architecture of a shared memory system

Shared memory is a memory space in which multiple processors or programs can simultaneously access and is often used to communicate data. Figure 2.1 is an abstract view of a shared memory system.

Physically, shared memory is a block of memory that can be accessed by several processors. Shared memory models alleviate some of the problems programmers face when data needs to be shared between processors by creating a unified address space in which all data may be found. Different processors may communicate data with similar overhead as reading and writing to a memory location, but also introduces several problems. Cached systems using shared memory must take into consideration cache coherence. If each processor has its own cache and access to the shared memory, as it operates on data values stored in cache, the system must ensure that any changes made are propagated throughout the other processors.
2.5.2 Distributed Memory

Figure 2.2: Architecture of a distributed memory system

In a distributed memory system, each processor has its own private memory space, thus each process can only access data local to the processor in which it is running. If processors need to share data, then it must be explicitly communicated. Figure 2.2 shows the layout of a distributed memory system.

Physically, a distributed memory system contains processors or memory banks tied together by an interconnection network. Communication is controlled by a network protocol or memories with multiple read and write ports.

Several challenges arise for programmers when dealing with distributed memory systems. The distribution of data over the memories is a key issue. The goal is to reduce the amount of inter-network communication that may contain a high overhead penalty. Optimally, each processor in a distributed memory system would operate only on local data.
Chapter 3

Architecture and Programming Interface

3.1 Introduction

The parallelization and manual tuning of a program is dependent on the underlying architecture and the available programming interfaces for the system. These features must be utilized to the maximum potential in order to construct a solution with optimum performance.

3.2 Cell Broadband Engine

Sony, Toshiba, and IBM worked jointly to create and manufacture the Cell Broadband Engine starting in 2000 [13]. The process took over four years and IBM filed several patents during this period with regards to both the architecture and the software environment used to develop code [2].
3.2. CELL BROADBAND ENGINE

3.2.1 Architecture

The basic configuration of the Cell Broadband Engine includes a single power processing element (PPE) and multiple synergistic processing elements (SPE) on the same chip. These components are connected by a high speed bus called the element interconnect bus (EIB) [1]. Sony’s Playstation 3 includes a Cell processor with a total of six SPEs enabled. Figure 3.1 shows a diagram of the Cell Broadband Architecture.

The PPE is used to control the SPEs present on the chip, which perform the majority of the computation. The PPE can schedule and control processes running on the SPEs, thus it is required to have additional instructions in order to control the SPEs. The PPE is based on the 64-bit PowerPC architecture, and thus can handle conventional operating systems.
The PPE allows for multithreading and contains a 64KB level 1 cache and a 512KB level 2 cache with 128 bit cache lines and can read and write directly to main memory. The PPE contains three different register sets: a 64 bit general purpose set, a 64 bit floating point set, and a 128 bit Altivec set that supports single instruction multiple data.

The synergistic processing element, SPE, is composed of an SPU, or synergistic processing unit, and a MFC, memory flow controller. The MFC contains the direct memory access, DMA, controller as well as the memory management unit and bus interface. Unlike the PPE, the SPE only contains 128 bit register sets. Each SPE contains a 256 KB local scratchpad memory known as the local store, LS, that holds both data and instructions. The SPE may only operate on data stored in the local store, therefore it must explicitly copied to and from the main memory.

The element interconnect bus, EIB, connects the various components on the chip, including the SPEs and the PPE. The EIB is a circular ring of four unidirectional 16B wide channels that may support up to three concurrent transactions. Because the EIB contains a total of twelve elements, including PPEs, SPEs, I/O and memory controllers, it takes at most six steps for data to reach a component. If it would take more than six steps, there would exist a shorter path in the opposite direction. Each component is connected to the EIB via a 16B read port and a 16B write port. Thus, each component can read and write 16B per EIB clock tick.

3.2.2 Direct Memory Access

Direct memory access, or DMA, allows hardware components to access the system memory independently of the central processing unit. In the case of
Cell, it allows for communication between components on the chip, copying data between the local stores of the SPEs and main memory. DMA transactions involve copying blocks of memory from one device to another. In the case of Cell, each DMA request can transfer a total of 16KB of data. The processor will issue a command to transfer data that will then be serviced by the DMA controller, freeing the processor from the details of the actual transfer. The DMA controller will issue an interrupt to the processor to signal the completion of an operation. DMA is used so that the processing core suffers a much lower overhead as it does not need to be occupied with memory requests. This allows the processing core to transfer data and perform computations concurrently. Without DMA, the processor would be occupied for the entire time that memory was being read from or written to, rather than being able to perform useful computation [12].

```c
// Get all data from main memory
for ( int block = 0; block < ARRAY_LENGTH / BLOCK_SIZE; block++ )
{
    (request block number (block) from main memory)
    (wait for DMA transaction to complete)
}

// Increment the values from the last block read
for ( int i = 0; i < ARRAY_LENGTH; i++ )
    A[i]++;

// Send answer back to main memory
(write result array A back to main memory)
```

**Algorithm 1:** Cell code without buffering

The nature of DMA transactions allows for a technique known as buffering where data transfer and computation are overlapped. Consider the operation performed in Algorithm 1. It is a simple method where an entire
Algorithm 2: Cell code exploiting single buffering

Array of data is read from main memory using DMA requests. The data may not be able to be read in a single DMA request as each transaction is limited to 16KB of data. Each value read is then incremented. Using this method, the entire array must be read before computation can continue. A more elegant approach is shown in Algorithm 2. Here, the array is split into blocks. A DMA request is issued for the first block and when it is complete, another request is issued for the next block. While the DMA controller is handling the data transfer for the second block, computation begins for the first block. The process of prefetching the next block for computation continues, allowing all or part of the latency associated with memory access to be hidden within computation. A requirement for this type of buffering scheme is that each block of data must be able to be processed indepen-
Figure 3.2: Impact of DMA single buffering on runtime of Cell program

dently of the other data. Figure 3.2 shows how overlapping data access and computation may have an effect on the overall running time of the program.

With software controlled explicit DMA transactions, the programmer must be aware of the coherency issues that may arise. For example, if SPE0 reads a value from main memory and stores a copy in its independent local store and then modifies the value, when SPE1 reads the value from main
memory, it will read an old, stale copy, not the updated value that resides only in SPE0’s local store. The data must be flushed from the local store back to main memory in order for the other SPEs to receive a valid copy.

### 3.2.3 Single Instruction Multiple Data

![Comparison of SISD and SIMD](image)

**Figure 3.3:** Comparison of SISD and SIMD

```c
for ( int i = 0; i < ARRAY_LENGTH; i++ )
{
    C[i] = A[i] + B[i];
}

**Algorithm 3:** Cell code without SIMD instructions

```c
for ( int i = 0; i < ARRAY_LENGTH; i = i + 4 )
{
    C[i] = spu_add( A[i], B[i] );
}

**Algorithm 4:** Cell code with SIMD instructions
3.2. CELL BROADBAND ENGINE

Figure 3.4: Example of an SIMD add operation as outlined in Algorithm 4

Single instruction multiple data, SIMD, architectures exploit data level parallelism. They consist of multiple processing elements that may perform a single operation on multiple data simultaneously. Figure 3.3 shows the difference between SISD and SIMD operating on multiple data streams. SPEs in the Cell contain special 128 bit Altivec vector registers that support SIMD instructions. Algorithm 3 shows an example of a loop that may benefit from using Cell’s SIMD instructions. Algorithm 4 shows an improved version that takes advantage of the vector registers. Two sets of four 32 bit floating point numbers (128 bits for each set) may be added together using a single SIMD instruction. The result is that the SIMD version uses roughly one quarter of the number of instructions compared to the previous version. Figure 3.4 shows how the Cell processor performs the SIMD addi-
tion operation shown in Algorithm 4. The arrays A, B, and C are assumed to contain floating point numbers.

3.3 Nvidia Fermi Graphics Processors and CUDA

The Nvidia GeForce 400 series, named Fermi after physicist Enrico Fermi, was released to the public in April 2010. The Fermi graphics processors (GPUs) were not only meant for video applications, but to act as a coprocessor to a host machine for computationally intensive tasks. GPUs have several applications in biology, cryptography, and other forms of scientific computation.

3.3.1 Architecture

The first Fermi based product included 512 processors divided into sixteen groups of 32. The GTX 470 used for the experiments outlined later in this paper has two of the groups disabled as well as one of the memory controllers, resulting in 448 stream processors and 1.25 GB of global memory. The stream processors are divided into sixteen streaming multiprocessors with thirty-two cores each. Each multiprocessor contains a shared memory space of up to 48KB that acts as a software controlled cache and offers a low latency means of sharing data between cores in a single multiprocessor [3]. Figure 3.5 shows the CUDA device architecture containing N streaming multiprocessors and M cores in each multiprocessor.
3.3. NVIDIA FERMI GRAPHICS PROCESSORS AND CUDA

Figure 3.5: Diagram of CUDA device architecture

3.3.2 Compute Unified Device Architecture (CUDA)

The engine for controlling the Fermi architecture GPUs developed by Nvidia is called the compute unified device architecture, or CUDA. Programmers can access CUDA through several programming languages such as C and C++. CUDA provides various extensions as well as a virtual instruction set and memory space for programming on a wide variety of Nvidia GPUs.
3.3.3 Kernels and Threads

Methods for computation on CUDA devices are called kernels. Kernel code is compiled specifically for the device using Nvidia’s nvcc compiler for C and C++ code. These kernels are then invoked from the host machine which will block until completion of the kernel. There are several identifiers that classify the kernels. Kernels marked as _global_ may be called from the host or device, while those marked as _device_ may only be invoked from the CUDA device.

```c
__global__ void increment(float *A, int N)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if ( id < N )
        A[id]++;
}
```

**Algorithm 5:** A CUDA kernel that increments values in an array

Algorithm 5 is an example of a kernel that can be called from the host machine and run on a CUDA enabled device. The kernel shown takes two arguments, an array of floating point values, A, and the size of the array, N. The thread id can be computed using the indices provided by the CUDA API. The structure blockIdx contains the logical coordinates of the block. The dimensions of the block are referenced using blockDim and the thread id is contained within threadIdx. Note that each of the indices provided by CUDA may be multiple dimensions, although only one dimension is shown in the example.

The following shows how a kernel may be invoked from the host machine:

`increment <<< N/BLOCK_SIZE + 1, BLOCK_SIZE >>> (A, N);`

The arguments contained within the angle brackets define how the prob-
lem is to be decomposed across the available streaming multiprocessors. The first argument is the number of thread blocks to create. A thread block is a logical division of the problem with the requirement that computation in a thread block is independent on the computation of another thread block within a kernel. It is not guaranteed that thread blocks will be scheduled to a multiprocessor in order. The second argument is the number of threads contained within each thread block. Thus, the total number of threads created for the kernel is the product of the number of thread blocks and the number of threads per block. The kernel will execute on a logical grid of thread blocks, where each thread and thread block is given a multidimensional identification number.

Kernels execute the same instructions on different sets of data. The data to be processed is related to the identification number of the thread. Thread blocks may be logically split into one, two, or three dimensional configurations, and within each kernel, there is a block identification number available for each of the three dimensions. Likewise, there is a thread identification number for each logical dimension of threads within a single block. Using these numbers, the kernel can compute the data set that each thread will operate on.

3.3.4 Memory Coalescing

Memory coalescing is a coordinated read performed by threads belonging to the same half-warp within a thread block. CUDA devices are capable of reading 64, 128, and 256 byte segments from global memory corresponding to each thread reading a 32 bit value, 64 bit value, and 128 bit value, respectively. The data must be aligned to a multiple of the region size
being read from the global memory. It is also a requirement that the i-th thread in a half-warp reads the i-th value in the region. When all of the requirements are satisfied, one large memory transaction may occur for the entire half-warp to read data from the global memory rather than each individual thread creating one read operation.
Chapter 4

Bitonic Sort

4.1 Introduction

Sorting is a fundamental, recurring problem in computer science. It is utilized in many applications such as indexing databases and organizing lists. Sorting may also be used to gain better performance for other algorithms such as a binary search, finding the median of a list, or finding duplicates within a list.

Parallel sorting algorithms, such as the bitonic sort proposed by Batcher in 1968, have adapted dramatically in recent years due to the proliferation of multicore architectures and the need for high performance sorting in applications such as database systems [6]. This chapter illustrates the implementation of Batcher’s bitonic sorting algorithm on modern multicore systems.
4.2 Bitonic Sequence

A monotonically increasing sequence is a sequence of numbers whose values grow larger as the sequence continues. Likewise a monotonically decreasing sequence is a sequence of numbers whose value grows smaller as the sequence continues. Consider a sequence A of n numbers. In a monotonically increasing sequence, for every $A_x$ in the sequence $A = \{A_0, A_1, ..., A_{n-1}\}$, $A_{x+1}$ is greater than $A_x$ for all $0 \leq x < n$. Similarly, for a monotonically decreasing sequence, $A_{x+1}$ is less than $A_x$ for all $0 \leq x < n$.

A bitonic sequence is a concatenation of a monotonically increasing sequence with a monotonically decreasing sequence. Any circular shift of a bitonic sequence is also bitonic [11]. For example, the following sequences are considered bitonic:

$\{2, 4, 6, 8, 7, 5, 3, 1\}$
$\{6, 8, 7, 5, 3, 1, 2, 4\}$
$\{5, 8, 9, 7, 6, 1, 2, 3\}$

4.3 Bitonic Merge and Sort

By definition, any sequence of length two is a bitonic sequence. Bitonic merge is a process that transforms a bitonic sequence into a monotonically increasing or decreasing sequence. Thus, recursive applications of the bitonic merge will result in a sorted set.

Given a sequence of n numbers $A = \{A_0, A_1, ..., A_{n-1}\}$ the sequence can be divided into two subsequences $B = \{B_0, B_1, ..., B_{n/2-1}\}$ and $C = \{C_0, C_1, ..., C_{n/2-1}\}$ where for every $0 \leq x, y < n/2 - 1$, $B_x < C_y$, and B and C are both bitonic.
sequences. This defines an application of the comparator network \( N_n \) [6]. The sequence is sorted using multiple applications of the comparison network.

```c
bool ASCENDING = true;
bool DESCENDING = false;

void sort( int[] a, int size )
{
    bitonicSort( a, 0, size, ASCENDING );
}

void bitonicSort( int[] a, int lo, int n, boolean dir )
{
    if ( n > 1 )
    {
        int m = n / 2;
        bitonicSort( a, lo, m, ASCENDING );
        bitonicSort( a, lo+m, m, DESCENDING );
        bitonicMerge( a, lo, n, dir );
    }
}

void bitonicMerge( int[] a, int lo, int n, boolean dir)
{
    if ( n > 1 )
    {
        int m = n / 2;
        for ( int i = lo; i < lo + m; i++ )
            if ( dir == ASCENDING && a[i] > a[i + m] 
                || dir == DESCENDING && a[i] < a[i + m] )
            {
                swap( a[i], a[i + m] );
            }
        bitonicMerge( a, lo, m, dir );
        bitonicMerge( a, lo+m, m, dir );
    }
}
```

**Algorithm 6:** Bitonic sort recursive implementation in C for single core system
CHAPTER 4. BITONIC SORT

Figure 4.1 is a visual representation of the bitonic merge. Given an unsorted sequence, $A$, it is divided into two subsequences $B$ and $C$ using the previously defined comparison network, so each element in $B$ is less than the elements in $C$. Then, the bitonic merge operation is performed recursively on each half of the sequence. Algorithm 6 is a C implementation of the recursive bitonic sorting algorithm. Figure 4.3 shows a concrete example of the bitonic merge operation on a bitonic sequence of length eight. The bitonic merge is done in three steps.

The input to the bitonic merge operation must be a bitonic sequence. The bitonic sequence is created by the bitonic split method. Figure 4.2 is a visual representation of the bitonic split operation. Given an unsorted sequence $A$, it is divided into two equal halves, $B$ and $C$ which are both unsorted. Bitonic sort is called recursively on each half, sorting the first
4.3. BITONIC MERGE AND SORT

Figure 4.2: Bitonic sort operation [14]

Figure 4.3: Concrete bitonic merge example (three steps for eight elements)
half will become an ascending sequence, while the second half will become a descending sequence. The two halves are then combined using the bitonic merge to result in a sorted set. Figure 4.3 shows a concrete example of the bitonic merge.

To form a sorted sequence of $n$ elements from a bitonic sequence that is two sorted sequences of length $n/2$, it takes $\log_2(n)$ rounds of comparisons. Thus, for the entire sorting procedure, it takes $T(n) = \log_2(n) + T(n/2)$ comparison rounds.

4.4 Bitonic Sort on the Cell B.E.

Both the PPU and the SPUs support single instruction multiple data (SIMD) commands. Each SPU contains a set of special 128-bit registers for use with the SIMD instructions. That means four single precision floating point numbers or two double precision floating point numbers can be processed using one SIMD instruction. The two instructions of note pertaining to the bitonic sort are fmaxf4() and fminf4() which both operate on single precision floating point numbers and are available for both the PPU and the SPU. Each of these functions takes two arguments (vectors of four single precision floating point numbers) and returns a vector containing the max or min respectively by comparing each of the four elements in the vector.

The Cell Broadband Engine is the target of the bitonic sort implementation because of the powerful synergistic processing elements and the availability of special SIMD instructions. Several assumptions are made during the implementation for simplicity. It is assumed that the sequence size is a power of two because of the logarithmic nature of the algorithm. This is
also why four SPUs instead of the available six.

During each round of the bitonic sort, the work of comparing and swapping elements are independent of the other elements. Also, consider that after two rounds of bitonic merges, the sequence can be divided into four parts with no dependence on each other. So, the goal of the implementation is to use the PPU to perform several merges until the data can be divided among the SPUs in a way that no SPU to SPU communication is required.

Before proceeding with the bitonic merges, the sequence must first be bitonic. For a n-element sequence, each of the four SPUs used will read n/4 elements into their local store and perform a quick sort. Each SPU is assigned an id number starting with zero. Odd numbered SPUs will sort in descending order, and even numbered SPUs will sort in ascending order. The result is a sequence that is the concatenation of two bitonic sequences. The first bitonic sequence can be sorted in an ascending order and the second bitonic sequence can be sorted in a descending order to create a single bitonic sequence.

After each of the SPUs appropriately sorts the data to create two bitonic sequences, the data is written back to main memory for the PPU to process. For each of the bitonic sequences, the PPU will perform a single bitonic merge. Then, each of the SPUs will again read n/4 consecutive elements from the main memory and perform the remaining bitonic merges before writing the data back to the main memory. SPUs with id = 0,1 will sort in ascending order, while SPUs with id = 2,3 will sort in descending order.

Now the main memory contains a single bitonic sequence. The PPU will perform two rounds of bitonic merges on the single sequence. The result can be divided into a set of four sequences, each containing n/4 elements.
Each sequence is again read by a SPU to perform the remaining bitonic merges.

1. Given an n-element sequence, each of four SPUs read n/4 consecutive elements (if the elements can fit in the local store) using DMA.

2. SPUs with id = 0 or 2 use quick sort to sort the data in ascending order, while SPUs with id = 1 or 3 will sort in descending order.

3. The result is written back to main memory via DMA transactions.

4. The PPU performs 1 round of the bitonic split on each of the two bitonic sequences created by step 2. The first bitonic sequence is split in the manner to create an ascending order while the second is split to create a descending order.

5. Each SPU reads n/4 elements and performs the remaining bitonic splits. SPUs with id = 0 or 1 sort in ascending order, while SPUs with id = 2 or 3 sort in descending order.

6. The result is written back to the main memory.

7. Now the PPU contains a single bitonic sequence on which it applies two rounds of bitonic splits to begin sorting the sequence in ascending order.

8. Each SPU again reads n/4 elements to finish the bitonic splits in ascending order.

9. The result is written back to the main memory.

10. The PPU now contains the sorted sequence.
typedef union {
    float a[4];
    vector float v;
} vec_float;
...
int offset = size / 4 / 4;
vector float min;
vector float max;
for ( i = 0; i < offset; i++ )
{
    max = fmaxf4( A[i].v, A[offset+i].v );
    min = fminf4( A[i].v, A[offset+i].v );
    A[i].v = min;
    A[i + offset].v = max;
    max = fmaxf4( A[i + 2*offset].v, A[3*offset+i].v );
    min = fminf4( A[i + 2*offset].v, A[3*offset+i].v );
    A[i + 2*offset].v = max;
    A[i + 3*offset].v = min;
}

**Algorithm 7:** Bitonic sort SIMD example (Cell)

Using SIMD instructions, the PPU and each SPU can perform four simultaneous comparisons during the splits. The Cell B.E. architecture takes advantage of 128 bit vector registers. Each SPU can perform efficient operations on four 32 bit double values at a time. Loop structures must be unrolled by at least a factor of four to take advantage of the SIMD instructions and vector registers. Algorithm 7 shows a portion of the PPU code that performs the first split as outlined in step 3.

### 4.5 Bitonic Sort using CUDA

The bitonic sorting algorithm is implemented using CUDA on an Nvidia GTX470. The algorithm is recursive in nature, but a suitable iterative
version was derived in order to parallelize it for CUDA.

```c
for ( dir = 2; dir <= N; dir = dir * 2 )
    for ( gap = dir >> 1; gap > 0; gap = gap >> 1 )
        for ( i = 0; i < N; i++ )
            {  
                cmp = i ^ gap;
                if ( cmp > i )
                    {  
                        if ( ( i & dir ) == 0 && A[i] > A[cmp] )
                            swap( A[i], A[cmp];
                        if ( ( i & dir ) != 0 && A[i] < A[cmp] )
                            swap( A[i], A[cmp];
                    }
            }
```

**Algorithm 8**: Bitonic sort iterative implementation in C

The iterative algorithm used as a basis for the CUDA implementation is shown in Figure 8. The algorithm shows an N element array A that contains a random sequence of values. In this algorithm, the variable ”gap” is the distance between the elements being compared. The variable ”gap” is also used as a bitmask to find the indices of the elements to be compared and conditionally swapped. Elements whose indices differ only in the position denoted by the gap will be compared. The variable ”dir” defines the direction in which the two values will be compared and swapped. That is, whether they are conditionally swapped to appear in an ascending or descending order. The ”dir” variable also acts as a bitmask. When the indices of two values being compared is exclusive or’d with the ”dir” bitmask, a zero value indicates an ascending order, while a non-zero value indicates a descending order.

The behavior of the algorithm is illustrated using a concrete example. Table 4.1 shows the indices of an eight element array and their correspond-
4.5. BITONIC SORT USING CUDA

<table>
<thead>
<tr>
<th>Index (decimal)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index (binary)</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
</tr>
</tbody>
</table>

Table 4.1: The indices of the array to be sorted with their binary representation.

<table>
<thead>
<tr>
<th>Index 1</th>
<th>Index 2</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>Ascending</td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
<td>Descending</td>
</tr>
<tr>
<td>0100</td>
<td>0101</td>
<td>Ascending</td>
</tr>
<tr>
<td>0110</td>
<td>0111</td>
<td>Descending</td>
</tr>
</tbody>
</table>

| Gap bit position | 0 |
| Direction bit position | 1 |

Table 4.2: First iteration of outer loop for the CUDA bitonic sort implementation.

For binary representations. Figure 4.2 above shows the first iteration of the outer loop of the algorithm for the example. Index 1 and Index 2 are the indices of the array elements to be compared and conditionally swapped. Note that in this first iteration of the outer loop, the direction bit is bit 1 (the least significant bit is bit 0). When the direction bit has a value of 0 in the elements being compared, the direction is ascending, and conversely, when the bit has a value of 1, the direction is descending. The gap bit here is in bit position 0, that is the value of gap is 1. This means that each of the indices compared only differ by bit position 0, which is the least significant bit.

Table 4.3 shows the second iteration of the outer loop. Note that if the direction bit is d, then there are d iterations of the inner loop, from d-1 to 0, inclusive. The first iteration of the inner loop compares elements that differ at bit position 1, and the second iteration compares elements that differ at bit position 0, while the direction bit remains constant throughout each iteration of the inner loop.
Table 4.3: Second iteration of outer loop for the CUDA bitonic sort implementation.

Table 4.4 shows the final iteration of the outer loop. The direction bit is now bit 3, so the inner loop will iterate 3 times in the range [2,0]. Note that each comparison uses the ascending ordering. With eight elements, only three bits are needed to represent the indices from 0 to 7 inclusive. So, the fourth most significant bit (bit 3) is zero, representing an ascending comparison and conditional swap.

Table 4.5 shows an example of each step in the bitonic sorting algorithm implemented. The heading shows the binary representations of the index of each element. The first row under the indices shows the initial sequence of numbers to be sorted. The next row shows the list after all of the conditional swaps were performed for the first iteration of the outer loop where the direction bit is bit 1 and the gap bit is bit 0 (see Table 4.2). The rows with direction bit 2 are compared and swapped with regards to the process outlined in Table 4.3, and the rows with direction bit 3 are shown in Table
4.5. BITONIC SORT USING CUDA

<table>
<thead>
<tr>
<th>Index 1</th>
<th>Index 2</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0100</td>
<td>Ascending</td>
</tr>
<tr>
<td>0001</td>
<td>0101</td>
<td>Ascending</td>
</tr>
<tr>
<td>0010</td>
<td>0110</td>
<td>Ascending</td>
</tr>
<tr>
<td>0011</td>
<td>0111</td>
<td>Ascending</td>
</tr>
</tbody>
</table>

Gap bit position: 2
Direction bit position: 3

<table>
<thead>
<tr>
<th>Index 1</th>
<th>Index 2</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0010</td>
<td>Ascending</td>
</tr>
<tr>
<td>0001</td>
<td>0011</td>
<td>Ascending</td>
</tr>
<tr>
<td>0100</td>
<td>0110</td>
<td>Ascending</td>
</tr>
<tr>
<td>0101</td>
<td>0111</td>
<td>Ascending</td>
</tr>
</tbody>
</table>

Gap bit position: 1
Direction bit position: 3

<table>
<thead>
<tr>
<th>Index 1</th>
<th>Index 2</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>Ascending</td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
<td>Ascending</td>
</tr>
<tr>
<td>0100</td>
<td>0101</td>
<td>Ascending</td>
</tr>
<tr>
<td>0110</td>
<td>0111</td>
<td>Ascending</td>
</tr>
</tbody>
</table>

Gap bit position: 0
Direction bit position: 3

Table 4.4: Third iteration of outer loop for the CUDA bitonic sort implementation.

<table>
<thead>
<tr>
<th>Index</th>
<th>Index 0000</th>
<th>Index 0001</th>
<th>Index 0010</th>
<th>Index 0011</th>
<th>Index 0100</th>
<th>Index 0101</th>
<th>Index 0110</th>
<th>Index 0111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.5: Complete concrete example of the bitonic sort algorithm

4.4.
CHAPTER 4. BITONIC SORT

4.6 Results and Conclusion

The bitonic sorting algorithm was implemented using both CUDA and the Cell Broadband Engine. The Cell implementation was run on a Playstation 3 utilizing four of the six available SPUs for reasons outlined in Section 4.4. The CUDA implementation was executed on a Nvidia GTX 470 GPU with an Intel i7 930 host machine. The baseline benchmark for the results is a single thread implementation similar to the recursive version shown in Algorithm 6 running on an Intel i7 930.

<table>
<thead>
<tr>
<th>Number of Elements</th>
<th>Cell B.E.</th>
<th>CUDA</th>
<th>Intel i7 930</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.022224</td>
<td>0.000267</td>
<td>0.001158</td>
</tr>
<tr>
<td>2048</td>
<td>0.022454</td>
<td>0.000289</td>
<td>0.002568</td>
</tr>
<tr>
<td>4096</td>
<td>0.023335</td>
<td>0.000332</td>
<td>0.005783</td>
</tr>
<tr>
<td>8192</td>
<td>0.024849</td>
<td>0.000394</td>
<td>0.012977</td>
</tr>
<tr>
<td>16384</td>
<td>0.028092</td>
<td>0.000450</td>
<td>0.020055</td>
</tr>
<tr>
<td>32768</td>
<td>0.033580</td>
<td>0.000503</td>
<td>0.063253</td>
</tr>
<tr>
<td>65536</td>
<td>0.040402</td>
<td>0.000756</td>
<td>0.128418</td>
</tr>
<tr>
<td>131072</td>
<td>0.060964</td>
<td>0.001334</td>
<td>0.233739</td>
</tr>
<tr>
<td>262144</td>
<td>0.064930</td>
<td>0.002849</td>
<td>0.465555</td>
</tr>
<tr>
<td>524288</td>
<td>0.107763</td>
<td>0.005753</td>
<td>0.958054</td>
</tr>
<tr>
<td>1048576</td>
<td>0.195652</td>
<td>0.012050</td>
<td>2.046101</td>
</tr>
</tbody>
</table>

Table 4.6: Running Times of the Bitonic Sorting Algorithm

The timing results are shown in Table 4.6. The table shows the number of floating point numbers in the sequence that is to be sorted. The input data is generated randomly. One hundred sorts are performed for each implementation and data set size, and the average running time is calculated.

The relationship between the running time and the problem size is shown in Figure 4.4. From the figure, it can be shown that as the input size grows, the CUDA implementation outperforms the others. This is due to several factors that distinguish the various architectures from each other.
4.6. RESULTS AND CONCLUSION

Figure 4.4: Graph Comparison of Bitonic Sort Running Times

The overhead for creating threads must be examined. CUDA uses very lightweight threads for the kernels. Thread blocks can be assigned to collections of cores with very little overhead. On the other hand, the Cell implementation must create a pthread for each piece of code running on a core. Each time the SPU reads data from the PPU in as shown in Section 4.4, new threads are created to perform the calculations on the SPU. Likewise, each time the SPU writes back to the PPU, there is a synchronization point. The PPU cannot prepare the data to be sent to the SPUs for the next step until all of the SPUs have completed the previous step. Thus, the PPU must wait for the slowest SPU to finish before moving on with the computation.

Another issue that arises is memory latency. The CUDA implementa-
CHAPTER 4. BITONIC SORT

tion copies the entire data set to the device memory of the GPU before computation. The GPU kernels can access the memory in the same fashion as the host code accesses the host memory. The Cell implementation requires an explicit DMA transfer of parts of the data set to its local store. The copying of the data results in a high overhead as the entire data set must be copied to and from local stores several times. In order to alleviate the problem, a buffering scheme may be used (see Section 3.2.2). In this case, a single DMA transaction still dominates the computation that can be performed after one such transfer. Adding complexity to the code, a double or triple buffering scheme may better hide the memory latency associated with the DMA transactions. The latency associated with the DMA transfer is also why the single thread Intel i7 930 version of the code has a better running time for input sizes equal to or less than 16K values. For smaller data sets, the computation performed in the SPEs is dominated by the data transfer.

The manner in which each architecture handles SIMD instructions also differs. The Cell architecture has special 128 bit SIMD registers. Four floating point values can reside in a single SIMD register, thus each Cell SIMD instruction operates on four values at a time. When using four of the SPUs, a maximum of sixteen floating point values can be processed using SIMD instructions at any given time. CUDA uses multiple threads to achieve SIMD behavior. The kernels define the instructions and each will operate on varying data. The maximum limit of simultaneous execution of instructions is limited by the number of cores contained in the GPU. Current Nvidia GPUs may contain over four hundred cores that can each process the data at the same time.
In conclusion, it appears that CUDA provides more scalability and less overhead for the bitonic sorting algorithm when compared to Cell.
Chapter 5

Matrix Multiplication

5.1 Introduction

Matrix multiplication is an integral step in many scientific computations. It has applications in the fields of physics, linear systems, computer graphics, probability and statistics among others. The parallelization of matrix multiplication may yield performance gains for numerous applications, thus it is a good candidate for manual tuning.

5.2 Matrix Multiplication Algorithm

5.2.1 Simple Matrix Multiplication

Matrix multiplication takes the form of $A \times B = C$, where $A$ and $B$ are the matrices to be multiplied and $C$ is the resulting matrix. If matrix $A$ has dimensions $m \times p$ and matrix $B$ has dimensions $p \times n$, the product is a $m \times n$ matrix.
5.2. MATRIX MULTIPLICATION ALGORITHM

for (i = 0; i < m; i++)
{
    for (j = 0; j < n; j++)
    {
        sum = 0;
        for (k = 0; k < p; k++)
        {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}

Algorithm 9: Simple matrix multiplication in C

For two dimensional matrices, the resulting product is defined by the equation $C_{i,j} = \sum_{k=1}^{p} A_{i,k}B_{k,j}$ where the subscripts $i$, $j$, and $k$ represent the position of a single element within a matrix. This computation can be performed using nested loops as shown in the pseudocode example in Algorithm 9.

5.2.2 Data Dependencies of Matrix Multiplication

The matrix multiplication algorithm holds no data dependencies in the innermost loop which would prevent parallelization. Each element of the resulting matrix is written to only once and the resulting matrix $C$ is never read as part of the computation, thus there exist no output, anti, or true data dependencies. Each iteration of the inner loop may be executed in parallel and out of order and the result will remain consistent.
5.2.3 Matrix Tiling

Tiled matrix multiplication is a decomposition of the matrix multiplication problem in order to better utilize temporal and spatial locality. Each of the input matrices \( A \) and \( B \) in the equation \( A \times B = C \) are divided into a grid of submatrices of size \( n' \times p' \) and \( p' \times m' \) respectively. The result of multiplying a submatrix from \( A \) and a submatrix from \( B \) is a matrix with dimensions \( n' \times m' \). Thus, the result matrix \( C \) contains of grid of \( n' \times m' \) submatrices.

\[
\begin{pmatrix}
A' & B' \\
C' & D'
\end{pmatrix}
\times
\begin{pmatrix}
E' & F' \\
G' & H'
\end{pmatrix}
= 
\begin{pmatrix}
A'E' + B'G' & A'F' + B'H' \\
C'E' + D'G' & C'F' + D'H'
\end{pmatrix}
\]

Figure 5.1: Example of tiled matrix multiplication

Given matrices \( A \) and \( B \) with dimensions \( n \times p \) and \( p \times m \) respectively, each submatrix of the result is defined by \( C_{sub_{i,j}} = \sum_{k=1}^{p} A_{sub_{i,k}}B_{sub_{k,j}} \) where i, j, and k represent the position of the submatrix within the given grid of tiles. Figure 5.1 shows an example of tiled matrix multiplication. The submatrices of \( A \) and \( B \) are denoted as \{\( A', H' \)\}. Each matrix is divided into four submatrices and the result matrix is a summation of matrix
5.2.4 Data Locality of Tiled Matrix Multiplication

The nested loops of the algorithm must be optimized to take advantage of the memory hierarchy through reuse. A reference is said to be reused if it is fetched or stored by more than one reference within a loop body or is used in multiple iterations. Temporal reuse happens when the same memory location is referenced at different times. Spatial reuse occurs when two uses of nearby references are used, such as consecutive memory locations. When data present within the higher levels of the memory hierarchy are reused, the reference to that location is said to display locality [18].

On cached systems, this will often lead to greater cache hit rates, thus reducing the overall memory latency. This is because the cache will read an entire line into the cache, not just one value. Thus, reading one value will cause neighboring values to be loaded into the cache which will then be used before they are evicted.

5.3 Matrix Multiplication on the Cell B.E.

5.3.1 Algorithm

The Cell implementation for the matrix multiplication problem $A \times B = C$ divides the matrices $A$, $B$, and $C$ into $64 \times 64$ submatrices. Floating point types are used for the matrix computations, thus each element is a total of four bytes. Floating point types were selected because the SPE function units are optimized for floating point operations. The size of $64 \times 64$ was
chosen because the maximum DMA transfer that can be completed in one transaction is 16KB. Each submatrix contains $64 \times 64 = 4096$ floating point numbers. Because each floating point number is represented as four bytes, the total size of a submatrix is $4096 \times 4 = 16384 = 16 \times 1024 = 16$KB.

```c
// Set the result matrix elements to zero
memset(result, 0, BLOCK_SIZE * BLOCK_SIZE * sizeof(float));

// For each block that must be read from matrix A and B
for (int block = 0; block < p / BLOCK_SIZE; p++)
{
    // Get corresponding blocks from matrix A and B
    request block (RESULT_BLOCK_X, block) from matrix A
    request block (block, RESULT_BLOCK_Y) from matrix B
    wait for DMA transaction to complete

    // Perform the matrix multiplication of the two blocks
    // and keep a running sum in the result submatrix
    for (row = 0; row < BLOCK_SIZE; row++)
        for (col = 0; col < BLOCK_SIZE; col++)
        {
            for (element = 0; element < BLOCK_SIZE; element++)
            {
            }
        }

    // Send answer back to main memory when one result
    // submatrix has been calculated
    write result submatrix (RESULT_BLOCK_X, RESULT_BLOCK_Y)
        back to main memory
}
```

**Algorithm 10**: Simple matrix multiplication SPU code for Cell

The PPU queues a thread for each of the submatrices of the result matrix. Threads are scheduled on open SPUs as they become available for computation. Each SPU will make a DMA request for each submatrix
5.3. MATRIX MULTIPLICATION ON THE CELL B.E.

5.3.2 Buffering Scheme

As outlined in Section 3.2.2, the memory latency of transferring matrix blocks from the main memory to the SPU local store may be partially hidden by using a buffering scheme. In this implementation, the SPU makes a request for the first block needed from matrix A and the first block needed from matrix B. Once an interrupt signals that the transfer is complete, the second block from each matrix is requested. While the request is being processed, the first two submatrices retrieved may be multiplied together.
// Set the result matrix elements to zero
memset(result, 0, BLOCK_SIZE * BLOCK_SIZE * sizeof(float));

// Get first corresponding blocks from matrix A and B
request block (RESULT_BLOCK_X, 0) from matrix A with id = 0
request block (0, RESULT_BLOCK_Y) from matrix B with id = 0
wait for DMA transaction to complete

// For each block that must be read from matrix A and B
for (int block = 0; block < p / BLOCK_SIZE; block++) {
    // Get next corresponding blocks from matrix A and B if they exist
    if (block + 1 < p / BLOCK_SIZE) {
        request block (RESULT_BLOCK_X, block + 1)
            from matrix A with id = id ^ 1
        request block (block + 1, RESULT_BLOCK_Y)
            from matrix B with id = id ^ 1
    }

    // Perform the matrix multiplication of the two blocks
    // and keep a running sum in the result submatrix
    for (row = 0; row < BLOCK_SIZE; row++)
        for (col = 0; col < BLOCK_SIZE; col++) {
            for (element = 0; element < BLOCK_SIZE; element++) {
                result[row][col] += A[id][row][element] * B[id][col][element];
            }
        }
    id = id ^ 1;
    wait for DMA transaction to complete
}

// Send answer back to main memory when one result
// submatrix has been calculated
write result submatrix (RESULT_BLOCK_X, RESULT_BLOCK_Y) back to main memory

Algorithm 11: Matrix multiplication SPU code with buffering for Cell
Then, the SPU will wait for the second set of blocks to finish transferring, request the third set, and continue computation on the second set. Algorithm 11 shows the single buffered version of the SPU matrix multiplication.

5.3.3 Single Instruction Multiple Data

The Cell implementation of matrix multiplication may take advantage of SIMD instructions, as outlined in Section 3.2.3. Utilizing the special 128 bit Altivec vector register set, operations can be performed on four 32 bit floating point values concurrently. The SPU needs only to add and multiply values, which are handled by the intrinsic functions spu_add() and sup_mul() respectively. By unrolling the innermost loop of Algorithm 11 by a factor of four, the SIMD instructions may be used directly. Algorithm 12 shows the updated code that uses SIMD instructions.

Loop unrolling will also have an impact on the running time of the implementation. The Cell SPUs do not contain a sophisticated branch prediction mechanism. Reducing the number of branch instructions can greatly increase performance. However, there does exist a drawback to unrolling the loop by a large factor. The vectors containing the data being used in computation may require a buffer so that the code does not access elements outside of the allocated region of the vector. Unrolling the loop by too large of a factor may also cause extraneous operations on the buffered regions of data. This is why the given implementation unrolls by a factor of four, just enough to allow for SIMD instructions.
// Set the result matrix elements to zero
memset(result, 0, BLOCK_SIZE * BLOCK_SIZE * sizeof(float));

// Get first corresponding blocks from matrix A and B
request block (RESULT_BLOCK_X, 0) from matrix A with id = 0
request block (0, RESULT_BLOCK_Y) from matrix B with id = 0
wait for DMA transaction to complete

// For each block that must be read from matrix A and B
for (int block = 0; block < p / BLOCK_SIZE; block++)
{
    // Get next corresponding blocks from matrix A and B if they exist
    if (block + 1 < p / BLOCK_SIZE)
    {
        request block (RESULT_BLOCK_X, block + 1) from matrix A with id = id ^ 1
        request block (block + 1, RESULT_BLOCK_Y) from matrix B with id = id ^ 1
    }

    // Perform the matrix multiplication of the two blocks
    // and keep a running sum in the result submatrix
    for (row = 0; row < BLOCK_SIZE; row++)
        for (col = 0; col < BLOCK_SIZE; col++)
        {
            for (element = 0; element < BLOCK_SIZE / 4; element++)
            {
                result[row][col] = spu_add(result[row][col],
                                            spu_mul(A[id][row][element], B[id][col][element]));
            }
        }

    id = id ^ 1;
    wait for DMA transaction to complete
}

// Send answer back to main memory when one result
// submatrix has been calculated
write result submatrix (RESULT_BLOCK_X, RESULT_BLOCK_Y) back to main memory

Algorithm 12: Matrix multiplication SPU code with buffering for Cell
5.4 Matrix Multiplication using CUDA

5.4.1 Algorithm

The CUDA implementation presented for the matrix multiplication problem $A \times B = C$ divides the matrices $A$, $B$ and $C$ into $16 \times 16$ submatrices. Each submatrix of matrix $A$ defines a thread block. Thus, a total of 256 threads are created in each thread block.

Each thread is responsible for loading one element of $A$ and one element of $B$ into shared memory. This will reduce the number of global memory accesses and thus reduce the memory latency. A synchronization point is used after the loads to ensure that the correct data resides in shared memory for the submatrices of $A$ and $B$.

5.4.2 Shared Memory

Shared memory is a mechanism which allows cooperation between various threads belonging to the same thread block. If multiple threads in the same block access the same global memory space, then it is beneficial to store the data in a shared memory space. Only one global memory access is required to store the data as opposed to one global memory access for each thread reading the data. The shared memory space is physically closer to the cores and is substantially smaller than the global device space, reducing the memory access latency significantly.

Algorithm 13 shows a simple CUDA matrix multiplication kernel. As shown in Figure 5.3, each thread calculates a single element in a single tile of the result matrix $C$ using an entire row of a block in matrix $A$ and an entire row of a block in matrix $B$. Each thread block computes exactly one
```
__global__ void matmul(float *a, float* b, float *c, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0f;

    for ( int t = 0; t < NUM_TILES; t++ )
    {
        for (int i = 0; i < TILE_DIM; i++)
        {
            sum += a[t*TILE_DIM + row*N+i] * b[t*TILE_DIM + col*N + i];
        }
    }
    c[row*N+col] = sum;
}
```

**Algorithm 13:** CUDA matrix multiplication not exploiting shared memory (A x B = C)

![CUDA matrix multiplication diagram not exploiting shared memory](image)

Figure 5.3: CUDA matrix multiplication diagram not exploiting shared memory

of the result matrix tiles. Note that in this code, if the block size of matrix A is i x j and the block size of matrix B is k x l, then the tiles of matrix A
and B are read k and j times, respectively.

```c
__global__ void matmul(float *a, float *b, float *c)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM], bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for ( int t = 0; t < NUM_TILES; t++ )
    {
        aTile[threadIdx.y][threadIdx.x] = a[t*TILE_DIM + row*N + threadIdx.x];
        bTile[threadIdx.y][threadIdx.x] = b[t*TILE_DIM + col*N + threadIdx.x];
        __syncthreads();
        for (int i = 0; i < TILE_DIM; i++)
        {
            sum += aTile[threadIdx.y][i] * bTile[i][threadIdx.x];
        }
    }
    c[row*N+col] = sum;
}
```

**Algorithm 14:** CUDA matrix multiplication exploiting shared memory

When threads in a block, or more specifically a half-warp (every 16 threads), access the same memory locations repeatedly, it is beneficial to move the data into shared memory. Algorithm 14 shows an example of exploiting shared memory. Each element in a tile of A and B are read exactly once from global memory. Each thread block will compute an entire tile of the result matrix. In Algorithm 14, the thread that calculates the resulting element at (i,j) in the result matrix also loads the (i,j) element from the matrix A tile and matrix B tile to shared memory. After the requests
to load the values into shared memory, there must exist a synchronization point before computation may resume. The \_syncthreads() call will block until all threads in the block reach this point. When the synchronization point is passed, it is guaranteed that all threads have read their respective values from both input matrices. The threads no longer have a need to read from global memory because all of the pertinent values have already been loaded into the faster shared memory space.

5.4.3 Memory Coalescing

The code shown in Algorithm 14 takes advantage of memory coalescing. Consider the reading of matrix A into shared memory. Each thread in a half-warp reads 32 bits, a floating point value. As long as the tile dimension is chosen appropriately as a multiple of 16, the beginning of each row read will be aligned by 64 bytes. Thus, the global memory read can be coalesced into a single read operation of 64 bytes rather than 16 read operations of 4 bytes, reducing the overhead and memory latency.

5.5 Results and Conclusion

The running time of both the Cell implementation and the CUDA implementation of matrix multiplication have been recorded. The Cell implementation is running on a Playstation 3 with two of the eight SPUs disabled. The CUDA implementation is running on a Nvidia GTX 470 graphics processor with an Intel i7 930 host machine.

For simplicity, the implementations assume square matrices whose dimensions are a power of two. This allows for a fixed optimal block size and
5.5. RESULTS AND CONCLUSION

alleviates the need for padding the data with extraneous computations on padded data. As a baseline for comparison, a single threaded implementation based on Algorithm 9 was executed on an Intel i7 930.

Figure 5.4: Graph of running time of matrix multiplication implementations

<table>
<thead>
<tr>
<th>Matrix Dimension</th>
<th>Cell B.E.</th>
<th>CUDA</th>
<th>Intel i7 930</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.083328</td>
<td>0.000228</td>
<td>0.355800</td>
</tr>
<tr>
<td>512</td>
<td>0.690601</td>
<td>0.001647</td>
<td>2.269023</td>
</tr>
<tr>
<td>1024</td>
<td>5.819833</td>
<td>0.012804</td>
<td>21.6795</td>
</tr>
<tr>
<td>2048</td>
<td>82.054369</td>
<td>0.042707</td>
<td>147.209629</td>
</tr>
</tbody>
</table>

Table 5.1: Table of running times (in seconds) of the matrix multiplication implementations

Figure 5.4 is a table displaying the running times of each implementation. Both of the parallel implementations have better performance than
the baseline single threaded matrix multiplication. Note that the CUDA implementation out performs the Cell version. There are several contributing factors for this.

There exists a greater overhead associated with data transfer in the Cell version compared to the CUDA version. Although the memory latency of the DMA transactions are partially hidden by overlapping them with computation, the memory copy from main memory to the local store is expensive. The CUDA implementation also benefits from using the shared memory as a software controlled cache. The shared memory region is physically close to the streaming multiprocessor that computes a submatrix of the result. Each core loads a single element from each of the input matrices concurrently. Rather than reading an entire block, each thread needs to read only two elements for each matrix tile.

The CUDA implementation also allows for more arithmetic operations to be performed simultaneously. Optimally, each of the 448 cores can perform one addition or multiplication at the same time. In contrast, the Cell Broadband Engine with six SPEs using floating point SIMD instructions can perform a maximum of $6 \times 4 = 24$ operations concurrently.

The Cell processor SPEs contains more powerful floating point function units than the GPU cores and has a lower latency for memory access. In conclusion, the GPU is more suited to problems that need to communicate large amounts of data, while the Cell processor may be better utilized with a higher computational load in which the DMA transactions may be completely hidden inside.
Ordinary Differential Equation Solver

6.1 Introduction

Ordinary differential equations (ODEs) are used in a wide variety of scientific applications to describe the behavior of systems in areas such as fluid dynamics, biology, and chemical kinetics. Often an analytical solution to the behaviour of a system over time is unknown, and a numerical method is required to show the actual behavior of the system. Presented is a method of solving initial value problems of stiff, coupled ordinary differential equations using the backward differentiation formula (BDF).

Initial value ODE problems are of the form \( y' = f(t, y), y(t_0) = y_0 \) where \( y \in \mathbb{R}^N \) and \( y' = \frac{dy}{dt} \). A system of N ordinary differential equations is shown and initial values at time \( t_0 \) are given.
6.1.1 LU Decomposition and Cholesky Factorization

In order to solve the ODEs, methods such as LU decomposition and Cholesky factorization are used to solve the linear equations $Ax = b$, where $A$ is a symmetric, positive definite matrix.

The factorization of the matrix $A$ is an important step in the solution of the linear system. The form of the LUP decomposition of an $n \times n$ matrix is of the form $PA = LU$, where $L$ is an $n \times n$ lower triangle matrix, $U$ is an $n \times n$ upper triangle matrix, and $P$ is a permutation matrix. Cholesky factorization is used to compute the lower triangle matrix $L$, such that $A = LL^T$.

6.1.2 Backward Differentiation Formula

BDF method is a multi-step implicit algorithm to solve a set of differential equations, also known as Gears method. BDF method will use an iterative Newton approximation method to converge on a solution.

1. $rhs = (w_0 - x_0) - hf(t_0 + h, w_0)$
2. $rhs = (w_0 - (4/3)x_0 + (1/3)x_1) - (2/3)hf(t_0 + h, w_0)$
3. $J' = I - hJ(t_0 + h, w_0)$
4. $J' = I - (2/3)hJ(t_0 + h, w_0)$
5. $J' = rhs$

The BDF2 algorithm utilizes the Jacobian matrix formed from the partial differentiation of the equation system. The first step in the BDF2 algorithm must be computed using a one step version of BDF. First, the right hand side ($rhs$) of the equation is calculated for the initial one step problem using equation (1) and subsequent steps use the two step method.
shown in equation (2). In both (1) and (2), w₀ is the initial guess for the solution vector (which is set to x₀ at the beginning of the iterations), x₀ and x₁ are the previously calculated vector of values from the previous two steps.

After the right hand side of the equation is calculated, Jacobian matrix J is found based on equation (3) in the one step method and equation (4) in the two step method.

The next step in the BDF algorithm is to find a correction value for the initial guess. This is done by solving the system of equations as shown in equation (5).

LU decomposition is used to find a correction vector dw. The correction vector is then subtracted from the initial guess vector w₀. This process iterates until the relative change of w₀ is below a set threshold. That is, for each element in the vector dw, dw / threshold, where threshold is the minimum accepted relative change in w₀.

The following equations define the OREGO system used to express chemical reactions:

\[
\begin{align*}
    x'_0 &= 77.27 \times (x_1 + x_0 \times (18.375 \times 0.000001 \times x_0 x_1)) \\
    x'_1 &= (x_2 - (1 + x_0) \times x_1) / 77.27 \\
    x'_2 &= 0.161 \times (x_0 x_2)
\end{align*}
\]

The Jacobian is derived from the system of equations where each row is the partial derivative of one of the equations with respect to each of the independent variables. Row one, two, and three correspond to equation one, two, and three respectively. Column one is the partial derivative with respect to x₀, two with respect to x₁ and three with respect to x₂.

This example will assume a step size of 0.01. The initial values for the
OREGO system shown are as follows:

\[ x'_0 = 1 \]
\[ x'_1 = 2 \]
\[ x'_2 = 3 \]

The first step in performing the BDF algorithm is calculating the right hand side of the equation that will be solved.

\[ rhs = (w_0 - x_0) - h * f(t + h, w_0) \]
\[ rhs = \langle 1, 2, 3 \rangle - \langle 1, 2, 3 \rangle - 0.01 * \langle 77.269, -0.013, -0.322 \rangle \]
\[ rhs = \langle -0.773, 0.000, 0.003 \rangle \]

The next step is to evaluate the Jacobian using the previously known values for the independent variables.

\[
J = \begin{bmatrix}
-77.271 & 0 & 0 \\
-0.026 & -0.026 & 0.013 \\
0.161 & 0 & -0.161 \\
\end{bmatrix}
\]

After the Jacobian has been evaluated, \( J' \) can be calculated.

\[
J' = I - \left( \frac{2}{3} \right) * h * J
\]
\[
J' = \begin{bmatrix}
1.773 & 0 & 0 \\
0 & 1 & 0 \\
-0.002 & 0 & 1.002 \\
\end{bmatrix}
\]

LU decomposition is then used to solve the system of equations \( J' \times dw = rhs \).

\[ dw = \langle 0.773, 0.000, -0.003 \rangle \]

The initial guess is then updated ( recall that the initial guess here is the initial value for the system ).
\[ w_0 = w_0 - dw = <0.227, 2.000, 2.997> \]

The process for finding \( dw \) and updating \( w_0 \) is repeated until all of the elements of \( dw \) fall within a provided tolerance level. That is, when the rate of change of the guess correction slows sufficiently and converges on a result. Steps of BDF2 are similar, but interpolate using two previously calculated values, rather than one.

### 6.2 Cell Broadband Engine Implementation

The Cell implementation of the BDF algorithm divides work between the six available SPUs in the Playstation 3. The difficulty in parallelizing the ODE solver for Cell is that the steps required for computation rely on previously calculated values, leading to a sequential behavior between steps. Because of this, each SPU will solve one complete initial value problem. It is, however, possible to parallelize steps such as the LU decomposition, vector, and matrix operations that are used in a single step of the algorithm. The problem is that most systems contain a small number of equations (in the order of tens). The associated overhead of the DMA transactions required to split such operations over the SPUs greatly outweighs any benefits of parallelizing the operation.

All of the ODEs to be solved, as well as the associated Jacobian matrix and initial values are loaded into the local store of each available SPU. When the PPU initiates the thread to be run on the SPU, it needs only to communicate the identification number of the system that needs to be solved and this is done as an argument to the thread invocation on the SPU. Thus, there are no explicit DMA transactions required within the
The matrix and vector operations required to perform the backward differentiation are implemented using SIMD instructions. Double precision numbers are used in the calculations for accuracy, thus up to two such 64 bit numbers may reside in a single 128 bit vector register. Loops are unrolled by a factor of two to accomadate the SIMD instructions.

6.3 Results and Conclusions

The ordinary differential equation solver was tested using a set of stiff and non-stiff ODEs. The equation sets used include the Van der Pol oscillator, E5, HIRES, OREGO, ROBER, and Lotka-volterra.

The Van der Pol oscillator models a non-linearly dampened harmonic oscillator. It is represented by a non-stiff system containing two equations. The Van der Pol oscillator has applications in both physical and biological sciences, modelling action potentials of neurons and seismic activities around tectonic plate faults among other systems [16, 7].

The E5, HIRES, OREGO, and ROBER systems all define the behavior of chemical reactions. HIRES models high irradiance response and the OREGO system models the Belousov-Zhabotinskii reaction. E5, HIRES, OREGO, and ROBER are stiff sets of four, eight, three, and three equations respectively.

The Lotka-volterra system is a set of two coupled, non-stiff ODEs. It is mainly used in biology to model the population of two species in a predator
and prey relationship.

<table>
<thead>
<tr>
<th>System</th>
<th>$h = 0.00001$</th>
<th>$h = 0.000005$</th>
<th>$h = 0.000001$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Van der Pol</td>
<td>155</td>
<td>270</td>
<td>1255</td>
</tr>
<tr>
<td>Orego</td>
<td>226</td>
<td>435</td>
<td>1983</td>
</tr>
<tr>
<td>E5</td>
<td>135</td>
<td>271</td>
<td>1354</td>
</tr>
<tr>
<td>HIRES</td>
<td>1597</td>
<td>3053</td>
<td>13806</td>
</tr>
<tr>
<td>ROBER</td>
<td>124</td>
<td>248</td>
<td>1240</td>
</tr>
<tr>
<td>Lokta-Volterra</td>
<td>124</td>
<td>223</td>
<td>928</td>
</tr>
</tbody>
</table>

Table 6.1: Running time (seconds) of various initial value ODE problems on Cell with variable step size $h$

Table 6.1 shows the running time, in seconds, of solutions to various ODE initial value problems of the solver written for Cell. The systems used an error tolerance of $10^{-9}$ and the step size, $h$, was varied. The running time is the time a system takes to complete running on a single SPE over the time interval from 0 to 100. The variation in running time is due to the size of the system as well as the stiffness. Systems with more equations have larger vectors representing their values as well as larger Jacobian matrices, resulting in more steps for matrix and vector operations. Stiffer systems may require more applications of the correction vector in order for the relative error to fall within tolerance levels.

In practice, there are two scenarios for the system definitions that will be assigned to SPEs. If each SPE is required to solve the same system with different initial values, then six such systems may be solved in roughly the time that it takes to solve once such system on an SPE. If each SPE is solving different initial value problems, then the time to complete the set of problems is equal to the maximum running time amongst all SPEs (assuming a maximum of six systems). If more systems are defined than SPEs, they will be scheduled sequentially in the order that they are defined.
This is not optimal, as the order in which the systems are scheduled has an impact on the total running time. Ideally, a scheduling algorithm, based on the size of the system and the range over which it must be computing, would be used to intelligently order the systems to reduce the overall running time. To reduce the complexity associated with such a scheduler, a maximum of six systems (the number of SPEs in the Playstation 3) is assumed for each run.

Figure 6.1: The solution of each equation of an OREGO initial value problem over the range of time units from 0 to 1000

Figure 6.1 shows the behavior of each of the OREGO equations from an initial state over a period of time from 0 to 1000. Each equation represents a concentration of a chemical compound found in a reaction. The reaction
causes the solution to change colors. The spikes and troughs of the graph of each equation are correlated to this color change.
Chapter 7

Conclusions and Future Work

This thesis presented several parallelizable applications that were manually tuned for the Cell Broadband Architecture as well as the Nvidia CUDA enabled graphics processors. Sorting is a fundamental problem in the realm of computer science and has applications in database systems as well as other algorithms such as binary searches. The matrix multiplication kernel has been studied extensively due to its appearance in numerous algorithms related to linear algebra and the physical sciences. The solving of initial value problems from coupled ordinary differential equations is important for chemical kinetics, electronics, and modelling other interactions. The performance gains of these computation kernels may be utilized in many applications.

Intelligent problem decomposition and distribution with respect to the available architecture and programming interface features is key to improving the performance of the kernels. The bitonic sort and tiled matrix multiplication algorithms used were selected because they are, by nature, easily parallelizable. Matrix multiplication can be written in a way that there
exist no relevant data dependencies between any of the calculations performed. Likewise, there exist no dependencies between the conditional swap operations performed within each round of the bitonic sort. The ordinary differential equation initial value problem solver was selected as a problem because the author was contracted by Wright Patterson Airforce Base to complete a parallel implementation. The target application lies within the realm of fluid dynamics, where a system is evaluated based on varying initial states.

Several levels of parallelism were exploited in each of the implementations. Data level parallelism in the form of SIMD instructions made possible by loop unrolling was used in each of the Cell implementations. Another exploitation of data parallelism is in the division of independent workloads into threads that were scheduled on each core. The initial value problem has a coarser grain task level parallelization in the distribution of separate systems to separate cores.

Utilizing the available memory hierarchy and reducing memory latencies is also an important factor in the performance gain of the parallel solutions. The matrix multiplication implementation in CUDA uses the shared memory space as a software controlled cache between threads of a warp to lower memory latency by reducing the number of device memory read operations. Both of the CUDA implementations also lower memory read latencies via memory coalescing. The memory access patterns allow for efficient reads within a half-warp by combining contiguous reads from multiple threads into a single read operation. The Cell implementations of bitonic sort and matrix multiplication attempt to hide the overhead associated with DMA transfers by partially overlapping them with computation
through the buffered prefetching of data.

The performance gains of parallel implementations of both bitonic sort and matrix multiplication over the single threaded, sequential version has been shown. The CUDA versions running on the Nvidia GTX 470 graphics processor and Intel i7 930 host machine has out performed the Cell Broadband Engine version running on the Sony Playstation 3. There are two reasons for the difference in performance. The GTX 470 contains a total of 448 simple execution cores, while the Playstation 3 has six SPEs enabled to share a workload. Although the SPEs are more sophisticated cores, the sheer number of cores available in the graphics processor allow for more simultaneous instruction streams that are taken advantage of by highly parallelizable algorithms. Perhaps algorithms with more sequential steps may benefit more from the Cell architecture and warrants further research.

Because the graphics processor architecture, with hundreds of cores, outperformed the Cell architecture with eight or fewer cores, the next logical step from multicore may be manycore platforms. Processors with cores numbering in the thousands presents new challenges to the programmer and architecture designers. Memory hierarchies, communication between threads, and synchronization become more complex as the number of cores increases. Scalability is also an issue. This thesis presents a manual tuning of algorithms for a specific architecture, but the solutions should also be able to be scalable as more cores are added to the chip. CUDA addresses this issue through the logical definition of thread blocks that are queued and scheduled on resources as they become available, allowing the program to scale with the specific graphics processor being used. Assessing
the scalability of the algorithms is another area of possible future research.
Bibliography


ftp://download.intel.com/museum/Moores_Law/Articles-
Press_Releases/Gordon_Moore_1965_Article.pdf. [cited at p. 3]

