2011

Parallel Sorting on Multi-core Architecture

Wei Wang
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Sciences Commons

Repository Citation
Wang, Wei, "Parallel Sorting on Multi-core Architecture" (2011). Browse all Theses and Dissertations. 484.
https://corescholar.libraries.wright.edu/etd_all/484

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@wwwlibraries.wright.edu, library-corescholar@wright.edu.
Parallel Sorting on Multi-core Architecture

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

By

Wei Wang
B.S., Zhengzhou University, 2007

2011
Wright State University

Meilin Liu, Ph. D.
Thesis Director

Mateen Rizki, Ph.D.
Department Chair

Committee on
Final Examination

Meilin Liu, Ph. D.

Jack Jean, Ph. D.

T.K. Prasad, Ph. D.

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


With the limitations given by the power consumption (power wall), memory wall and the instruction level parallelism, the computing industry has turned its direction to multi-core architectures. Nowadays, the multi-core and many-core architectures are becoming the trend of the processor design. But how to exploit these architectures is the primary challenge for the research community. To take advantage of the multi-core architectures, the software design has undergone fundamental changes.

Sorting is a fundamental, important problem in computer science. It is utilized in many applications such as databases and search engines. In this thesis, we will investigate and auto-tune two parallel sorting algorithms, i.e., radix sort and sample sort on two parallel architectures, the many-core nVIDIA CUDA enabled graphics processors, and the multi-core Cell Broadband Engine. We redesign and manually tune these two parallel sorting algorithms to take advantage of multiple-level parallelism simultaneously, i.e., thread level parallelism, loop level parallelism, data level parallelism (SIMD instructions). At the same time, we try to take advantage of the high-speed shared memory. The experimental results showed that the parallel implementation of these two sorting algorithms on these two multi-core architectures achieved significant performance improvement compared to the corresponding sequential version.
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 1. INTRODUCTION ........................................ 1</td>
</tr>
<tr>
<td>1.1 Motivation .................................................... 1</td>
</tr>
<tr>
<td>1.2 Sorting ......................................................... 5</td>
</tr>
<tr>
<td>1.3 Problem Statement ............................................. 8</td>
</tr>
<tr>
<td>1.4 Organization .................................................. 9</td>
</tr>
<tr>
<td>CHAPTER 2. BASIC CONCEPTS ABOUT PARALLEL COMPUTING ...... 10</td>
</tr>
<tr>
<td>2.1 Introduction .................................................. 10</td>
</tr>
<tr>
<td>2.2 Data Dependence ............................................... 10</td>
</tr>
<tr>
<td>2.3 Locality of Reference ......................................... 13</td>
</tr>
<tr>
<td>2.4 Granularity .................................................... 14</td>
</tr>
<tr>
<td>2.4.1 Instruction-Level Parallelism ............................. 15</td>
</tr>
<tr>
<td>2.4.2 Data-Level Parallelism ..................................... 16</td>
</tr>
<tr>
<td>2.4.3 Thread-Level Parallelism ................................... 17</td>
</tr>
<tr>
<td>2.5 Amdahl’s Law .................................................. 19</td>
</tr>
<tr>
<td>2.6 Memory Organization .......................................... 20</td>
</tr>
<tr>
<td>2.6.1 Shared Memory ............................................... 20</td>
</tr>
<tr>
<td>2.6.2 Distributed Memory ......................................... 21</td>
</tr>
<tr>
<td>CHAPTER 3. THE ARCHITECTURE OF CELL BROADBAND ENGINE ... 23</td>
</tr>
<tr>
<td>3.1 Introduction .................................................... 23</td>
</tr>
<tr>
<td>3.2 Decreasing the Impact of Serial-performance Walls .......... 24</td>
</tr>
<tr>
<td>3.3 Architecture Overview ........................................ 25</td>
</tr>
<tr>
<td>3.4 PowerPC Processor Element ..................................... 26</td>
</tr>
<tr>
<td>3.4.1 PowerPC Processor Unit .................................... 26</td>
</tr>
<tr>
<td>3.4.2 PowerPC Processor Storage Subsystem ..................... 28</td>
</tr>
<tr>
<td>3.5 Synergistic Processor Elements ............................... 28</td>
</tr>
</tbody>
</table>
3.5.1 Synergistic Processor Unit ........................................... 28
3.5.2 Memory Flow Controller ............................................. 29
3.6 The Element Interconnect Bus ........................................ 30
3.7 The Memory interface Controller ..................................... 30
3.8 Cell Broadband Engine Interface Unit ............................... 31
3.9 Programming Model for the PPE and SPEs .......................... 31
  3.9.1 Language-extension Differences Between PPE and SPE ...... 33
  3.9.2 Communication between the PPE and SPEs ................... 34

CHAPTER 4. THE ARCHITECTURE OF GPU ............................... 35
4.1 Introduction .................................................................. 35
4.2 An Overview of the GPU Architecture .............................. 36
  4.2.1 Streaming Processor Array ....................................... 37
  4.2.2 Streaming Multi-processor ....................................... 37
  4.2.3 Streaming Processor ............................................... 39
4.3 Memory Hierarchy ....................................................... 39
  4.3.1 Global Memory ...................................................... 39
  4.3.2 Shared Memory ...................................................... 40
  4.3.3 Local Memory ....................................................... 40
4.4 Programming Model for CUDA GPU ............................... 40
  4.4.1 CUDA Programming Paradigm .................................. 40
  4.4.2 Single Instruction Multiple Threads and Warp Divergence .. 42

CHAPTER 5. PARALLEL RADIX SORT ................................. 45
5.1 The Prefix Sum Primitives ............................................. 45
  5.1.1 Sequential Scan ..................................................... 46
  5.1.2 An Inefficient Parallel Scan ...................................... 47
  5.1.3 An Efficient Parallel Scan ........................................ 48
5.2 Radix Sort ................................................................... 50
  5.2.1 Serial Radix Sort ..................................................... 51
  5.2.2 CUDA Parallel Radix Sort ....................................... 53
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Number</th>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basic Structure of UMA System</td>
<td>21</td>
</tr>
<tr>
<td>2.2</td>
<td>Basic Structure of NUMA System</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>Basic Architecture of Distributed Memory System</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview of CBEA</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>PPE Structure Diagram</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Structure of SPE</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Vector Add Operation</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>The G80 GPU Architecture</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Texture/Processor Cluster (TPC)</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>CUDA Memory Hierarchy</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Warp Scheduling</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>An Illustration of the Inefficient Scan</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>An Illustration of the Up-Sweep Phase of the Efficient Scan</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>An Illustration of the Down-Sweep Phase of the Efficient Scan</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>The Operation of Radix Sort on 6 4-digit Numbers</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>The Layout of the Histogram Table</td>
<td>55</td>
</tr>
<tr>
<td>5.6</td>
<td>The \textit{Split} Operation Based on the Least Significant Bit</td>
<td>57</td>
</tr>
<tr>
<td>5.7</td>
<td>The Extended Parallel Scan Algorithm for a Large Array</td>
<td>58</td>
</tr>
<tr>
<td>5.8</td>
<td>The Comparison of Radix Sorting Algorithms</td>
<td>61</td>
</tr>
<tr>
<td>6.1</td>
<td>An Illustration of the sample Sort</td>
<td>65</td>
</tr>
<tr>
<td>6.2</td>
<td>The Comparison of Parallel Sample Sort and the Sequential Quick Sort</td>
<td>74</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 Vector Data Type Supported by PPE and SPE ......................... 33

5.1 The Running Time for Transferring $2^{20}$ Keys to/from Shared Memory . 54
5.2 Running Time of the Radix Sorting Algorithm ......................... 60

6.1 Running Time of the Parallel Sample Sort and Serial Quick Sort ...... 74
ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Meilin Liu, for her ideas, advice, and constant encouragement. Without her guidance and advice, this body of work would not have been possible.

I also would like to thank my thesis committee members: Drs. Jack Jean and T.K. Prasad for taking their invaluable time to serve on my committee and help in improving this document.

Next, I would like to thank Mark Haines for correcting this thesis.

I also wish to thank my friends in Wright State University for their help and support throughout my master study.

August 2011
CHAPTER 1
INTRODUCTION

1.1 Motivation

On 19th April 1965, Gordon E. Moore published a paper with the title "Cramming more components onto integrated circuits" in "Electronic Magazine" [14]. In this paper, he stated that "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000 [21].” The author presented that he believed such a large circuit can be built on a single wafer [21]. In the revised version of that paper, the author stated that transistor density on integrated circuits doubles every 18 months. Although Moore is adamant that he did not use the term “18 months”, we still refer it as Moore’s law, which is a very significant principle affecting the development of computer industry. Under the guidance of Moore’s law, the computer industry has been witnessing tremendous advances in microprocessor technology over the past couple of decades. Clock cycle speed of processors has been improved from about 4.77MHz in the 1980s to about 2GHz in the 2000s.

While Moore’s law continues to be a reliable predictor in the computer industry for the coming decade, we can expect little improvement in system performance of
the general purpose CPUs if we only double the number of transistors integrated on a chip to increase clock rate. The first reason for this is power dissipation. As we know, power dissipation in an electronic device is proportional to the clock rate. While the clock frequency has increased by a factor of 4000 in the last decade, the technology of power dissipating has almost reached physical limitation now. It is impossible to increase the clock frequency without taking cooling techniques into account. Therefore, a significant increment of the clock rate will not be achieved without a breakthrough in the technology of power dissipation. This is the power wall which impedes the improvement of system performance.

The gap between the performance improvement of processors and the memory systems is another factor impacting system performance. Because the performance improvement of the memory systems can not keep up with the growth of CPU speed, which has introduced a large gap between them. In order to reduce the average memory access time, the cache is introduced to the memory hierarchy. For modern computer architecture, increasing the cache size and utilizing multi-level cache are two primary methods to reduce the cache miss rate. However, even with the help of cache, we can not prevent memory performance improvement from lagging behind the processor speed yet.

System performance can be improved not only from the growth of the clock frequency, but also from the instruction level parallelism (ILP). The primary benefit we can obtain from ILP is that the performance of a sequential program will be increased without any modification. However, the speedup of system performance by utilizing ILP is usually stalled. The data dependences are the main factor which prevents the consecutive instructions from being executed in parallel. And the conditional branch is the second. For the branch instruction, its execution path can not be decided before the
result of the current branch is known. But in order to achieve ILP, the target instruction must be predicted and \textit{speculatively executed} before the correct result is produced. Since the branch is very difficult to predict successfully, it is very likely that the processor speculatively executes an incorrect branch. Because the penalty of taking the incorrect branch is significant, the improvement of system performance by using ILP can not be guaranteed.

David Patterson from U.C Berkeley has a formula which summarizes the problem of serial performance: The power wall + the memory wall + the ILP wall = the wall for system performance \cite{12}. While Moore’s law will continue to guarantee that more gates can be integrated into a chip, the improvement of system performance obtaining from increasing clock frequency has reached the limitation. Therefore, the hardware engineers have to veer their focus to parallel architecture to keep making progress. Rather than aggregating more transistors within a single processor, computer architects use more possible gates to create additional independent processors in the chip.

The nVIDIA CUDA enabled graphics processors, as well as the Sony Cell Broadband Engine architecture are two well-designed parallel platforms. CUDA is a software and hardware architecture which allows programmers to code in high level languages without considering the graphics interfaces of the GPU. Instead of using APIs provided by GPU, a programmer can write a general-purpose application with CUDA C/C++. Compared with general-purpose CPU, the programming model for CUDA is single program multiple data (SPMD), in which each instruction of a program will be executed by multiple threads in parallel. The Cell Broadband Engine architecture is a single-chip heterogeneous multiprocessor system cooperating with both shared and distributed memory. It has nine processors: One PowerPC Processor Ele-
ment, which is optimized for scheduling tasks and running operating system, and eight Synergistic Processor Elements, which are optimized for executing compute-intensive tasks.

Parallelism introduces much more complexity into the process of programming. In parallel programming, the most important aspect is to exploit concurrency, which means decomposing a problem into sub-problems which can be executed in parallel. Therefore, how to find parallelism in a sequence program, and how to expose the parallelism are the major problems. In order to implement concurrency, it is important to write code to decompose problems and exploit parallelism in the running time. The two decomposition strategies, task decomposition and data decomposition, are usually used to achieve this goal. Intuitively, this pattern usually maps similar operations onto different elements of data arrays.

Parallelism brings unique challenges to programmers. It is very common that concurrent tasks which constitute the problems involve dependences which must be identified and managed appropriately. The execution of tasks must follow program order. A good parallel programmer must rule out all the non-deterministic issues. Structuring correct parallel programs may take considerable effort from programmers when compared to writing sequential programs.

Even though a parallel program is safe, it is possible that the performance improvement realized from exploiting concurrency is not as good as expected. Under these circumstances, we must make sure that the overhead of incurred parallelism does not overwhelm the program’s runtime. Moreover, distributing tasks in a balanced way is not straightforward. Therefore, the performance of one parallel program is largely dependent on how well the carefully designed parallel algorithm exploits the underlying parallel platform.
1.2 Sorting

Sorting plays a very important role in the area of computer science, because it not only can be used in many applications, but also has theoretical significance. It has been applied in numerous fields such as database management systems and knowledge base management systems. And for the architecture of modern advanced computers, the ratio of the sorting time to the total CPU time is very significant. Someone has conducted a survey, and concluded that general-purpose computers can spend to 75% total CPU time on sorting [10]. When computing or data processing is executed, it usually involves the sorting problem. Programmers need to select different sorting algorithms to obtain the expected performance on various computer architectures. Moreover, for the well-known algorithms, sorting is a very interesting topic deserving studying and analyzing. Thus, it is critical for students whose major is computer science to best understand the various sorting algorithms.

Generally speaking, sorting is a rearrangement process conducted in accordance with the increasing or decreasing relationship between keys’ values of records, which changes the original set of records with arbitrary order into a set of records with monotone increase or monotone decrease order based on the values of keys. Suppose a file with n records is represented as \( \{R_1, R_2, ..., R_n\} \), and the keys’ values corresponding to each record are \( \{k_1, k_2, ..., k_n\} \), then we can determine a permutation \( t(1), t(2), ..., t(n) \), such that the order of the keys’ values satisfy the following requirement:

\[
k_{t(1)} \leq k_{t(2)} \leq ... \leq k_{t(n)}
\]

or...
the corresponding file will become \{R_{t(1)}, R_{t(2)}, ..., R_{t(n)}\}, which is permuted in terms of the order of keys’ values. We call this type of operation sorting, and state that the previous relation is an ascending order, and the later order is a descending order. In practice, we can simply consider a sorting operation as a process in which a sequence without order is transformed into a sequence with order based on a given value.

Sorting, also known as classification, is a very efficient way to improve timing performance for the search operation. If the search is executed on an unordered sequence with the size of \(n\), its time complexity is \(O(n)\). But if this sequence is in order, its time complexity will be decreased to \(O(\log_2^n)\).

Sorting can be divided into several categories from different points of view:

Based on the usage of different storage in the process of sorting, sorting algorithms can be categorized as either internal or external sorting algorithms. For internal sorting, the size of the keys is small enough that all the keys can be entirely stored in the main memory. On the contrary, an external sorting algorithm needs the help of the external storage such as hard disk. Processors must exchange data between main memory and external storage. This kind of sorting is named external sorting. The speed of external sorting is much slower than internal sorting. Since there is a limitation of the storage ability of main memory, some files with large amounts of data cannot be loaded into the main memory at one time to sort. Therefore, external sorting is the only way in which we can sort files with large amount of data.

Sorting algorithms can also be categorized as stable sorting or unstable sorting algorithms. Stable sorting keeps the relative order for the keys with same value. A sorting algorithm is stable if the keys with same value after sorting are in the same
order as they are before sorting. If all the keys are distinct, the property of stability is trivial. Most of the sorting algorithms such as \textit{count sort} are stable. Unstable sorting does not need to keep the relative order for keys with same value. The \textit{quick sort} is an unstable sorting algorithm. The property of stability is very important because it is the primary reason which some sorting algorithms such as the \textit{radix sort} can work correctly.

Sorting algorithms can be divided into \textit{comparison-based sorting} and \textit{non comparison-based} sorting algorithms. A comparison-based sorting algorithm sorts an unordered sequence by repeatedly comparing two sequential keys, and exchanges them if they are out of order. Non comparison-based sorting algorithms exploit some properties of the keys, such as their binary representation, to sort.

Some of the most important applications for sorting are illustrated in [10]:

Solving the problem of togetherness, in which all keys with the same value are concatenated together. Keys with the same value will appear in consecutive positions after sorting. This is the conventional meaning of a sorting problem.

Matching keys among files: When the keys in several files are in order, the problem of matching keys can be satisfied in one sequential pass without backing up. Accessing the information table sorted on the key is usually faster than the randomly table, unless the table is so small that all the keys can be stored entirely in high-speed random access memory.

Searching information based on the value of the keys: Sorting can be used to accelerate the speed of searching. Sorting makes the data record easier to be read, because in most cases people feel more comfortable reading an information report sorted in alphabetical order than a report without any order.
The emergence of parallel computer architecture with large caches turns our attention more on the efficient parallel approaches to solve sorting problems. Parallelizing sequential sorting algorithms usually raises a number of issues. And those problems vary based on different sorting algorithms. For the parallel radix sorting algorithm which we will discuss in chapter 5, our primary concern is how to eliminate data dependences in the HISTOGRAM phase. For the parallel sample sort presented in chapter 6, our focus is on how to distribute keys into buckets as efficiently as possible.

1.3 Problem Statement

Nowadays, the multi-core and many-core architectures are becoming the trend of the processor design. From the above introduction we can conclude that sorting has been widely used. In order to achieve better performance, a large number of parallel sorting algorithms have been published. In this thesis, we will investigate and auto-tune two parallel sorting algorithms, i.e., radix sort and sample sort on the two parallel architectures, the many-core nVIDIA CUDA enabled graphics processors, and the multi-core Cell Broadband Engine. We re-design and manually tune these two parallel sorting algorithms to take advantage of multiple-level parallelism simultaneously, i.e., thread level parallelism, loop level parallelism, and data level parallelism (SIMD instructions). At the same time, we try to take advantage of the high-speed shared memory. Our experimental results showed that the parallel radix sort implemented on CUDA and Cell B.E. yield, on average, 45 and 7 times performance improvement, respectively, compared to the sequential radix sort. The experimental results also showed that the parallel sample sort implemented on CUDA and Cell B.E. have roughly 68 times and 4 times speedup on average, respectively, over the sequential quick sort.
1.4 Organization

The organization of the paper is as follows:

Chapter 1 gives the motivation and the background of the problem, also outlines the problems which this thesis will address.

Chapter 2 overviews some basic parallel concepts and principles related to this thesis, which include data dependency, instruction/data/thread level parallelism, granularity. This chapter will also introduce Amdahls law, which is the basis for later chapters. In the end of this chapter, we will introduce memory hierarchy.

Chapter 3 introduces the Cell Broadband Engine Architecture, as well as the architecture features which will be utilized in the Cell B.E. parallel sorting algorithms.

Chapter 4 illustrates the architecture of the nVIDIA graphics processors, along with the CUDA C language extension which will be exploited in the CUDA parallel sorting algorithms.

Chapter 5 presents the radix sort algorithm and its implementations on both target architectures.

Chapter 6 describes the sample sort algorithm in detail, as well as the implementations on both target architectures.

Chapter 7 concludes the master thesis.
CHAPTER 2

BASIC CONCEPTS ABOUT PARALLEL COMPUTING

2.1 Introduction

In this chapter, we elaborate on some basic definitions and concepts related to the techniques will be presented in this thesis, which includes data dependence, locality of reference, granularity, and instruction/data/thread level parallelism. Also, this chapter will introduce Amdahl’s law, which is the fundamental for later chapters. And this chapter will conclude with memory hierarchy.

2.2 Data Dependence

Dependence is a kind of relationship that represents how one instruction depending on the others. It determines to what extent parallelism can exist in a program as well as how to exploit this parallelism. Also, it restricts how the data is generated and consumed in correct program order. This relationship produced by this constraint is called data dependence. If two instructions are dependent, they should be executed in order. On the contrary, if two instructions are not dependent, they are parallel and can be run simultaneously.

There are three different types of dependences existing among instructions: data dependence, name dependence, and control dependence. In this chapter, we only discuss data dependence, which is highly related to the following chapters. Consider
the following C style segment of code:

S1 A = 5
S2 B = 2
S3 C = A + B

It is safe if we change the order of statement S1 and S2, but it will produce the potentially incorrect result when we reschedule S3 before either S1 or S2. In order to keep the program order that statement S1 and S2 execute before S3, we introduce data dependence from S1 and S2 to S3. There is no data dependence between S1 and S2, since the interchange of S1 and S2 will generate the same value as the execution order of S1, S2, S3. The following definition of data dependence is from [1].

**Definition 2.2.1.** There is a data dependence from statement S1 to statement S2 (statement S2 depends on statement S1) if and only if

- both statements access the same memory location and at least one of them stores into it,

- there is a feasible runtime execution path from S1 to S2 [1].

In order to produce correct result, we must take care of the order of load and store to the same memory location. Also, we must make sure that the order of two stores to the same location is correct so that the following reference will get the expected value.

There are three different kinds of data dependences which can be produced in a program based on [1]:
• True dependence or flow dependence. The former statement writes to, and the latter statement reads from the same memory location:

\[ S1 \quad M = \ldots \]
\[ S2 \quad \ldots = M \]

This type of dependence guarantees that the result read by \( S2 \) is written by \( S1 \) at first.

• Anti-dependence. The former statement reads from, and the latter statement writes to the same memory location:

\[ S1 \quad \ldots = M \]
\[ S2 \quad M = \ldots \]

The interchange of \( S1 \) and \( S2 \) will lead to \( S1 \) using an unexpected value produced by \( S2 \). Also, it will introduce true dependence if we change the order of these statements. This type of dependence guarantees that this case can not happen.

• Output dependence. Both statements store into the same memory location:

\[ S1 \quad M = \ldots \]
\[ S2 \quad M = \ldots \]

This type of dependence prevents \( S1 \) and \( S2 \) from interchange, which will cause the latter statement statement reading from an unexpected value. Consider the following example:

\[ S1 \quad M = 1 \]
\[ S2 \quad \ldots \]
\[ S3 \quad M = 2 \]
\[ S4 \quad N = 4 \times M \]

The result of variable \( N \) in \( S4 \) will be 8 if the above code fragment is executed in order. Suppose the order of \( S1 \) and \( S3 \) is exchanged, it will produce 4 instead
of 8 as a result for variable $N$, which is not what we expected.

From the perspective of hardware design, dependence is often called \textit{hazard} or \textit{stall}. True dependence is equal to read after write hazard; anti-dependence is equivalent with read after write hazard; and output dependence is the same as write after write hazard \cite{1,5}.

\section*{2.3 Locality of Reference}

In the area of computer science, locality of reference, also known as the principle of locality, is the phenomenon of the same value or related storage locations being frequently accessed \cite{20}. There are two basic types of locality which should be considered \cite{16}:

\begin{itemize}
\item Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon.
\item Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon.
\end{itemize}

Consider the following example:

\begin{verbatim}
FOR I = 0 TO N
ENDFOR
\end{verbatim}

The reference of $A[I]$ is temporal locality, since memory location of $A[I]$ is used twice in each iteration. Accessing array $B$ not only owns temporal locality but also has
spatial locality. Because this array is allocated to a contiguous memory space, after the access of an element of B in the first instruction, the item that next to this element will be accessed immediately in the second statement of each iteration.

The reason that locality occurs often is twofold. Firstly, highly related data is always stored in neighboring memory locations. For instance, we prefer storing a set of related data into an array or a vector. Secondly, programs involving loops usually access arrays or vectors by index. Due to these properties, it is feasible to use cache to improve system performance. Cache is a hardware component which is used to store data. It is proximate to CPU with small storage compared to main memory. Hence fetching data from cache is much faster than fetching from main memory. because not only is it close to CPU, but also it spends short time to addressing. Therefore, as the existence of locality, we prefer putting frequently accessed data into cache to improve system performance.

2.4 Granularity

In the field of parallel computing, granularity refers to the amount of computation in relation to communication [19]. It is a basic concept about data decomposition, and has a significant influence on the efficiency of parallel computing. Fine-grained parallelism, also called fine-grained data decomposition, which means there are larger number of individual tasks with smaller amounts of computation. On the other hand, for coarse-grained parallelism or coarse-grained decomposition, there are smaller number of individual tasks with larger amounts of computation. Although fine-grained decomposition can better utilize the potential parallelism, the overhead of synchronization and communication will become larger. To gain the best performance, care should be taken to find the balance of the overhead between computation and communication.
If the decomposition is too fine-grained, the timing performance can be dominated by communication; if it is too coarse-grained, computation can overwhelm the communication time.

In most cases, programmers manually tune the data size to obtain the best parallel performance based on their experience instead of deriving an optimum granularity mathematically. The shape of the data for each task also affects the parallel performance. Normally, the performance is better if the data stored in memory could be coalesced. The potential reason is that the function unit may combine multiple memory access instructions together and then issue them, which results in accessing memory only once.

### 2.4.1 Instruction-Level Parallelism

Since about 1985, all processors adopt pipelining to overlap the execution of instructions and improve performance. This potential parallelism among instructions is called instruction-level parallelism (ILP), because instructions may be executed in parallel [5].

Instruction-level parallelism can be increased by two main methods. The first is overlapping more instructions to be executed simultaneously by increasing the pipeline depth. The other method is doubling the functional units inside CPU so that multiple instructions can be issued in each clock cycle. We call this technique super-scalar. However, if data dependence exists among instructions, the performance speedup which can be achieved by using these methods is limited. In following code segment:

\[
\begin{align*}
S1 & \quad M = A \times B \\
S2 & \quad N = C + D \\
S3 & \quad P = M \times N
\end{align*}
\]

Statements S1 and S2 are independent, so they can be executed in parallel. Suppose
each instruction can be finished in one clock cycle, and if we only take statements S1 and S2 into account, the ILP is 2. However, since statement S3 depends on the results of S1 and S2, it cannot be executed until its required values are available. For all of these three instructions, it takes two clock cycles to complete them. Therefore, the ILP is $\frac{3}{2}$.

Typically, programmers write code in a sequential manner by which they suppose instructions are executed in the order specified by programs. In this model, they do not need to consider how to schedule their code to best utilize ILP. In contrast, it is the task of compiler or underlying hardware to identify and exploit as much ILP as possible to improve performance.

### 2.4.2 Data-Level Parallelism

Data-level parallelism (DLP), or loop-level parallelism, describes applying the same instruction to different data. A typical example of DLP is matrix addition. Since all the elements of two matrices have no data dependences for addition, the operation of addition can be performed simultaneously.

SIMD, single instruction multiple data, is a basic concept highly related with DLP. From the hardware perspective, it allows all function units to execute in parallel, and allows all of them to perform a single instruction but with different data sets. From the programmer’s point of view, a set of data can be operated at the same time, such as addition of vectors.

The benefit behind DLP is that it amortizes the cost of instruction fetch and decode, because one issued instruction can be used in multiple function units. DLP is very efficient when it works on arrays in loops.
FOR \( I = 0 \) TO \( 7 \)
\[
C[I] = A[I] \times B[I]
\]
ENDFOR

In the above example, if there is only one function unit residing in the processor, it will take 8 iterations to finish this loop. However, suppose we have two function units. Then array A, B and C can be fairly distributed to each unit. Therefore, the first unit needs only to execute an even number of iterations, while the second one is taking care of odd number of iterations. All of the function units will perform multiplication but on different data which is decided by the loop index.

2.4.3 Thread-Level Parallelism

It is totally transparent to programmers that using instruction-level parallelism improves system performance, but ILP’s application is very limited, especially for instructions with data dependences. Thread-level parallelism, as known as task-level parallelism, is higher-level parallelism. It logically groups instructions as separate threads which can be executed in parallel. A thread is the smallest execution unit which can be scheduled independently by the operating system. Compared with ILP, which focuses on finding and exploiting the parallelism within loops or straight-line instructions, thread-level parallelism explicitly distributes tasks to multiple threads by using the inherently parallelism of the program, and executes them in parallel.

Thread-level parallelism is an important alternative to data-level or instruction-level parallelism [5]. As we will discuss in later chapters, the high performance can be achieved in the CELL B.E. platform by exploiting the data-level parallelism such as SIMD instructions. But as to according to taking advantage of the thread-level parallelism, even higher performance is gained in the platform of CUDA.
Following code segment of CUDA is a straightforward example showing how to use thread-level parallelism:

```c
int main()
{
    ... 
    int A[4], B[4], C[4];
    ... 
    array_add<<<1, 4>>>(A[], B[], C[]);
    ... 
}

... 

__global__ void array_add(g_A[], g_B[], g_C[])
{
    idx = threadIdx.x;
    g_C[thid] = g_A[idx] + g_B[idx];
}
```

In this example, four threads are explicitly declared and dispatched to execute the function of array addition. Each thread performs addition only once, and the data it needs can be located by using its unique thread number, so these four threads work together to complete the operation of array addition.
2.5 Amdahl’s Law

In parallel computing, the improvement of system performance which can be obtained is limited by the program segment that can be parallelized. In other words, not an entire program can be parallelized. It is usually impossible to optimize some part of a program. Even if we can, the return in doing that is very trivial or may cause side-effect. Amdahl’s law serves as a guideline to find the maximum performance improvement. It states that the performance improvement which can be obtained from using parallelism is limited by the fraction of a program in which the parallelism can be used [5]. We can use a formula to express it:

\[ T_{new} = T_{old} \times Fraction_{unaffected} + \frac{T_{old} \times Fraction_{affected}}{Amount \ of \ improvement} \]

For example, suppose the execution time of a sequential program is 100 seconds. And we use 4 cores to run 80% of this program in parallel after optimizing. Therefore, for this problem:

\[ T_{new} = 100 \times 20\% + \frac{100 \times 80\%}{4} = 20 + 20 = 40 \text{ seconds} \]

Amdahl’s law also defines the speedup to tell us how much faster a parallelized program will run compared with the original sequential program. Speedup is the ratio:

\[ speedup = \frac{Execution \ time \ for \ an \ original \ program}{Execution \ time \ for \ this \ parallelized \ program} \]

For the above example, the speedup = \( \frac{100}{40} \) = 2.5. Note that if we take the parallel overhead, such as communication between cores into account, the time spent on a parallelization program may be more than the sequential one. That is, the overhead of
exploiting parallelism overwhelms the parallel computation and causes a degradation of performance.

2.6 Memory Organization

For modern multi-processors, there are two different approaches used for accessing data among them. In the first method, data is accessed based on the address of shared space. We call this type of memory architecture shared memory. Another way of multi-processors access data is by utilizing the distributed memory, in which each processor has in its own private memory.

2.6.1 Shared Memory

In a shared memory system, all processors share a single memory address and communicate with each other by writing to and reading from shared variables [13]. The most popular class of shared memory systems is called uniform memory access (UMA), which arises from the fact that the latency of accessing a memory location is independent from the processor making this request. Figure 2.1 shows its basic structure. Although this type of memory system is easy for programming, UMA is not well-scaled, since the more memory access requests the processors present, the less memory bandwidth per processor has.

The other category of shared memory systems is nonuniform memory access (NUMA), as shown in Figure 2.2. Memory in this system is still accessible for all processors, but the latency depends on the physical distance between the requesting processor and the target memory block. This eliminates the scalable problem. But care must be taken to distribute data into different memory blocks when programming, because the system performance is influenced by the layout of the data. System perfor-
mance will be improved if we arrange the data most often used close to its associated processor. Obviously, this requires extra work for programming.

2.6.2 Distributed Memory

Alternatively, in distributed memory systems, each process has its private address space which is not addressable by other processors. Instead, they must use message passing to communicate with each other. Figure 2.3 shows its layout. This type of memory system can be divided into two further categories: massively parallel proces-
sors (MPP) and clusters [13]. For an MMP system, processors are highly coupled on a board and specialized for parallel computing. One example of this system is CELL B.E. architecture. Clusters are composed of independent computers connected by a network. Depending on how closely the multi-processors are integrated together, the communicating time is varied greatly. For CELL B.E. architecture, the communication time between PPE or SPE and SPE is not longer than shared memory system. In contrast, the communication time of clusters is always several orders of magnitude longer than CELL B.E..

Figure 2.3. Basic Architecture of Distributed Memory System
CHAPTER 3
THE ARCHITECTURE OF CELL BROADBAND ENGINE

3.1 Introduction

In Randall Hyde’s book, *Write Great Code*, one of his basic ideas is that, for program optimizing, you need to not only know the APIs supplied by the target platform but also understand how the code is executed on it. This principal still works for programmers who pursue high performance coding by using Cell Broadband Engine Architecture (CBEA). It is not good enough to know the extension of C/C++ commands in CELL. In order to obtain high performance, one needs to understand the design principles of this architecture.

Although the initial purpose of Cell Broadband Engine is used for media-rich applications such as game consoles, this architecture exposes the outstanding performance for scientific applications. The Cell Broadband Engine is a single-chip heterogeneous multi-processor system cooperating with both shared and distributed memory. It has nine processors:

- one PowerPC Processor Element (PPE), which is optimized for scheduling tasks and running operating system.

- eight Synergistic Processor Elements (SPEs), which is optimized for running compute-intensive tasks.
The PPE has the same memory hierarchy as the ordinary CPU, and it loads data or instructions into cache from main memory which is shared by all processors before consuming them. In contrast, the SPE accesses the main memory by using direct memory access (DMA) commands, and stores data or instructions into the private local store to be used in the future. Replacing cache with local store is a radical innovation in computer architecture.

The reason for this radical change is based on the fact that the bottleneck of system performance is, in most cases, determined by the memory latency rather than computation ability or peak bandwidth of memory. When cache miss occurs for a sequential program running on the conventional architecture, there will be several hundred clock cycles penalty. But for SPE, if a miss happens, the setting up time of DMA is only few cycles; which is very small compared with several hundred clock cycles halting.

3.2 Decreasing the Impact of Serial-performance Walls

As we have discussed in chapter 1, the performance of modern computer architecture is limited by the problems of power dissipation, memory latency and frequency. However, due to the innovation of the Cell Broadband Engine Architecture, these impacts are scaled down.

Power dissipation of a circuit is proportional to the clock frequency, the technology used by manufacturers to dissipate heat has almost reached a physical limit. Hardware engineers can not integrate more transistors because of lacking necessary power dissipation technique support. Therefore, the performance limitation has been shifted from clock speed to the available technology for power dissipation. CBEA improves power efficiency to alleviate this problem. Rather than using common general-
purpose processors, two types of special designed cores are adopted. The PPE is specialized for control-intensive code, and eight SPEs are optimized for computing-rich programs.

Currently, the ratio of memory latency to clock speed is approaching 1000. As a result, the serial performance is determined by the way of how to efficiently accessing memory. CBEA uses two mechanisms to mitigate the problem of memory latency:

- a 3-level memory hierarchy of SPE; that is main memory, with local stores and private register file;
- asynchronous DMA transfers between memory and local store [8].

The long latency can be efficiently covered by scheduling data transfer and computation simultaneously.

Traditionally, processors need to make pipelines deeper to achieve higher throughput. But this approach has reached its limitation and even garners negative return if we take power consumption into account. The reason for the high performance of the PPE is that it is able to run two threads simultaneously. Each SPE achieves efficiency primarily by using a large register file and asynchronous DMA transfer.

### 3.3 Architecture Overview

Figure 3.1 shows the primary block diagram of the CBEA: the PowerPC Processor Element (PPE), eight Synergistic Processor Elements (SPEs), the Element Interconnect Bus (EIB), the Cell Broadband Engine Interface (BEI), the Memory Interface Controller (MIC), etc. We will explore some components in greater depth in the following sections.
3.4 PowerPC Processor Element

The PowerPC Processor Element (PPE) is the center of CBEA. It runs the operating system, manages and dispatches workload among SPEs, and coordinates operations. The PPE is a general-purpose, dual-threaded, 64-bit RISC processor with the vector/SIMD multimedia extension processor [7]. As shown in Figure 3.2, the PPE consists of two main elements: the PowerPC Processor Unit, or PPU, and the PowerPC Processor Storage Subsystem (PPSS) [7].

3.4.1 PowerPC Processor Unit

The PPU has a level-one instruction cache, a level-one data cache, a register file and several functional units. It executes the 64-bit RISC PowerPC Architecture instruction set and the vector/SIMD multimedia extension instructions [7], and performs 32-byte load and 16-byte store operations. One notable feature of the PPU is that it allows two threads to run simultaneously, which is called symmetric multi-threading (SMT). The
PPU has the following functional units [7]:

- **Instruction Unit (IU)** – The IU contains a 2-way set associative level-1, 32 KB instruction cache with cache-line size of 128 bytes. It executes instruction fetch, decode, issue, and branch.

- **Fixed-Point Unit (FXU)** – The FXU performs fixed-point operations, such as arithmetic and logic operations.

- **Load and Store Unit (LSU)** – The LSU is responsible for data access. It has a level-1, 32 KB, 4-way set associative and write-through data cache. The cache-line size is 128 bytes.

- **Vector/Scalar Unit (VSU)** – The VSU contains a floating-point unit (FPU) which executes floating-point operations, and a 128-bit vector/SIMD multimedia extension unit (VXU) which performs vector/SIMD multimedia extension instructions.
• **Memory Management Unit (MMU)** – The MMU takes care of address translation, which includes one 64-entry segment look-aside buffer (SLB) and one 1024-entry translation look-aside buffer (TLB).

### 3.4.2 PowerPC Processor Storage Subsystem

The PPSS handles all memory requests from the PPU and the Element Interconnect Bus (EIB). It has a unified, 512 KB, 8-way set-associative, write-back level-2 instruction and data cache. The cache-line size is 128 bytes, which is the same as L1 cache [8]. The L2 cache does not need to contain all the contents of L1 instruction cache, but includes all the contents of the L1 data cache. The interface between the PPU and the PPSS supports 32-byte load and 16-byte store operation. And both the load port and store port between the PPSS and the EIB are 16 bytes wide.

### 3.5 Synergistic Processor Elements

The PPE is a general-purpose processor, but the eight Synergistic Processor Elements (SPEs) are specifically designed for only one purpose: performing high-speed SIMD operations. Each SPE is a 128-bit RISC processor including two parallel pipelines which execute SIMD instruction set. Every SPE provides a separate computing environment for tasks. Since the SPE does not have cache, cache miss is not affecting its performance. Each SPE contains two main units: the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC) [7], shown in Figure 3.3.

#### 3.5.1 Synergistic Processor Unit

The SPU receives tasks from the PPU and executes them. The required instructions and data for each SPU are stored in its local store. Each SPU is an independent pro-
Figure 3.3. Structure of SPE

The SPE processor with four execution units, and specified to run SPU programs. The SPU loads instructions and data to its local store using DMA transfer, then consumes them and stores the result back to local store. Local store, along with SPU’s large register file, is the memory hierarchy that the SPU can access directly.

Local store is a unified non-caching memory with a size of 256 KB. Each SPU has its own local store memory. And it is not addressable from another remote SPUs. The bandwidth of the SPU instruction and data transfer are 128 bytes and 16 bytes per clock cycle, respectively. But the data access needs to be quad-word aligned. For DMA transfer, the bandwidth is 128 bytes per clock cycle. There is no address translation when an SPU accesses its local store.

### 3.5.2 Memory Flow Controller

Each SPE has one MFC which plays a role as the SPU’s interface. The primary function of the MFC is executing DMA commands autonomously. When the PPE needs
to send data to the SPEs, it specifies the source address of the main memory, the destination address of their local stores, and notifies the DMA controller of each MFC to start working. Also, when the SPEs need to communicate with the main memory, they set up DMA transfers and tell MFC to move data from local stores. This makes SPUs focusing only on computing, and frees it from dealing with data transfer. While DMA transfer allows up to 16K bytes of data transfer for each DMA transfer instance, all the transferred data need to be 128-bit aligned. Moreover, the size of the transferred data needs to be 1, 2, 4, 8, 16 or multiples of 16 bytes.

### 3.6 The Element Interconnect Bus

The Element Interconnect Bus (EIB) is the separated communication paths for commands and data among the PPE, the SPEs, the controllers for the main memory, and the I/O interface. Functionally, it consists of four data rings. Two of them transfer data in the clockwise direction, and the other two run in the counterclockwise direction. The peak bandwidth of the EIB is 204.8 GB/s [4].

### 3.7 The Memory Interface Controller

The Memory Interface Controller (MIC) connects the EIB and physical memory, and provides two channels to physical memory. This memory, called Extreme Data Rate (XDR) DRAM, differs from the conventional DRAM because it supports eight data transfer per clock cycle rather than the usual two or four. Therefore, the memory can achieve a high data rate without very high clock frequency [17].
3.8 Cell Broadband Engine Interface Unit

The on-chip Cell Broadband Engine Interface Unit (BEI) serves as the I/O interface. It contains a Bus Interface Controller (BIC), an I/O controller (IOC), and an Internal Interrupt Controller (IIC). It manages data transfer between the EIB and the I/O devices [7].

3.9 Programming Model for the PPE and SPEs

Both the PPE and the SPE instruction sets support C-language commands. Additionally, these sets also include vector data types and a large set of scalar and vector intrinsics. Intrinsics are, in essential, inline assembly-language instructions which have similar syntax to C-language [8]. A vector is an operand containing multiple unified data elements. The type of the element consists of the C-language data type. Most PPE and SPE instructions operate on vector operands, which are also called SIMD operands or packed operands [7].

![Figure 3.4. Vector Add Operation](image)
SIMD which exploits data-level parallelism is widely applied in the Cell Broadband Engine. In the PPE, it is supported by the Vector/SIMD Multimedia Extension instruction set. In the SPE, it is supported by a new instruction set which is called SPU instruction set. For both the PPE and the SPE, the vector registers are 128-bit wide, which can hold multiple data elements. For instance, four 32-bit integers can be loaded into a single vector register simultaneously. The vector add operation is shown in Figure 3.4.

Programmers can manually vectorize scalar data array to vector. This process is called vectorization or SIMDization. The following code is an example showing the difference between scalar addition and vector addition:

```c
int a[4] = {1, 2, 3, 4};
int b[4] = {5, 6, 7, 8};
int c[4];
for (i = 0; i < 4; i++)
    c[i] = a[i] + b[i];
```

```c
int a[4] = {1, 2, 3, 4};
int b[4] = {5, 6, 7, 8};
int c[4];
vector signed int *va = (vector signed int *) a;
vector signed int *vb = (vector signed int *) b;
vector signed int *vc = (vector signed int *) c;
*vc = vec_add(*va, *vb); // PPE: 1 + 5, 2 + 6, 3 + 7, 4 + 8
```

Data multi-buffering is another mechanism to improve system performance. It can efficiently hide the overhead of DMA transfer: The SPUs are busy consuming the current data set, while the MFC is loading data which will be used in the next round to another data buffer. So each SPE loads new data into its local store when it operate on the data already in its local store. Thus, the overhead of the data transfer can be hidden.
3.9.1 Language-extension Differences Between PPE and SPE

Both instruction sets of the PPE and the SPE are operated on *128-bit SIMD vectors*. But from the programmer’s perspective, they are quite different because of the different intrinsics and different supported data types. Table 3.1 specifies the vector data supported by PPE and SPE [8].

<table>
<thead>
<tr>
<th>Vector Data Type</th>
<th>Meaning</th>
<th>PPE</th>
<th>SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector unsigned char</td>
<td>16 8-bit unsigned values</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>vector signed char</td>
<td>16 8-bit signed values</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>vector bool char</td>
<td>16 8-bit unsigned boolean</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>vector unsigned short</td>
<td>8 16-bit unsigned values</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>vector signed short</td>
<td>8 16-bit signed values</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>vector bool short</td>
<td>8 16-bit unsigned boolean</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>vector unsigned int</td>
<td>4 32-bit unsigned values</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>vector signed int</td>
<td>4 32-bit unsigned values</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>vector bool int</td>
<td>4 32-bit unsigned values</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>vector float</td>
<td>4 32-bit signed precision</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>vector unsigned long long</td>
<td>2 64-bit unsigned values</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>vector signed long long</td>
<td>2 64-bit signed values</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>vector double</td>
<td>2 64-bit signed values</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 3.1. Vector Data Type Supported by PPE and SPE

The differences are:

- The PPE does not support long long type;
- The SPE does not support boolean value.
In a specific instruction set, instruction names are usually started by a unified
prefix. That is still true for both the PPE and the SPE instruction sets. For the SIMD
instructions of the PPE, they are named by the prefix of "vec_", such as vec_add and
vec_or. For the SPE, instructions are denominated by the prefix of "spu_", for instance,
spu_add and spu_or which are the same functions as vec_add and vec_or in the PPE.

3.9.2 Communication between the PPE and SPEs

While DMA transfers allow the data passing, using mailbox is another way to comму-
icate between the PPE and the SPE. The mailbox is designed for transferring small
message, such as a memory address and status information between the PPE and the
SPE, because it can only transfer 32-bit data each time. In essence, a mailbox is a
queue which can hold only small number of entries. Here, we only introduce two
types of mailboxes which are related to our later chapters.

- **SPU Inbound Mailbox**: It can only store up to 4 32-bit messages, and is used to
  transfer data from the PPE to the SPE. If no message is available when it is read
  by an SPE, this SPE will stall until there is data available.

- **SPU Outbound Mailbox**: It only has space for holding 1 32-bit message, and is
  used to send data from the SPE to the PPE. If it is full, writing of the incoming
  data will be suspended until the PPE reads data from it.
CHAPTER 4
THE ARCHITECTURE OF GPU

4.1 Introduction

The first Graphics Processing Unit (GPU), invented by nVIDIA in 1999, was aiming at processing and generating graphics. Driven by the market demand for real-time graphics, the GPU has evolved into a many-core processor with multi-threading and is highly parallel. It has over-performed the general purpose CPUs in arithmetic throughput and memory bandwidth from 2003. Since then, the research on non-graphics applications by exploiting the GPUs began to emerge. We call these general-purpose programming on GPU GPGPU.

Although the GPU has better performance, it suffers some flaws. Firstly, it required programmers to have knowledge about the GPU architecture and its programming languages, for instance, DirectX and openGL. Moreover, some programming features such as double precision and random memory read and write were not supported. These factors impeded the making of the GPU into a universal platform for scientific applications.

In order to address these problems, nVIDIA introduced two significant architectures in November 2006 [3]: the G80 unified graphics and compute architecture and Compute Unified Device Architecture (CUDA). CUDA is a software and hardware architecture that allows programmers to code in high level languages without considering the graphics interfaces of the GPU. Instead of using APIs provided by GPU, a
programmer can write a general-purpose application with CUDA C/C++. Compared with the general-purpose CPU, the programming model for CUDA is single program multiple data (SPMD), in which each instruction of a program will be executed by multiple threads in parallel.

4.2 An Overview of the GPU Architecture

Figure 4.1 shows the typical GPU architecture. It is based on a programmable processor array called Streaming Multiprocessor (SM) which is highly parallel. In G80 architecture, two SMs form a Texture/Processor Cluster (TPC), and each SM has 8 highly multi-threaded streaming Processors (SPs).
The G80 GPU has up to 4 gigabytes of graphics double data rate (GDDR) DRAM which is known as global memory [9]. The global memory is a very high bandwidth, off-chip memory. Its available bandwidth is 84.6 GB/s, which is overwhelming the 8 GB/s bandwidth of system memory. Since the latency is a little more than typical system memory, the high bandwidth is a compensation for the longer latency in parallel applications.

Each SM supports 768 threads, and there are 16 SMs resident in the G80 GPU, so more than 12,000 threads coexist in the chip. The level of parallelism provided by the GPU is very high, which is deserved to putting effort on exploiting this feature for massively parallel applications.

### 4.2.1 Streaming Processor Array

As shown in Figure 4.1, the G80 GPU with 16 SMs is organized into 8 TPCs, and this 8-TPC vector forms the Streaming Processor Array (SPA) which manages the thread scheduling and executes GPU computing programs. The number of TPCs determines the programming ability of a GPU. The more TPCs available, the higher performance a GPU can deliver.

### 4.2.2 Streaming Multi-processor

In Figure 4.2, each Texture/Processor Cluster (TPC) consists of a geometry controller, an SM controller (SMC), two SMs, and a texture unit [11]. The Streaming Multiprocessor (SM) is a homogeneous multi-processor which can execute both graphics and parallel computing applications. Each SM contains 8 Streaming Processors (SPs), 2 Special Function Units (SFUs), and a read/write shared memory.

To efficiently create and execute hundreds even thousands of threads concur-
Currently when running a program, the SM is hardware multi-threaded. In other words, the threads are managed and scheduled by hardware without any overhead. In order to achieve parallel computing, each thread has its own execution state, which allows threads to execute independently. The concurrent threads of a program can synchronize at a barrier with a single CUDA C/C++ instruction "__syncthreads" [11]. Moreover, compared to the Cell B.E. architecture which runs SIMD instruction, the threads in the SM execute scalar CUDA C/C++ instructions because scalar instructions are simpler and compiler-friendly [16]. The low-cost thread creation and management with an easy way of synchronization, along with a small instruction set, make the GPU to be an efficient platform to realize parallelism.

Figure 4.2. Texture/Processor Cluster (TPC)
4.2.3 Streaming Processor

The Streaming Processor (SP) is the key functional element in the SM. Each SP has a scalar register file with 1024 32-bit entries, and supports up to 64 threads. The SP, optimized for fully pipelined, performs all 32-bit and 64-bit integer arithmetic, comparison, and logical instructions [11]. It also implements the fundamental float-point operations such as add and multiply-add.

4.3 Memory Hierarchy

The CUDA-capable GPU presents three levels of memories to allow thread access: the global memory, the shared memory and the local memory. It provides different memory space to programmers for storing value in the best-performance way.

4.3.1 Global Memory

The global memory is implemented in external DRAM [11], and shared by all threads of a program. It has two key functions: providing a relative large storage space for a GPU program and a communication way among SMs. In fact, reference a location in global memory by multiple threads is executed sequentially rather than in parallel, and there is no any guarantee that these threads will have sequential consistency.

Within a thread, the order of memory access to the same address is maintained, but the order of memory read and write to different addresses is not preserved [16]. The memory access requests are processed without order. Within a thread block, the CUDA barrier/fence instruction "\texttt{syncthreads}\" can be used to synchronize these threads. This instruction provides a way to synchronize threads which commit prior memory operations and make them available to later memory access.
4.3.2 Shared Memory

The shared memory cannot be shared by all threads of a program. It is only accessible to the threads in a thread block. As shown in Figure 4.1, the shared memory resides on-chip, which makes memory accessing much faster. Compared to the global memory, the shared memory does not suffer the problem of competing with the limited off-chip bandwidth within a thread block. So it is practical to provide very high-bandwidth memory space for each thread block.

4.3.3 Local Memory

The local memory is only visible to an individual thread. It is physically larger than per-thread’s register file, and is addressable by a program. In order to achieve larger memory space for each single thread, the operation of memory allocation is implemented in external DRAM.

4.4 Programming Model for CUDA GPU

CUDA, invented by nVIDIA in late 2006, is an extension to the conventional C/C++ language. It is a scalable parallel programming language which exploits a large degree of thread-level parallelism for general applications. It provides three key utilities for parallelism — a hierarchy of the thread group, shared memory and barrier/fence synchronization [16].

4.4.1 CUDA Programming Paradigm

In order to solve a problem efficiently in the platform of CUDA GPU, the programmer usually partitions the problem into a number of relatively small sub-problems which
can be solved in *kernels* in parallel. A kernel is a function that can be executed by many threads simultaneously. The programmer organizes several threads into a *thread block* or a *Cooperative Thread Array* (CTA) for better management and scheduling. A thread block is usually used to solve a sub-problem. A group of thread blocks form a grid to solve a certain problem by executing the same kernel function in parallel.

The declaration of a kernel is prefixed by the **`__global__`** qualifier. Its return type must be **`void`**. The syntax of kernel procedure called from a CUDA program is:

```
kernel_name <<< dimGrid, dimBlock >>> (...);
```

where *dimGrid* and *dimBlock* are three-element vectors which specify the dimensions of the grid in blocks and the dimensions of the blocks in threads respectively [16]. A kernel can call device functions as well. The declaration of a function started with the **`__device__`** identifier indicates that it is a device function and must be called from the kernel only.

When a kernel is invoked, there are $\text{dimGrid} \times \text{dimBlock}$ threads will be generated. Each thread in a block is assigned to a thread number *threadIdx*. Each block has a unique block number *blockIdx* which is visible to all threads in the block.

As Figure 4.3 shows, a thread may use different level memories in its execution. A CUDA program uses the local memory for per-thread storage, which may be larger than its register file, for stacks and register spilling. The local variable, which is stored in local memory, is declared by the **`__device__`** identifier proceeding the variable type. Each thread block has a shared memory, which can be accessed by all the threads in this block. The threads in a thread block always work together loading the most often used data into their shared memory for future utilization. Declaring variables in shared memory is prefixed by **`__shared__`** qualifier.
Figure 4.3. CUDA Memory Hierarchy

The global memory space for a kernel is managed at run time through the CUDA APIs: `cudaMalloc()` and `cudaFree()`. It is visible to the whole grid running this kernel. Moreover, `cudaMemcpy()` can be used to transfer data between GPU and CPU or between GPU’s.

4.4.2 Single Instruction Multiple Threads and Warp Divergence

The CUDA architecture exploits thread-level parallelism to improve system performance. This coarse-grained parallelism is called Single Instruction Multiple Threads (SIMT) which executes a single instruction by multiple independent threads in parallel. Threads are created, scheduled and executed as groups called *warps*. Each SM has a pool consisting of 16 warps [16], and each warp contains 32 threads.

Figure 4.4 shows how the thread scheduler dispatch threads to execute instructions. Since warps are independent, at each instruction issue time slot, the instruction
Figure 4.4. Warp Scheduling

The high performance of a CUDA program can be achieved when all threads in a warp have the same execution path. And the performance will be degraded if a warp diverges, which means the execution path is not unified within the thread warp. For instance, a branch condition is taken in half of a warp, and is not taken in the other half. In this case, the instructions of branch taken and not taken must be executed sequentially rather than in parallel for this warp. Different execution paths among warps have no effect on system performance.

From the perspective of code correctness, programmers may ignore the concept...
of warp and its divergence. But potential performance will be improved by putting effort into minimizing the warp divergence.
CHAPTER 5
PARALLEL RADIX SORT

In this chapter, we elaborate on the parallel radix sort implemented on both Cell Broadband Architecture and CUDA GPU Architecture. Parallel prefix sum is a key primitive for the CUDA radix and sample sorts, so it will be introduced in the first subsection of this chapter. To achieve high performance, these algorithms are carefully designed to exploit the features of the target platforms. The results show that the CUDA radix sort yields about 45 times performance improvement compared to the sequential radix sort, and the radix sort on Cell B.E. is about 7 times faster.

5.1 The Prefix Sum Primitives

In computer science, the exclusive prefix sum, or exclusive scan is an operation that takes a binary operator \( \oplus \) with identity \( I \), a sequence of elements:

\[
[x_0, x_1, \ldots, x_{n-1}],
\]

and returns a new sequence [2]:

\[
[I, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})].
\]

For the exclusive prefix sum, each element \( i \) in the input is replaced by the sum of all elements up to but not including the element \( i \). In contrast, the inclusive prefix sum, or inclusive scan returns:

\[
[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})],
\]

45
where each input element is substituted by the sum up to and including \( i \). For example, for the binary operation addition, given an array:

\[
[3, 1, 2, 2, 7, 4, 6, 5],
\]

the exclusive scan will return the result:

\[
[0, 3, 4, 6, 8, 15, 19, 25],
\]

but the result of the inclusive scan is:

\[
[3, 4, 6, 8, 15, 19, 25, 30].
\]

The exclusive and inclusive scan can be transformed mutually. An exclusive scan can be produced by shifting the inclusive scanned array one element right and inserting the identity in the leftmost position. Similarly, an inclusive scan can be generated by shifting the exclusive scanned array one element left and inserting the sum of the whole input array into the rightmost position \([2]\). The typical binary functions are addition, max, min and logical operations. But for this and the following chapters, we focus only on the discussion of the exclusive scan with binary operation addition, and simply refer to it as \textit{scan}.

### 5.1.1 Sequential Scan

\begin{algorithm}
\textbf{Algorithm 5.1.1} The Sequential Version of Scan Algorithm
\begin{tabbing}
\textbf{Input:} \hspace{1cm} an array \( A \) \tab \textbf{Output:} \hspace{0.5cm} a scanned array \( B \) \\
\hspace{1cm} \begin{align*}
B[0] & \leftarrow 0 \\
\text{for} \ i & \leftarrow 1 \ \text{to} \ length[A] - 1 \ \text{do} \\
& \quad B[i] \leftarrow A[i - 1] + B[i - 1] \\
\text{end for}
\end{align*}
\end{tabbing}
\end{algorithm}
From Algorithm 5.1.1, in order to get the result of $B[i]$ in output ($1 \leq i \leq length[A]$), the previous element of input $A[i - 1]$ needs to be added to the previous element of output $B[i - 1]$. The minimum number of additions needed to produce the scanned array is $O(n)$ for an array with length of $n$. In other words, $n - 1$ iterations are required for traversing all the elements of the input to generate the output data, so the time complexity is $O(n)$.

### 5.1.2 An Inefficient Parallel Scan

Algorithm 5.1.1 does not take advantage of any parallelism. We prefer to develop the parallel version of the scan algorithm by exploiting the platform’s parallelism. Hillis and Steele proposed the first parallel scan in 1986 which is shown in Algorithm 5.1.2 [6]. Its operation is illustrated in Figure 5.1. Algorithm 5.1.2 executes $\log_2^n$ iterations, where $n$ is the size of array $A$, so totally $O(n\log_2^n)$ additions are performed. The time complexity for Algorithm 5.1.2 is $O(\log_2^n)$, if there are as many ALUs as the array elements. But it is usually the case that the number of elements needed to be processed is greater than the number of ALUs. Hence, there is no guarantee that the performance of Algorithm 5.1.2 is better than Algorithm 5.1.1. In the worst case, if

---

**Algorithm 5.1.2 An Inefficient Parallel Scan Algorithm**

**Input:** an array $A$

**Output:** the scanned array $A$

```
for $i \leftarrow 0$ to $\log_2^{length[A]} - 1$ do
    for all $t$ in parallel do
        if $t \geq 2^i$ then
        end if
    end for
end for
```
Algorithm 5.1.2 is executed on the platform with only one processor, the time complexity will be degraded to $O(n \log^2 n)$, which is apparently worse than the sequential one.

![Figure 5.1. An Illustration of the Inefficient Scan](image)

### 5.1.3 An Efficient Parallel Scan

The performance of Algorithm 5.1.2 is not guaranteed to be good, especially when working with large arrays. Sengupta and Harris et al. [18] presented a work-efficient parallel scan based on the pattern of the balanced tree in 2007. They partitioned the scan algorithm into two phases: up-sweep, or reduce phase and down-sweep phase. The up-sweep phase is illustrated in Figure 5.2. Pseudocode for the up-sweep phase is

**Algorithm 5.1.3 The Up-Sweep Phase of the Efficient Scan Algorithm**

**Input:** an array $A$

**Output:** the scanned array $A$

for $i$ ← 0 to $\log_2^\text{length}[A] - 1$

for all $t = 0$ to $n - 1$ by $2^{i+1}$ in parallel do

$A[t + 2^{i+1} - 1] \leftarrow A[t + 2^i - 1] + A[t + 2^{i+1} - 1]$

end for

end for
given in Algorithm 5.1.3 [18]. The up-sweep phase performs \( n - 1 \) additions for an \( n \) elements array, where \( n = 2^d \). And \( \log_2 n \) iterations are needed to finish the up-sweep phase.

![Figure 5.2. An Illustration of the Up-Sweep Phase of the Efficient Scan](image)

Algorithm 5.1.4 is the pseudocode for down-sweep phase [18]. And its operation is shown in Figure 5.3. For the down-sweep phase, Algorithm 5.1.4 starts from assigning 0 to the last element of array A. After \( n - 1 \) additions and \( n - 1 \) swaps,

```
Algorithm 5.1.4 The Down-Sweep Phase of the Efficient Scan Algorithm
Input: an array A
Output: the scanned array A
A[length[A] − 1] ← 0
for i ← log_2 length[A] − 1 down to 0 do
    for all t = 0 to length[A] − 1 by 2^{i+1} in parallel do
        temp ← A[t + 2^i − 1]
        A[t + 2^i − 1] ← A[t + 2^{i+1} − 1]
        A[t + 2^{i+1} − 1] ← temp + A[t + 2^{i+1} − 1]
    end for
end for
```

we get the scanned array. Also, Algorithm 5.1.4 still needs \( \log_2 n \) iterations, so the time complexity for this efficient scan algorithm is \( O(\log_2 n) \). There are totally \( 3 \times (n - 1) + 1 \)
arithmetic operations performed \(2 \times (n - 1)\) additions, \(n - 1\) swaps and 1 assignment). Therefore, the efficient parallel scan algorithm executes \(O(n)\) operations, which has better performance than Algorithm 5.1.2, especially when working with large arrays.

![Figure 5.3. An Illustration of the Down-Sweep Phase of the Efficient Scan](image)

### 5.2 Radix Sort

The *radix sorting* algorithm depends on the representation of the keys to be sorted. Since numbers are stored as a sequence of binary bits in the memory, it is usually the case that radix sort is performed based on the binary representation of the keys rather than their decimal representation. The radix sorting algorithm considers a bits sequence as a multiple-digit number, and sorts it from its least significant digit to its most significant digit. The value of each digit is in the range \([0, 1, \ldots, m]\), where \(m\) is called the radix.

For a sequence of 32-bit integers, for instance, they could be treated as a group of 8-digit numbers with the radix \(r\) being 16 \((r = 2^{32/8} = 2^4 = 16)\). The radix sort requires 8 iterations to sort these integers. For iteration \(i\), it sorts the keys based
on their $i^{th}$ least significant digits. Usually, the radix $r$ is selected to obtain the best performance and highly relies on the size of the keys to be sorted.

Radix sort is a stable non-comparison sorting algorithm, where stable means the output preserves the relative input order for keys with same value. However, because of randomly accessing memory, the performance of radix sort is often worse than quick sort for the sequential version.

### 5.2.1 Serial Radix Sort

The main idea of radix sort is very straightforward. It divides keys into multiple digits, and uses a stable sorting algorithm to sort one digit in each iteration from the least significant digit. Figure 5.4 is an example showing how radix sort operates on six 4-digit numbers.

![Figure 5.4. The Operation of Radix Sort on 6 4-digit Numbers](image)

The counting sort is an alternative usually used to sort each digit. Suppose $n$-bit unsorted keys are considered as $m$-digit numbers, where each digit is composed of $k$ bits, so there are $r = 2^k$ possible values for each digit. And the counting sort will be called $t = n/k$ times to sort the keys. In each iteration, $r$ buckets are required with one for each possible digit value. The functions of these buckets are counting how many times each possible value appears, and giving each key its corresponding temporary
position in the output of each iteration. Pseudocode is given in Algorithm 5.2.1 [22], where \( s \) is the size of the unsorted array \( in \), \( d[i] \) is the given digit value of the \( i^{th} \) input element. The sorted keys will be written to the array \( out \).

**Algorithm 5.2.1 COUNTING-SORT**

/* HISTOGRAM-KEYS */

for \( i \leftarrow 0 \) to \( r - 1 \) do
    
    \( \text{bucket}[i] \leftarrow 0 \)

end for

for \( i \leftarrow 0 \) to \( s - 1 \) do
    
    \( \text{bucket}[d[i]] \leftarrow \text{bucket}[d[i]] + 1 \)

end for

/* SCAN-BUCKETS */

\( \text{sum} \leftarrow 0 \)

for \( i \leftarrow 0 \) to \( r - 1 \) do
    
    \( \text{val} \leftarrow \text{bucket}[i] \)
    
    \( \text{bucket}[i] \leftarrow \text{sum} \)
    
    \( \text{sum} \leftarrow \text{sum} + \text{val} \)

end for

/* RANK-AND-PERMUTE */

for \( i \leftarrow 0 \) to \( s - 1 \) do
    
    \( p \leftarrow \text{bucket}[d[i]] \)
    
    \( \text{out}[p] \leftarrow \text{in}[i] \)
    
    \( \text{bucket}[d[i]] \leftarrow p + 1 \)

end for

Counting sort consists of three phases. In the HISTOGRAM-KEYS phase, all the elements of the *buckets* are initially set to 0. Then, for iteration \( i \), the value of the given digit of the \( j^{th} \) element is used as an offset to locate the corresponding buckets, then the value of this bucket is incremented by 1. After all keys have been looped over, \( \text{bucket}[i] \) is the number of keys with value \( i \) for the given digit. The sum of the *buckets*
is equal to the size of the input.

For the phase of SCAN-BUCKETS, the scan algorithm which we have discussed in section 5.1 is performed on the buckets. After the execution of the scan, the first position in the output for all possible values of a given digit is known.

In the phase of RANK-AND-PERMUTE, each key with digit \( i \) is placed in the output position based on the value of \( \text{bucket}[i] \). Then the value of \( \text{buckets}[i] \) is increased by 1 to record the next location for the following key with the same value \( i \).

### 5.2.2 CUDA Parallel Radix Sort

Since radix sort uses counting sort for each pass, it is one of the sorting algorithms easy to be parallelized. For the serial radix sort, HISTOGRAM-KEYS and RANK-AND-PERMUTE phases have no data dependences, so it is very straightforward to parallelize them. But due to the fact that the position in the output of each key relies on the number of previous keys with the same value, the phase of SCAN-BUCKETS is our central concern. By replacing the second phase with the parallel scan as we discussed in section 5.1, this constraints can be eliminated.

To design an efficient radix sorting algorithm for CUDA, we should first consider the size of the sub-task for each thread block. As we know, accessing shared memory is much faster than accessing global memory, so all the operations on the data should be performed on shared memory instead of global memory to achieve better performance. Because of the relatively small size of shared memory (16 KB for G80, 48 KB for GTX480), and with extra overhead taken into account, processing 1024 elements in each thread block is the best choice. Therefore, for a sequence with the size of power-of-2, the number of thread blocks is in proportion to the sequence size.

Intuitively, assigning one element for each thread would be a natural choice,
because each thread only has to access global memory twice (loading and storing). But from Table 5.1, we can find that assigning two elements per thread is more efficient. Table 5.1 comes from the result of our experiment running 10000 times. As Table 5.1 shows, for 1024 thread blocks of 1024 elements assigned to each thread block, the execution time for loading all the elements from global memory to shared memory then writing back varies with the threads number. Due to the fact that the overhead of creating 1024 threads can not be totally hidden by the relatively small global memory access time. 512 threads in each thread block for 1024 elements provides the overall best performance.

<table>
<thead>
<tr>
<th>Number of Threads per Block</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.886704s</td>
</tr>
<tr>
<td>256</td>
<td>0.934335s</td>
</tr>
<tr>
<td>512</td>
<td>0.564985s</td>
</tr>
<tr>
<td>1024</td>
<td>0.778919s</td>
</tr>
</tbody>
</table>

Table 5.1. The Running Time for Transferring $2^{20}$ Keys to/from Shared Memory

There are eight passes in total, and four phases for each pass in the radix sorting algorithm:

- An unsorted sequence is divided into $p$ chunks with 1024 elements each chunk. Each thread block loads its chunk into shared memory, then performs 4 iterations of 1-bit split sorting operation;

- Each thread block generates its own $2^4$-entry local histogram, and writes the histogram along with the sorted chunk back to global memory. The $p \times 2^4$
local histogram table is stored in column-major in global memory as shown in Figure 5.5;

- $p$ thread blocks cooperatively perform the parallel scan on this histogram table to produce the global position for each possible digit value;

- The $i^{th}$ threads block writes its elements to the correct position in global memory according to the $i^{th}$ column in the global histogram table.

![Figure 5.5. The Layout of the Histogram Table](image)

The 1-bit split operation "splits" keys with the value of 0 for a given bit to the left side of the output, and the keys with 1 value to the right side. Hence, for n-bit integers, a 1-bit split requires $n$ iterations to sort them. If we treat each key with integer type as a 32-digit number, the split operation will need to access global memory 64 times. Since transferring data to and from global memory is relatively expensive, we should avoid this kind of strategy. To perform split operation efficiently, we consider each key as an 8-digit number, and each digit consists of 4 bits. Although the split operation will still be called 32 times, it needs to access global memory 2
times for each 4 split operations on the shared memory. The pseudocode is given in Algorithm 5.2.2. And its operation is shown in Figure 5.6.

**Algorithm 5.2.2 1-bit Split Operation**

```plaintext
for all t from 0 to n – 1 in parallel do
    if the given bit of in[t] = 0 then
        flag[t] ← 1
        b[t] ← 0
    else
        flag[t] ← 0
        b[t] ← 1
    end if
end for
f ← flag[n – 1]
perform parallel scan on flag
totalF alses ← f + flag[n – 1]
for all t from 0 to n – 1 in parallel do
    d[t] ← b[t] ? (t – flag[t] + totalF alses) : flag[t]
end for
for all t from 0 to n – 1 in parallel do
    out[d[t]] ← in[t]
end for
```

The efficient parallel scan given in section 5.1 can only operate on an array within a single thread block. And the size of the array can not exceed twice the number of maximum threads in a block, since each thread processes two elements at most. For the G80 and GTX 480 GPUs, the maximum number of elements are 1024 and 2048 respectively. However, the histogram table with size of $p \times 2^4$ is usually larger than these numbers, so we need to extend the efficient parallel scan algorithm so that it could operate on relative large arrays.

The input array is partitioned into tiles with suitable size which can be privately
scanned by each thread block, then the first phase of the scan will be conducted. When it is done, we write the last element of each tile which is called total sum into an auxiliary array, then continue the second phase of the scan. Next, the efficient parallel scan is performed on this auxiliary array to generate the localsum array. At last, each element $i$ in the localsum is added to all elements of tile $i$ to get the scanned output. The illustration of this procedure is shown in Figure 5.7.

### 5.2.3 Parallel Radix Sort on the Cell B.E

The implementation of the Cell radix sorting algorithm for integer keys uses the SPE SIMD instructions. Since each DMA can only transfer 4K integers at a time between the PPE and a SPE, the communication time is significant, and can not be ignored. Thus, the double buffering technique is also adopted to hide this penalty.
Figure 5.7. The Extended Parallel Scan Algorithm for a Large Array

We treat integer keys as 8-digit numbers which is the same as the CUDA radix sort, so there are still 8 iterations of counting sort. In each pass, the keys are sorted based on its $2^4 = 16$ possible values instead of using the 1-bit scan operation with 4 iterations.

For each pass, the main task of the PPE is dividing the sequence into equal-sized chunks, and sending them to 4 SPEs. When the operation is done by the SPEs, it performs serial scan, then writes each element of all chunks to its correct position. The job of each SPE is receiving data chunks, sorting them based on a given digit (4 bits), and writing it back with its corresponding local histogram.

- Given an integer sequence $\text{input}$ with size of power of two, the PPE partitions it into $n$ chunks with 4096 elements each, and sends $n/4$ chunks to each SPE.
- Each SPE receives a chunk by the double buffered DMA.
• Each SPE moves all the elements to the 16 buckets using \textit{SIMDshift} and \textit{SIMDand} instructions, and records the number of elements in each bucket.

• Each SPE packs these buckets to a chunk, and writes it back to PPE with double buffered DMA.

• After all chunks have been written back, the PPE builds up the global histogram, and writes each element to its correct position in memory.

Since the Cell B.E. architecture uses 128-bit vector register file, then each SPU can perform four efficient \textit{shift} and \textit{and} operations for 32-bit integers simultaneously by using SIMD instructions. In order to improve the system performance, the loop structure can be unrolled by at least a factor of four to exploit the vector register file.

There are two reasons that the CUDA radix sorting algorithm is not appropriate for the Cell architecture. Firstly, for the counting sort, it accesses memory randomly in the phase of key permutation. This is impossible for Cell since the DMA transfer issued by both the PPE and the SP needs to be 128-bit aligned. The size of the data which can be transferred has to be 1, 2, 4, 8, 16 or multiples of 16 bytes. Also, the maximum size of transferred data is 16 KB.

Secondly, though both the SPE and the thread block are the basic units to solve the sub-problem, we can not treat a SPE as the analog of a thread block in CUDA programming. There are hundreds even thousands of thread blocks which can be generated in a CUDA program. In contrast, we only have eight SPEs in one Cell processor. The SPE achieves parallelism using SIMD instruction. Therefore, the well-designed parallel scan algorithm executed by multiple threads is useless for Cell radix sort.
5.2.4 Results

The radix sorting algorithm is implemented on both CUDA GPU and the Cell Broadband Engine Architecture. The CUDA radix sort is executed on the platform of the nVIDIA GTX 480 GPU. The Cell B.E radix sorting algorithm is run on the Sony playstation 3 by using 4 out of 6 SPEs. For better comparison, the serial version of radix sort described in Algorithm 5.2.1 is also implemented. And its performance is also given.

<table>
<thead>
<tr>
<th>Number of Elements</th>
<th>Serial Radix</th>
<th>CUDA</th>
<th>Cell B.E.</th>
</tr>
</thead>
<tbody>
<tr>
<td>65536</td>
<td>0.112145s</td>
<td>0.004195s</td>
<td>0.023989s</td>
</tr>
<tr>
<td>131072</td>
<td>0.231582s</td>
<td>0.006578s</td>
<td>0.040264s</td>
</tr>
<tr>
<td>262144</td>
<td>0.466402s</td>
<td>0.011950s</td>
<td>0.072786s</td>
</tr>
<tr>
<td>524288</td>
<td>0.940496s</td>
<td>0.022037s</td>
<td>0.138929s</td>
</tr>
<tr>
<td>1048576</td>
<td>1.881464s</td>
<td>0.042267s</td>
<td>0.271606s</td>
</tr>
</tbody>
</table>

Table 5.2. Running Time of the Radix Sorting Algorithm

Table 5.2 shows the timing performances of each implementation of radix sort. Each radix sort with different data size is executed 100 times, then the average is calculated. The relationship between the data set size and the running time is shown in Figure 5.8.

As Figure 5.8 shows, the CUDA radix sort outperforms the Cell sort. This is determined by the fact that Cell B.E. and CUDA are two totally different architectures.

The primary reason for the outformance of CUDA is memory latency. In Cell B.E. architecture, accessing the cache-like local store of the SPE is very fast, but first the data must be transferred from memory to local store by DMA. The communication
time is the most important concern of ours. Even though we can use double buffer or triple buffer to overlap certain part of data transfer time and computing time in SPE, but the PPE can build up the global histogram only after all the chunks are written back. In other words, the PPE has been idle since it distributed the task. And it can not execute the next task before all the SPEs finish its job. However, this is not the case for CUDA. Like the Cell B.E architecture, each thread block needs to load data from global memory to its private shared memory before the computations begin, but this memory operation can be performed by multiple threads simultaneously within a few clock cycles. That means the data transfer time is almost hidden by the large amount of thread-level parallelism. Moreover, the overhead of the threads creation and synchronization is extremely trivial for the CUDA program.

The different parallel strategy is the secondary reason. Each SPE has a 128-bit register file, and four 32-bit integers can be stored in a single register. Thus, each
SIMD instruction can only operate four integers within one SPE at any given time. The entire sixteen integer values can be processed when using four SPEs. In contrast, CUDA uses SPMD mode to implement parallelism. By defining the number of threads in a block, each thread can operate on a different scalar integer value simultaneously. Therefore, the maximum number of integer values which can be processed in each threads block can vary from 1 to 1024 in GTX 480 GPU at any given time.
CHAPTER 6
PARALLEL SAMPLE SORT

Sample sort is a comparison based sorting algorithm, which is known as a generalization of quick sort. It is very suitable for multi-cores and many-cores processors. In this chapter, we re-design, implement, and manually auto-tune the parallel sample sort on both Cell Broadband Architecture and CUDA GPU Architecture. To achieve high performance, these algorithms eliminate the expensive key comparison operations using bit shift operations. The experiments showed that, for the CUDA sample sort, we have obtained an increase in speed of up to 68 times over the sequential quick sort. And the Cell B.E. sample sort is 4 times faster than the sequential quick sort.

6.1 Introduction

In general, for a serial external sorting algorithm, the combination of an internal sort and some other phases is a common strategy to solve the sorting problem with large input size. Firstly, in the phase of key partition, the input with a size exceeding the available storage capacity is partitioned into tiles. Each divided tile can vary from only one element to as large as the memory size. Next, one selected internal sorting algorithm operates on these tiles, and generates the sorted tiles. We call this stage local computation phase. Then, in the key combination phase, the resulting sorted tiles are written back to main memory.

Due to limitations given by the data dependences, these phases have to be al-
ternately executed several times to generate the sorted output. The efficiency is usually
gained from making the size of each tile as large as possible that still can be resident
in the memory. Because of the similar phase patterns between the sample sort and
the external sort, the designing strategy for the external sort is still applicable for the
sample sort. However, rather than using one processor sorting each tile, the sample
sort distributes tiles to different processors to be sorted, then writes the results back.

6.2 Sample Sort

The sample sort, also called splitter sort, drives this style of alternating phases to the
extreme, since each phase is executed only once. The principle behind the sample
sort is quite simple. In the phase of key partition, $s$ samples could be selected from
an unsorted input with $n$ items, then these samples are sorted by a sequential sorting
algorithm. Because the number of samples is relatively small, the chosen algorithm for
sorting these samples has no influence on the performance of the sample sort. Then,
$p - 1$ elements called splitters are picked up from the sorted samples array, where $p$ is
usually the number of processors. After defining $p$ buckets by the $p - 1$ splitters, each
key of the input sequence will be written to the correct bucket. Finally, each bucket is
sent to a different processor.

In the local computation step, each processor receives one bucket and stores
it to its private memory. Since this phase dominates the overall performance of the
sample sort, the local sorting algorithms should be carefully designed. Usually, the
best candidate for the local sorting algorithm is recursive quick sort, because quick
sort is well-known to be the fastest serial-sorting algorithm in practice. And its average
time complexity is $O(n \log n)$.

After the local sorting is done, all the buckets are sent back to main memory
in the key combination phase. If these buckets can be written from the local memory back to the original input position directly, this algorithm is called *in-place* algorithm. On the other hand, if these buckets are first transferred from local memory to some other position in main memory, then overwrite the input, we call this algorithm *out-in-place* algorithm. Figure 6.1 shows an example of in-place sample sort which sorts 24 elements.

Figure 6.1. An Illustration of the sample Sort

As shown in Figure 6.1, we can find that sample sort is not a balanced sorting algorithm; that is, the workload distributed to each processor is not balanced. So the goal for the phase of key partition is to make buckets as balanced as possible. Therefore, sample selection and the sample size selection serves this goal. One of the strategies is randomly generating some samples and sorting them, then selecting $p - 1$ splitters. However, using the other strategy, we can obtain more balanced buckets: a position of the input array is randomly produced, then a certain number of consecutive
elements are selected from this location as a part of the samples. After several repetitions of this process, we get the final sample array. Due to the fact that the samples can reflect the distribution of their population, the splitters coming from these samples can make the buckets more balanced than the previous method. Intuitively, the more the samples, the more balanced the buckets are. However, there is a tradeoff between the sample size and the balance. Since with the increment of the sample size, the performance of the sample sort will be degraded.

While the balance problem is one concern for us, the large number of conditional branch is the other. Given $p - 1$ splitters, each key would be moved to the correct bucket under $p/2$ operations of conditional branch on average. Assume the size of the input sequence is $n$, the sample sort algorithm has to perform $p/2 * n = O(n)$ conditional branches in the key partition phase. With the increment of the value of $n$, its performance will be degraded drastically since the conditional branch may generate significant penalty.

When a conditional branch comes into the first stage of the pipeline, the instruction which will be executed next must be predicted since the result of this branch will not be known until it reaches the third stage. If the prediction hit, there will not be any penalty. If the prediction misses, the pipeline must be flushed because of its incorrect execution path. And the penalty produced by the missed prediction is in proportion to the pipeline depth.

For the input sequence with a integer type, conditional branches can be eliminated by the bitwise shift operation. Also, some steps such as sample selection, sample sorting and splitter selection can be skipped as well. The only thing we need to do in the key partition phase is to shift each key to given bits, and distribute it to the correct bucket. For instance, suppose there are 8 processors available, and the same number
of buckets have been established. For the input sequence of 32-bit integers, we need to
shift each key $31 - 3 = 28$ bits right (the most significant bit is sign bit). After these
shift right operations, the value of each key will fall into the range between 0 and 7.
Then we use each altered value as the offset for the buckets, and distribute the key to
its corresponding bucket.

In essence, some special splitters are still chosen to put keys into different
buckets. And the number of splitters is one less than the number of processors in the
regular sample sort. From the above example, we can find that there are 7 splitters: $2^{29}$,
$2^{30}$, $2^{29} + 2^{30}$, $2^{31}$, $2^{29} + 2^{31}$, $2^{30} + 2^{31}$, $2^{29} + 2^{30} + 2^{30}$.
By using these splitters with the
pattern of power-of-two, the key partition phase based on comparison is transformed
to the phase based on bitwise shift. Even though the operation of the shift will make
the buckets quite unbalanced, this problem can be totally hidden by the improvement
of the performance by using bitwise shift operations.

6.3 CUDA Parallel Sample Sort

In order to map the sorting problem to the CUDA GPU architecture efficiently, the
task needs to be divided into data-independent subproblems which can be solved in
parallel by thread blocks. So the input sequence with size $n$ must be partitioned into
$b = n / (t \times s)$ blocks, where $t$ threads are assigned to each block with each thread
processing $s$ elements. Because the phase of local computation plays a vital role in the
performance of sample sort, the average number of elements within a block on average
first needs to be determined according to the given number $n$. Then the specific values
of $b$, $t$, and $s$ can be decided.

According to the discussion of section 6.2, we know that the performance of the
sample sort can be improved by replacing conditional branches with the shift operation.
This is one benefit provided by the bitwise shift operation. On the other hand, it also implies that some of these most significant bits of the input sequence have been sorted by the shift operation. For example, suppose we define 16 buckets. When the key partition phase is done, each input element is moved to the correct bucket. The keys with the same three most significant bits are resident in a bucket having the same offset. Thus, in the next step, we need only to sort the keys locally based on their 28 least significant bits. Moreover, CUDA provides atomic write functions. And the parallel scan is a mature technique applied in the CUDA GPU platform. All of these features make the counting sort the best candidate for the local computation phase.

As we know, counting sort includes three phases: HISTOGRAM-KEYS, SCAN-BUCKETS, and RANK-AND-PERMUTE. The HISTOGRAM-KEYS can be parallelized by the atomicAdd instruction as the following code shows. Here, each thread has its unique thid, 64 is the threads number in each group, and bucket_num is the number of defined buckets.

```c
grp_num = thid / 64;
p = data[thid] >> 31 - log2(bucket_num);
atomicAdd(&temp[p * 4 + grp_num], 1);
```

The function atomicAdd reads the old value located at the address in global or shared memory, computes \((old + val)\), and writes the result back to the same address, then returns the old value. These four operations are performed in one atomic transaction [15]. When multiple threads try to change the number of the elements in the same bucket simultaneously, this function guarantees the altered value is correct. Therefore, the shift and addition operations can be performed at the same time by the threads within a block.

The phase of the HISTOGRAM-KEYS can be parallelized by the atomic write
transaction, but this is not the case for the RAND-AND-PERMUTE phase. Since there is no atomic read operation supplied by the CUDA, the reading operation from the scanned array is needs to be executed sequentially. So the phase of RAND-AND-PERMUTE is a bottleneck of system performance. Because of this reason, each bucket can not hold too many keys. We empirically choose 128 elements on average for each bucket. With the growth of the input size, we increase the number of the buckets manually. And each bucket has a 128 elements on average.

The CUDA sample sorting algorithm can be described in 3 phases with the given input size \( n \):

Each thread block defines a \( k \times h \)-entry local histogram and loads \( t \) keys into its shared memory, where \( n/h = 128 \) on average. The allocated shared memory for the local histogram and the loaded keys can not exceed 16 KB for G80 GPU, or 48 KB for GTX 480 GPU. Each thread computes its group number according to its thread ID, and executes the shift operation on the keys. Since the local histogram is a two dimensional array, each thread uses the shifted value as a row index and its group number as a column index to find the target entry, then increases the value of that entry by one with the atomic write operation. The local parallel scan is executed and the total sum of each row in the local histogram is written to one column of an auxiliary table. The auxiliary table is stored in column-major and located in global memory. It is a two-dimensional array with size of \( b \times h \), where \( b \) is the number of thread blocks created in this phase.

The extended parallel scan algorithm discussed in Figure 5.7 is performed on the \( b \times h \) auxiliary table to produce the global histogram. Each thread block loads one column from the global histogram, indexed by its block ID to shared memory, and adds it on each column of the \( k \times h \)-entry local histogram. Each thread block now
has the knowledge of the bucket position for its keys, so it distributes each key to the correct bucket.

Each thread block loads one bucket to its shared memory, and performs a local radix sorting algorithm which consists of several iterations of counting sort. For the radix sort, the number of bits of the keys which can be sorted in each pass and the number of times the counting sort is needed are all determined by the buckets count. For instance, if there are $2^7 = 128$ buckets available, the counting sort needs to be looped 3 times sorting 8 bits of the integer keys in each iteration. After the sorting, all the threads in each block cooperatively write their bucket back to the global memory.

In the second phase, before a key can be sent to the correct bucket, its corresponding position must be read from shared memory. Due to the fact that CUDA does not support the atomic read transaction, the step of dispatching keys cannot be done in parallel. If we try to parallelize it with several threads, it is very possible that multiple threads read the same position for different keys, which will cause a writing conflict. This is the primary reason we use $k$ sets of the local histogram rather than only 1 set in the previous phase. By using a $k \times h$ local histogram, distributing keys to the global memory can be performed by $k$ threads simultaneously. Since $k$ threads access the global memory in a random way, their memory accessing requests cannot be coalesced. Moreover, due to the small size of the shared memory, the value of $k$ cannot be very large. Thus, each thread still needs to access the global memory many times to write the keys back. Therefore, dispatching keys to the buckets is the first bottleneck for the CUDA sample sort.

For the counting sort in the local computation stage, each key needs to be sent to the correct shared memory position after the parallel scan is performed. For the same reason that CUDA does not supply atomic read for us, the operation of reading
a position for each key must be done sequentially. Although each key is written to the
shared memory instead of the global memory at this time, it is the other bottleneck for
the sample sort.

6.4 Cell B.E. Sample Sort

In the Cell B.E. radix sorting algorithm, the PPE divides the input sequence into tiles
with the size of 4096 each, and sends them to its SPEs to perform radix sort. After
the radix sort is complete, each SPE writes its tiles back to the main memory. For the
sorting process in the SPEs, the SIMD instruction and double buffer can be utilized
to improve system performance. Since the Cell radix sorting algorithm sorts only 4
bits for the keys each time, it requires 8 iterations to complete the sorting process.
Unlike the radix sort, the Cell B.E. sample sort only needs one iteration to produce the
sorted output. Before the local computation phase starts, each key has been moved to
the correct bucket. This operation eliminates the data dependences among the buckets
before the execution of the sorting algorithm. So after the sorting, the output can be
generated.

The Cell B.E. sample sort can be described in the following stages:

- Given an integer input sequence with the size of $n$, the PPE distributes each key
  into one of $b$ buckets, and sends these buckets to the SPEs.

- Each SPE receives a bucket by the double buffered DMA.

- Each SPE performs quick sort on this bucket.

- Each SPE sends the sorted bucket to the main memory by the double buffered
  DMA, then go to the second step.
• The PPE discards the padding elements in each bucket, and concatenates these buckets to form the sorted output.

In the first step, due to the constraint of the DMA transfer, we cannot define the size of each bucket arbitrarily. As we discussed in chapter 3, for each DMA transfer in the Cell B.E. architecture, the transferred data size needs to be 1, 2, 4, 8, 16 or multiples of 16 bytes. In order to make the data transfer more efficient, we specify the bucket size to be 16 KB, which is the maximum DMA transfer size as well. Moreover, we need to guarantee that the number of elements cannot exceed 4096 in each bucket. The PPE uses the SIMD shift right instruction to move each key into the correct bucket, and also needs to compute the number of keys in each bucket. The following pseudocode shows the associative operations in the first stage. Here, the \texttt{vec\_splats} function returns a vector with all of its elements being the given scalar value.

\textbf{Algorithm 6.4.1 The First Phase of the Cell B.E Sample Sort}

\begin{verbatim}
vector int *vs ← (vector int *)in
num_buckets ← n / 2048
vector int shft ← vec\_splats(31 - log(num\_buckets))
for i ← 0 to (num\_buckets - 1) do
  count\_buckets[i] ← 0
  malloc(buckets[i], 4096 * sizeof(int))
end for
for i ← 0 to n/4 do
  vector int vd ← vec\_sr(vs[i], shft)
  buckets[vd[0]][count\_buckets[vd[0]++]] ← vs[i][0]
  buckets[vd[1]][count\_buckets[vd[1]++]] ← vs[i][1]
  buckets[vd[3]][count\_buckets[vd[3]++]] ← vs[i][3]
end for
\end{verbatim}
As we know, the sample sort is an unbalanced sorting algorithm. The number of elements in each bucket can not be known until the run time. So we need another data structure to store it in the program. And this is the main function of the array count_buckets. After the PPE sends the array of buckets along with the count_buckets to the SPEs, the SPEs perform the recursive quick sort only on the first count_buckets[i] elements for buckets[i], then write the whole bucket back.

Due to the data size limitation of the DMA transfer, the padding elements are necessary in each bucket. Therefore, when each bucket is written back to main memory, the PPE first discards them, then concatenates these buckets to generate the sorted output.

Although the Cell B.E. processors support SIMD instructions, key distribution is a bottleneck for system performance because this operation is done by the PPE sequentially. With the increment of the input size, the performance will be degraded significantly. On the other hand, when all the buckets are written back to the main memory, the PPE will access main memory too many times to discard the padding elements and concatenate these buckets to generate the output. Thus, this is another bottleneck for the Cell B.E. sample sort.

6.4.1 Results

The sample sorting algorithm is implemented on both CUDA GPU and the Cell B.E. architecture. The CUDA sample sort is executed on the platform of the nVIDIA GTX 480 GPU. And the Cell B.E radix sorting algorithm is run on the Sony playstation 3 by using 4 out of 6 SPEs. For better comparison, the serial version of the quick sort is also implemented. The following table shows the results.

Table 6.1 shows the timing performances of the parallel sample sort algorithms
Table 6.1. Running Time of the Parallel Sample Sort and Serial Quick Sort together with the sequential quick sort. Each sorting algorithm with different data size is executed 100 times, then the average timing performance is calculated. The relationship between the data set size and the running time is shown in Figure 6.2.

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Serial Quick Sort</th>
<th>CUDA Sample Sort</th>
<th>Cell Sample Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>65536</td>
<td>0.048718s</td>
<td>0.001142s</td>
<td>0.022649s</td>
</tr>
<tr>
<td>131072</td>
<td>0.102867s</td>
<td>0.001901s</td>
<td>0.037609s</td>
</tr>
<tr>
<td>262144</td>
<td>0.220722s</td>
<td>0.003754s</td>
<td>0.067547s</td>
</tr>
<tr>
<td>524288</td>
<td>0.470589s</td>
<td>0.007245s</td>
<td>0.128581s</td>
</tr>
<tr>
<td>1048576</td>
<td>0.980413s</td>
<td>0.014418s</td>
<td>0.251127s</td>
</tr>
</tbody>
</table>

Figure 6.2. The Comparison of Parallel Sample Sort and the Sequential Quick Sort

As Figure 6.2 shows, the CUDA sample sort outperforms the Cell sort. The reasons are the same as we discussed in section 5.2.4. The different memory accessing
times is the first reason. For the Cell B.E. sample sort, the PPE has to distribute all the keys to buckets by using SIMD instruction. Since this task involves the indirect addressing mode, the cache miss penalty will dominate this operation. So the PPE needs to access main memory approximately $O(n)$ times to complete this work. On the contrary, for the CUDA sample sort, all the threads in each block first cooperatively load their tile to the shared memory, and perform the shift operation for the keys, then write them back to global memory. On average, in this process, each thread needs only to access the global memory $O(1)$ times.

The different parallel degree is the second reason. The PPE has a 128-bit register file, so four 32-bit integers can be stored in a single entry. Therefore, by using the SIMD instruction, the PPE can operate on four integers at any given time. For the CUDA GPU, each thread can operate on only one key at any time, because the kernel function does not support the SIMD instructions. However, each thread block can create up to 1024 threads, the number of keys which can be processed in the GPU is much greater than the Cell B.E. architecture at any given time.
CHAPTER 7
CONCLUSION AND FUTURE WORK

This thesis has presented the implementation and the tuning of parallel radix sort and parallel sample sort on the nVIDIA CUDA enabled graphics processors, as well as on the Cell B.E. architecture. From our experimental results, we can summarize that the sorting algorithms on the GPU processors outperforms the parallel sorting algorithm on the Cell B.E. processors.

For the CUDA GPU, the high performance is achieved by exploiting three levels of memory hierarchy. Rather than consuming data resident in the low-speed global memory directly, all the threads in a thread block first cooperatively load the required data into the high-speed shared memory, then process it. This can effectively reduce the average memory access time, especially when the threads need to access the global memory multiple times in one kernel function. Also, by combination consecutive access requests to the global memory issued by the threads within a thread block into a single operation, the memory latency can be further optimized. Unlike the CUDA GPU, the Cell B.E. architecture adopts the unified cache-like local store for its SPEs, and attempts to overlap the overhead of the DMA transfer with the computation through double buffer to obtain better system performance. Before the data can be processed by the SPEs, it must be transferred from the main memory to the local store. Although the memory latency for local store is trivial, the extra cost for DMA transfer can not be ignored. And in most cases, it is usually very difficult to completely hide
the overhead of DMA transfer by the computation.

The many-core GPU exploits fine-grained parallelism. The programming mode for CUDA is SPMD, so a large number of data can be processed simultaneously. This can be reflected from our parallel scan primitive and bitwise shift operation. In contrast, the parallel structure for the multi-core Cell B.E. architecture is more suitable to exploit coarse-grained parallelism. Because of its limited number of processors, even with the help of the SIMD instructions, the total number of integer data which can be processed is trivial when compared to the CUDA at any given time. For the process of building the global histogram in the radix sort, the Cell B.E. implementation has to do it sequentially. In some cases, the Cell B.E. can only process one piece of data at a time.

The other conclusion we can gain from our experiments is that the performance of the sample sort is better than that of the radix sort for both CUDA and Cell B.E. implementations. For the radix sort, the parallel counting sort needs to be invoked eight times to complete the sorting process. But for the sample sort, the similar steps need only to be executed once as the algorithm of sample sort eliminates the data dependences before the sorting procedure begins. Compared with the eight iterations’ execution of the counting sort within the radix sort, the overhead introduced from removing the data dependences of the sample sort is inconsequential.

Although a processor core of the multi-core architecture is more sophisticated than that of the many-core architecture, the latter is the trend for the parallel computing, since the potential improvement performance produced by the many-core architecture would surpass that of the multi-core architecture. However, the parallel computing also presents some new challenges for the programmer. Since sequential algorithms are not appropriate for parallel architectures, almost all of the parallel versions of these
algorithms should be re-designed to achieve higher system performance on parallel architectures. Because of the existence of data dependences, the parallel algorithms are always more difficult to design than their sequential versions.

The future work of this thesis will include tuning and improving the implementations of the radix and sample sorts on parallel architectures while identifying the bottlenecks resident in these algorithms. It is also an interesting future direction to extend our algorithms to sort key-value pairs. Finally, we are also interested in developing efficient parallel scan primitives on the Cell B.E. architecture.
REFERENCES


