PARALLEL SORTING AND MOTIF SEARCH

By

SHIBDAS BANDYOPADHYAY

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2012
© 2012 Shibdas Bandyopadhyay
I dedicate this to my family.
ACKNOWLEDGMENTS

This work would not have been possible without continuous guidance of my advisor Dr. Sartaj Sahni. His ideas and pertinent questions inspired me to explore different ways to solve problems. I’m also grateful to my parents for encouraging and supporting me during all these years.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>8</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>9</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>12</td>
</tr>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>1 MULTI-CORE ARCHITECTURES</td>
<td>13</td>
</tr>
<tr>
<td>1.1 The Cell Broadband Engine</td>
<td>13</td>
</tr>
<tr>
<td>1.2 Graphics Processing Units</td>
<td>16</td>
</tr>
<tr>
<td>2 SORTING ON THE CELL BROADBAND ENGINE</td>
<td>20</td>
</tr>
<tr>
<td>2.1 High-level Strategies For Sorting</td>
<td>20</td>
</tr>
<tr>
<td>2.2 SPU Vector and Memory Operations</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Sorting Numbers</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Single SPU Sort</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1.1 Shellsort variants</td>
<td>24</td>
</tr>
<tr>
<td>2.3.1.2 Merge sort</td>
<td>28</td>
</tr>
<tr>
<td>2.3.1.3 Comparison of single SPU sorting algorithms</td>
<td>37</td>
</tr>
<tr>
<td>2.3.2 Hierarchical Sort</td>
<td>38</td>
</tr>
<tr>
<td>2.4 Sorting Records</td>
<td>41</td>
</tr>
<tr>
<td>2.4.1 Record Layouts</td>
<td>41</td>
</tr>
<tr>
<td>2.4.2 High-level Strategies for Sorting Records</td>
<td>42</td>
</tr>
<tr>
<td>2.4.3 Single SPU Record Sorting</td>
<td>44</td>
</tr>
<tr>
<td>2.4.4 Hierarchical Sorting for Records - 4way Merge</td>
<td>45</td>
</tr>
<tr>
<td>2.4.5 Comparison of Record Sorting Algorithms</td>
<td>49</td>
</tr>
<tr>
<td>2.4.5.1 Run times for ByField layout</td>
<td>50</td>
</tr>
<tr>
<td>2.4.5.2 Run times for ByRecord layout</td>
<td>52</td>
</tr>
<tr>
<td>2.4.5.3 Cross layout comparison</td>
<td>54</td>
</tr>
<tr>
<td>2.5 Summary</td>
<td>55</td>
</tr>
<tr>
<td>3 SORTING ON A GRAPHICS PROCESSING UNIT (GPU)</td>
<td>59</td>
</tr>
<tr>
<td>3.1 Sorting Numbers on GPUs</td>
<td>59</td>
</tr>
<tr>
<td>3.1.1 GPU Sample Sort</td>
<td>61</td>
</tr>
<tr>
<td>3.1.1.1 Step 1–Splitter selection</td>
<td>62</td>
</tr>
<tr>
<td>3.1.1.2 Step 2–Finding buckets</td>
<td>63</td>
</tr>
<tr>
<td>3.1.1.3 Step 3–Prefix sum</td>
<td>63</td>
</tr>
<tr>
<td>3.1.1.4 Step 4–Placing elements into buckets</td>
<td>63</td>
</tr>
<tr>
<td>3.1.2 Warpsort</td>
<td>64</td>
</tr>
</tbody>
</table>
3.1.2.1 Step 1–Bitonic sort by warps .......... 64
3.1.2.2 Step 2–Bitonic merge by warps .......... 65
3.1.2.3 Step 3–Splitting long sequences .......... 65
3.1.2.4 Step 4–Final merge by warps .......... 66
3.1.3 SRTS Radix Sort .......... 66
3.1.3.1 Step 1–Bottom level reduce .......... 68
3.1.3.2 Step 2–Top level scan .......... 69
3.1.3.3 Step 3–Bottom level scan .......... 69
3.1.4 SDK Radix Sort Algorithm .......... 70
3.1.4.1 Step 1–Sorting tiles .......... 71
3.1.4.2 Step 2–Calculating histogram .......... 73
3.1.4.3 Step 3–Prefix sum of histogram .......... 73
3.1.4.4 Step 4–Rearrangement .......... 74
3.1.5 GPU Radix Sort(GRS) .......... 75
3.1.5.1 Step 1–Histogram and Ranks .......... 76
3.1.5.2 Step 2–Prefix sum of tile histograms .......... 79
3.1.5.3 Step 3–Positioning numbers in a tile .......... 80
3.1.6 Comparison of Number Sorting Algorithms .......... 81
3.2 Sorting Records on GPUs .......... 82
3.2.1 Record Layouts .......... 82
3.2.2 High level Strategies for Sorting Records .......... 83
3.2.3 Sample Sort for Sorting Records .......... 84
3.2.4 SRTS for Sorting Records .......... 87
3.2.5 GRS for Sorting Records .......... 89
3.3 Experimental Results .......... 91
3.3.1 Run Times for ByField Layout .......... 92
3.3.2 Run Times for Hybrid Layout .......... 93
3.4 Summary .......... 95
4 MOTIF SEARCH .......... 97
4.1 Introduction .......... 97
4.2 PMS5 .......... 100
4.2.1 Notations and Definitions .......... 100
4.2.2 PMS5 Overview .......... 100
4.2.3 Computing $B_d(x, y, z)$ .......... 101
4.2.4 Intersection of $Q_s$ .......... 104
4.3 PMS6 .......... 105
4.3.1 Overview .......... 105
4.3.2 Computing $B_d(C(n_1, \ldots, n_5))$ .......... 105
4.3.3 The Data Structure $Q$ .......... 109
4.3.4 Complexity .......... 110
4.4 Experimental Results .......... 110
4.4.1 Preprocessing .......... 111
4.4.2 Computing $B_d$ .......... 112
4.4.3 Total Time .................................................. 112
4.4.4 Total Time Plus Preprocessing Time ...................... 113
4.5 PMS6MC ........................................................... 113
  4.5.1 Overview ..................................................... 113
  4.5.2 Outer-Level Parallelism .................................... 114
  4.5.3 Inner-Level Parallelization ................................. 114
    4.5.3.1 Finding equivalence classes in parallel .............. 115
    4.5.3.2 Computing $B_d(C)$ in parallel ...................... 116
    4.5.3.3 Processing $Q$ in parallel ........................... 117
    4.5.3.4 The outputMotifs routine ........................... 119
4.6 Experimental Results ......................................... 120
  4.6.1 PMS6MC Implementation ................................... 121
  4.6.2 PMS6 and PMS6MC .......................................... 121
  4.6.3 PMS6MC and Other Parallel Algorithms ................. 122
4.7 Summary ....................................................... 123

5 CONCLUSION AND FUTURE WORK ............................... 125

REFERENCES ...................................................... 127

BIOGRAPHICAL SKETCH ........................................... 132
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1 Comparison of various SPU sorting algorithms</td>
<td>39</td>
</tr>
<tr>
<td>4-1 Preprocessing times for PMS5 and PMS6</td>
<td>112</td>
</tr>
<tr>
<td>4-2 Total storage for PMS5 and PMS6 in MegaBytes(MB)</td>
<td>112</td>
</tr>
<tr>
<td>4-3 Time taken to compute $B_d(C(n_1, \cdots, n_5))$ and $B_d(x, y, z)$</td>
<td>112</td>
</tr>
<tr>
<td>4-4 Total run time of PMS5 and PMS6</td>
<td>113</td>
</tr>
<tr>
<td>4-5 Total + preprocessing time by PMS5 and PMS6</td>
<td>113</td>
</tr>
<tr>
<td>4-6 Degree of inner and outer level parallelism for PMS6MC</td>
<td>121</td>
</tr>
<tr>
<td>4-7 Run times for PMS6 and PMS6MC</td>
<td>122</td>
</tr>
<tr>
<td>4-8 Run times for PMS5 and PMSPrune</td>
<td>122</td>
</tr>
<tr>
<td>4-9 Total run time of different PMS Algorithms</td>
<td>123</td>
</tr>
<tr>
<td>4-10 Comparing mSPELLER and gSPELLER with PMS6MC</td>
<td>123</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1-1</td>
<td>Architecture of the Cell Broadband Engine</td>
</tr>
<tr>
<td>1-2</td>
<td>NVIDIA’s Tesla GPU</td>
</tr>
<tr>
<td>1-3</td>
<td>Cuda Programming Model</td>
</tr>
<tr>
<td>2-1</td>
<td>Comparisons for mySpu_cmpswap_skew</td>
</tr>
<tr>
<td>2-2</td>
<td>Comb sort</td>
</tr>
<tr>
<td>2-3</td>
<td>Single SPU column-major AA-sort</td>
</tr>
<tr>
<td>2-4</td>
<td>Collecting numbers from a $4 \times 4$ Matrix</td>
</tr>
<tr>
<td>2-5</td>
<td>Column-major to row-major</td>
</tr>
<tr>
<td>2-6</td>
<td>Column-major brick sort</td>
</tr>
<tr>
<td>2-7</td>
<td>Column-major shaker sort</td>
</tr>
<tr>
<td>2-8</td>
<td>Merge sort example</td>
</tr>
<tr>
<td>2-9</td>
<td>Phase 2 of merge sort</td>
</tr>
<tr>
<td>2-10</td>
<td>Phase 3 counters</td>
</tr>
<tr>
<td>2-11</td>
<td>Phase 3 of merge sort</td>
</tr>
<tr>
<td>2-12</td>
<td>Phase 4 counters</td>
</tr>
<tr>
<td>2-13</td>
<td>Phase 4 of merge sort with counters in different columns</td>
</tr>
<tr>
<td>2-14</td>
<td>Plot of average time to sort 4-byte integers</td>
</tr>
<tr>
<td>2-15</td>
<td>SIMD odd-even merge of two vectors</td>
</tr>
<tr>
<td>2-16</td>
<td>SIMD 2-way merge of 2 vectors $v_1$ and $v_2$</td>
</tr>
<tr>
<td>2-17</td>
<td>Plot of average time to sort 1 to 67 million integers</td>
</tr>
<tr>
<td>2-18</td>
<td>The $\text{findmin}$ operation for records</td>
</tr>
<tr>
<td>2-19</td>
<td>The $\text{fields_select}$ operation in $\text{ByField}$ layout</td>
</tr>
<tr>
<td>2-20</td>
<td>The $\text{fields_select}$ operation in $\text{ByRecord}$ layout</td>
</tr>
<tr>
<td>2-21</td>
<td>Shuffling two records in $\text{ByField}$ layout</td>
</tr>
<tr>
<td>2-22</td>
<td>Shuffling two records in $\text{ByRecord}$ layout</td>
</tr>
</tbody>
</table>
Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

PARALLEL SORTING AND MOTIF SEARCH

By

Shibdas Bandyopadhyay

August 2012

Chair: Sartaj Sahni
Major: Computer Engineering

With the proliferation of multi-core architectures, it has become increasingly important to design versions of popular algorithms which exploit different micro-architectural features of these chip multi-processors to gain maximum speed-up compared to a single core processor. In this work, we have developed parallel algorithms for solving two common problems: Sorting of large records and motif search. Most of our work is done on a Cell processor and on Graphics Processing Units (GPU). We proposed a novel parallel merge sort algorithm on a Cell processor. We then used the parallel merge algorithm to sort a large number of records across different cores of Cell processor. We developed a parallel radix sort algorithm to be run on a GPU. We also extended this algorithm to sort a large number of records. Extensive experiments are done to compare the performance of the proposed sorting algorithms with other contemporary algorithms.

The motif search is an approximate pattern search problem in computational biology where a common pattern, albeit with a few mismatches, needs to be found from a set of strings. We first proposed a sequential algorithm for motif search which shows an improvement over the recently proposed algorithms. We then proposed a parallel implementation of the sequential algorithm which exploits the parallelism present in a multi-core system.
CHAPTER 1
MULTI-CORE ARCHITECTURES

During recent years, a great number of high performance multi-core processors are released. Instead of having a single high performance core, effective performance of a processor is increased by having many cores and designing algorithms to take advantage of those cores. Most of them fall into the category of chip multi-processors (CMP) where all cores are packed onto a single chip with a very high speed bus connecting them. In an homogeneous CMP, all cores are of same type while in case of heterogeneous CMPs not all cores are of same type. Typically, the cores have their own small local storage and they are connected to the external memory using some kind of memory interface controller. The maximum performance can only be achieved when these cores are carefully programmed to do a lot of the processing locally and then effectively communicating the results among each other. Depending on the architecture of these cores, one might have to use different programming paradigms and techniques to build code which can be run on these systems. We focus to two different CMPs in this work. The first one is a heterogeneous CMP called Cell Broadband Engine which is most commonly found in Playstation 3 game consoles. The next one is widely popular graphics processing units (GPUs) which are present in almost all the personal computers made nowadays. Both of these CMPs are low cost high performance processors compared to a traditional Intel x86 processor. However, programming Cell and GPUs are a bit different from a typical processor found in a modern PC but they can be programmed using a high level language like C. In this chapter, we talk about the architecture and programming models for Cell and GPUs.

1.1 The Cell Broadband Engine

The Cell Broadband Engine (CBE) is a heterogeneous multicore architecture jointly developed by IBM, Sony, and Toshiba. CBE was originally designed for multimedia and graphics applications. It first appeared in Sony’s Playstation 3 game console. Toshiba
has also used CBE in their High Definition Televisions assisting in decoding multiple high definition streams. The CBE, however, was designed to do any kind of general purpose compute-intensive task. Consequently, IBM built Cell blade servers and also built supercomputers using CBE with more than a petaflop in peak performance. The CBE came into picture when traditional CPUs have hit a point of diminishing return with single core designs and multi-core CPUs were being considered to be next phase of processor evolution. The CBE was a departure from the way CPUs were designed mainly because of the applications for which it was designed. The design decision was to have a set of very simple and fast cores to process data at very high speeds along with a complicated general purpose control core which is capable of running an operating system. To this end, CBE is an in-order processor unlike most modern processors. Programmers and compiler are responsible for any kind of instruction scheduling to optimize the performance. A CBE (Figure 1-1) consists of a Power PC core (PPE), eight Synergistic Processing Elements (SPEs), an on-board memory controller and a flexible IO controller [1]. All these components are connected in a ring called Element Interconnect Bus (EIB). Two CBEs can also be connected to each other using the flexible IO connection. PPE is designed to perform like an in-order CPU capable of fast task switching and dispatching compute intensive works to SPE. SPEs are designed to be very high speed in-order SIMD compute cores without any cache and without any kind of branch prediction logic. Because of their minimalist design, SPEs can run as fast as 4GHz. But, it can only run at that speed, if it has very fast access to the data needed by the instructions being executed. To assist fast execution, each SPE has a local store of size 256 KB with very fast access time like a traditional cache. The local store along with the execution unit is referred to as Synergistic Processing Unit (SPU). SPEs also have a Memory Flow Controller (MFC) which can move data from main memory to local store and vice versa via DMA transfers. MFC also allows many DMA transfers to be scheduled and it runs independent of the execution engine of SPE. Hence, SPE
can continue execution while the data is being moved. To achieve peak performance, programmers need to exploit this concurrency by carefully scheduling data transfers interleaved with execution. SPEs have two different execution pipeline. Programmers or compiler can schedule instructions such that consecutive instructions fall into different pipelines and two instructions are issued in a cycle. As SPE has a SIMD Instruction set, it is also necessary to pack data in vectors to exploit full potential of a SPE. Along with aforementioned programming techniques, the absence of branch prediction logic in an SPE poses a challenge when developing high performance CBE algorithms.

Figure 1-1. Architecture of the Cell Broadband Engine [2]
1.2 Graphics Processing Units

Contemporary graphics processing units (GPUs) are massively parallel manycore processors. NVIDIA’s Tesla GPUs, for example, have 240 scalar processing cores (SPs) per chip [3]. These cores are partitioned into 30 Streaming Multiprocessors (SMs) with each SM comprising of 8 SPs. Each SM shares a 16KB local memory (called shared memory) and has a total of 16,384 32-bit registers that may be utilized by the threads running on this SM. Besides registers and shared memory, on-chip memory shared by the cores in an SM also includes constant and texture caches. The 240 on-chip cores also share a 4GB off-chip global (or device) memory. Figure 1-2 shows a schematic of the Tesla architecture. With the introduction of CUDA (Compute Unified Driver Architecture) [4], it has become possible to program GPUs using C. This has resulted in an explosion of research directed toward expanding the applicability of GPUs from their native computer graphics applications to a wide variety of high-performance computing applications.

![NVIDIA’s Tesla GPU](image)

Figure 1-2. NVIDIA’s Tesla GPU [5]

GPUs operate under the master-slave computing model (see [6] for e.g.) in which there is a host or master processor to which is attached a collection of slave processors. A possible configuration would have a GPU card attached to the bus of a PC. The PC CPU would be the host or master and the GPU processors would be the slaves. The CUDA programming model requires the user to write a program that runs on the host
processor. At present, CUDA supports host programs written in C and C++ only though there are plans to expand the set of available languages [4]. The host program may invoke kernels, which are C functions, that run on the GPU slaves. A kernel may be instantiated in synchronous (the CPU waits for the kernel to complete before proceeding with other tasks) or asynchronous (the CPU continues with other tasks following the spawning of a kernel) mode. A kernel specifies the computation to be done by a thread. When a kernel is invoked by the host program, the host program specifies the number of threads that are to be created. Each thread is assigned a unique ID and CUDA provides C-language extensions to enable a kernel to determine which thread it is executing. The host program groups threads into blocks, by specifying a block size, at the time a kernel is invoked. Figure 1-3 shows the organization of threads used by CUDA.

Figure 1-3. Cuda Programming Model [4]

The GPU schedules the threads so that a block of threads runs on the cores of an SM. At any given time, an SM executes the threads of a single block and the threads of a block can execute only on a single SM. Once a block begins to execute on an SM, that SM executes the block to completion. Each SM schedules the threads in its assigned block in groups of 32 threads called a warp. The partitioning into warps is fairly intuitive
with the first 32 threads forming the first warp, the next 32 threads form the next warp, and so on. A half warp is a group of 16 threads. The first 16 threads in a warp form the first half warp for the warp and the remaining 16 threads form the second half warp. When an SM is ready to execute the next instruction, it selects a warp that is ready (i.e., its threads are not waiting for a memory transaction to complete) and executes the next instruction of every thread in the selected warp. Common instructions are executed in parallel using the 8 SPs in the SM. Non-common instructions are serialized. So, it is important, for performance, to avoid thread divergence within a warp. Some of the other factors important for performance are:

1. Since access to global memory is about two orders of magnitude more expensive than access to registers and shared memory, data that are to be used several times should be read once from global memory and stored in registers or shared memory for reuse.

2. When the threads of a half-warp access global memory, this access is accomplished via a series of memory transactions. The number of memory transactions equals the number of different 32-byte (64-byte, 128-byte, 128-byte) memory segments that the words to be accessed lie in when each thread accesses an 8-bit (16-bit, 32-bit, 64-bit) word. Given the cost of a global memory transaction, it pays to organize the computation so that the number of global memory transactions made by each half warp is minimized.

3. Shared memory is divided into 16 banks in round robin fashion using words of size 32 bits. When the threads of a half warp access shared memory, the access is accomplished as a series of 1 or more memory transactions. Let \( S \) denoted the set of addresses to be accessed. Each transaction is built by selecting one of the addresses in \( S \) to define the broadcast word. All addresses in \( S \) that are included in the broadcast word are removed from \( S \). Next, upto one address from each of the remaining banks is removed from \( S \). The set of removed addresses is serviced by a single memory transaction. Since the user has no way to specify the broadcast word, for maximum parallelism, the computation should be organized so that, at any given time, the threads in a half warp access either words in different banks of shared memory or they access the same word of shared memory.

4. Volkov et al. [7] have observed greater throughput using operands in registers than operands in shared memory. So, data that is to be used often should be stored in registers rather than in shared memory.
5. Loop unrolling often improves performance. However, the #pragma unroll statement unrolls loops only under certain restrictive conditions. Manual loop unrolling by replicating code and changing the loop stride can be employed to overcome these limitations.

6. Arrays declared as register arrays get assigned to global memory when the CUDA compiler is unable to determine at compile time what the value of an array index is. This is typically the case when an array is indexed using a loop variable. Manually unrolling the loop so that all references to the array use indexes known at compile time ensures that the register array is, in fact, stored in registers.
2.1 High-level Strategies For Sorting

As noted in [8], a logical way to develop a sorting algorithm for a heterogeneous multicore computer such as the CBE is to (1) begin with a sorting algorithm for a single SPU, then (2) using this as a core, develop a sort algorithm for the case when data fits in all available cores, then (3) use this multi-SPU algorithm to develop a sort algorithm for the case when the data to be sorted does not fit in the local stores of all available SPEs but fits in main memory. The strategy would be to extend this hierarchical plan to the case where data to be sorted is so large that it is distributed over the main memories of a cluster of CBEs.

An alternative strategy to sort is to use the master-slave model in which the PPU serves as the master processor and the SPUs are the slave processors. The PPU partitions the data to be sorted and sends each partition to a different SPU; the SPUs sort their partition using a single SPU sort; the PPU merges the sorted data from the SPUs so as to complete the sort of the entire data set. This strategy is used in [9] to sort on the nCube hypercube and in [10] to sort on the CBE.

Regardless of whether we sort large data sets using the hierarchical strategy of [8] or the master-slave strategy of [9, 10], it is important to have a fast algorithm to sort within a single SPU. The absence of any branch prediction capability and the availability of vector instructions that support SIMD parallelism on an SPU make the development of efficient SPU sort algorithms a challenge. SPUs also have two instruction pipelines which make them capable of issuing two instructions in the same cycle if they fall in different pipelines. It pays to hand-tune the generated assembly code so that two instructions are issued in as many of the cycles as possible.
2.2 SPU Vector and Memory Operations

We shall use several SIMD functions that operate on a vector of 4 numbers to describe the SPU adaptation of sorting algorithms. We describe these in this section.

In the following, \( v_1, v_2, \text{min}, \text{max}, \) and \( \text{temp} \) are vectors, each comprised of 4 numbers and \( p, p_1, \) and \( p_2 \) are bit patterns. Also, \( \text{dest} \) (destination) and \( \text{src} \) (source) are addresses in the local store of an SPU and \( \text{bufferA} \) is a buffer in local store while \( \text{streamA} \) is a data stream in main memory. Furthermore, Function names that begin with \( \text{spu} \) are standard C/C++ Cell SPU intrinsics while those that begin with \( \text{mySpu} \) are defined by us. Our description of these functions is tailored to the sorting application.

1. \( \text{spu\_shuffle}(v_1, v_2, p) \) \( \cdots \) This function returns a vector comprised of a subset of the 8 numbers in \( v_1 \) and \( v_2 \). The returned subset is determined by the bit pattern \( p \). Let \( W, X, Y, \) and \( Z \) denote the 4 numbers (left to right) of \( v_1 \) and let \( A, B, C, \) and \( D \) denote those of \( v_2 \). The bit pattern \( p = XCCW \), for example, returns a vector comprised of the second number in \( v_1 \) followed by two copies of the third number of \( v_2 \) followed by the first number in \( v_1 \). In the following, we assume that constant patterns such as XYZD have been pre-defined.

2. \( \text{spu\_cmpgt}(v_1, v_2) \) \( \cdots \) A 128-bit vector representing the pairwise comparison of the 4 numbers of \( v_1 \) with those of \( v_2 \) is returned. If an element of \( v_1 \) is greater than the corresponding element of \( v_2 \), the corresponding 32 bits of the returned vector are 1; otherwise, these bits are 0.

3. \( \text{spu\_add}(v_1, v_2) \) \( \cdots \) Returns the vector obtained by pairwise adding the numbers of \( v_1 \) with the corresponding numbers of \( v_2 \).

4. \( \text{spu\_sub}(v_1, v_2) \) \( \cdots \) Returns the vector obtained by pairwise subtracting the numbers of \( v_2 \) from the corresponding numbers of \( v_1 \).

5. \( \text{spu\_and}(p_1, p_2) \) \( \cdots \) Returns the vector obtained by pairwise anding the bits of \( p_1 \) and \( p_2 \).

6. \( \text{mySpu\_not}(p) \) \( \cdots \) Returns the vector obtained by complementing each of the bits of \( p \). Although the CBE does not have a \( \text{not} \) instruction, we can perform this operation using the \( \text{nor} \) function that is supported by the CBE and which computes the complement of the bitwise \( \text{or} \) of two vectors. It is easy to see that \( \text{spu\_nor}(p, v_0) \) where \( v_0 \) is an all zero vector, correctly computes the complement of the bits of \( p \).
7. \text{spu\_select}(v_1, v_2, p) \ldots \text{Returns a vector whose } i\text{th bit comes from } v_1 (v_2) \text{ when the } i\text{th bit of } p \text{ is } 0 (1).\]

8. \text{spu\_slqwbyte}(v_1, n) \ldots \text{Returns a vector obtained by shifting the bytes of } v_1 \text{ } m \text{ bytes to the left, where } m \text{ is the number represented by the 5 least significant bits of } n. \text{ The left shift is done with zero fill. So, the rightmost } m \text{ bytes of the returned vector are 0.}\]

9. \text{spu\_splat}(s) \ldots \text{Returns a vector comprised of 4 copies of the number } s.\]

10. \text{mySpu\_cmpswap}(v_1, v_2) \ldots \text{Pairwise compares the numbers of } v_1 \text{ and } v_2 \text{ and swaps so that } v_1 \text{ has the smaller number of each compare and } v_2 \text{ has the larger number. Specifically, the following instructions are executed:} \]
\[
P = \text{spu\_cmpgt}(v_1, v_2); \]
\[
\text{min} = \text{spu\_select}(v_1, v_2, p); \]
\[
v_2 = \text{spu\_select}(v_2, v_1, p); \]
\[
v_1 = \text{min};\]

11. \text{mySpu\_cmpswap\_skew}(v_1, v_2) \ldots \text{Performs the comparisons and swaps shown in Figure 2-1. Specifically, the following instructions are executed:} \]
\[
\text{temp} = \text{spu\_slqwbyte}(v_2, 4); \]
\[
P = \text{spu\_cmpgt}(v_1, \text{temp}); \]
\[
\text{min} = \text{spu\_select}(v_1, \text{temp}, p); \]
\[
v_1 = \text{spu\_shuffle}(\text{min}, v_1, \text{WXYD}); \]
\[
\text{max} = \text{spu\_select}(\text{temp}, v_1, p); \]
\[
v_2 = \text{spu\_shuffle}(\text{max}, v_2, \text{AWXY});\]

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2-1.png}
\caption{Comparisons for \textit{mySpu\_cmpswap\_skew}}
\end{figure}

12. \text{mySpu\_gather}(v\text{Array}, v_1) \ldots \text{Here } v\text{Array} \text{ is an array of vectors. Let } W, X, Y \text{ and } Z \text{ be the numbers of } v_1. \text{ The function returns a vector whose first number is the first number of } v\text{Array}[W], \text{ its second number is the second number of } v\text{Array}[X], \text{ its third number is the third number of } v\text{Array}[Y], \text{ and its fourth number is the fourth number of } v\text{Array}[Z]. \text{ One implementation of this function first extracts } W, X, Y, \text{ and } Z \text{ from } v_1 \text{ using the function } \text{spu\_extract} \text{ and then executes the code:} \]
\[
\text{temp} = \text{spu\_shuffle}(v\text{Array}[W], v\text{Array}[X], \text{WBWW}); \]
\[
\text{temp} = \text{spu\_shuffle}(\text{temp}, v\text{Array}[Y], \text{WXCC}); \]
\[
\text{return} \text{spu\_shuffle}(\text{temp}, v\text{Array}[Z], \text{WXYD});\]

22
13. \textit{mySpu\_gather12(vArray, v1)} \textellipsis{} This function, which is similar to \textit{mySpu\_gather}, returns a vector whose first number is the first number of \textit{vArray}[W] and whose second number is the second number of \textit{vArray}[X]. The third and fourth numbers of the returned vector are set arbitrarily. Its code is:
\begin{verbatim}
return spu\_shuffle(vArray[W], vArray[X], WBWW);
\end{verbatim}

14. \textit{mySpu\_gather34(vArray, v1)} \textellipsis{} This function, which is similar to \textit{mySpu\_gather12}, returns a vector whose first number is the third number of \textit{vArray}[W] and whose second number is the fourth number of \textit{vArray}[X]. The third and fourth numbers of the returned vector are set arbitrarily. Its code is:
\begin{verbatim}
return spu\_shuffle(vArray[W], vArray[X], YDYY);
\end{verbatim}

15. \textit{mySpu\_gatherA(vArray, v1)} \textellipsis{} This function is similar to \textit{mySpu\_gather} and returns a vector whose first number is the first number of \textit{vArray}[W], its second number is the third number of \textit{vArray}[X], its third number is the first number of \textit{vArray}[Y], and its fourth number is the third number of \textit{vArray}[Z]. The code:
\begin{verbatim}
temp = spu\_shuffle(vArray[W], vArray[X], WCWW);
temp = spu\_shuffle(temp, vArray[Y], WXAA);
return spu\_shuffle(temp, vArray[Z], WXYC);
\end{verbatim}

16. \textit{mySpu\_gatherB(vArray, v1)} \textellipsis{} This too is similar to \textit{mySpu\_gather}. The function returns a vector whose first number is the second number of \textit{vArray}[W], its second number is the fourth number of \textit{vArray}[X], its third number is the second number of \textit{vArray}[Y], and its fourth number is the fourth number of \textit{vArray}[Z]. The code is:
\begin{verbatim}
temp = spu\_shuffle(vArray[W], vArray[X], XDXX);
temp = spu\_shuffle(temp, vArray[Y], WXBB);
return spu\_shuffle(temp, vArray[Z], WXYD);
\end{verbatim}

17. \textit{memcpy(dest, src, size)} \textellipsis{} copies the \textit{size} number of bytes from the local store location beginning at \textit{src} to \textit{dest}.

18. \textit{dmaIn(bufferA, streamA)} \textellipsis{} This function triggers a DMA transfer of the next buffer load of data from \textit{streamA} in main memory into \textit{bufferA} in the local store. This is done asynchronously and concurrently with SPU execution.

19. \textit{dmaOut(bufferA, streamA)} \textellipsis{} This function is similar to \textit{dmaIn} except that a buffer load of data is transferred asynchronously from \textit{bufferA} in the local store to \textit{streamA} in main memory.

2.3 Sorting Numbers

2.3.1 Single SPU Sort

 Recently, three sorting algorithms—AA-sort \cite{11}, CellSort \cite{8} and Merge sort \cite{12}—were proposed for the CBE. AA-sort is an adaptation of comb sort, which was originally
proposed by Knuth [13] and rediscovered by Dobosiewicz [14] and Box and Lacey [15]. CellSort is an adaptation of bitonic sort (e.g., [13]). Both AA-sort and CellSort are based on sorting algorithms that are inefficient on a single processor. Hence, parallelizing these algorithms begins with a handicap relative to the fastest serial sorting algorithms—merge sort for worst-case behavior and quick sort for average behavior. Comb sort is known to have a worst-case complexity that is $O(n^2)$ [16]. Although the best upper bound known for its average complexity is also $O(n^2)$, experimental results indicate an average complexity of $O(n \log n)$ [15, 16]. On the other hand, the average complexity of quick sort is known to be $O(n \log n)$. Since experiments indicate that comb sort runs in about twice as much time on a single processor as does quick sort [16], attempts such as [11] to develop a fast average-case sort, for a single SPU of the CBE that begin with comb sort, are handicapped by a factor of two compared to attempts that begin with quick sort. This handicap is overcome by the CBE adaptation of the Merge sort described in [12].

For integers and floats, the CBE supports 4-way parallelism within a single SPU as 4 integers (floats) may be stored in each of the SPU’s 128-bit vector registers. Hence, we expect at best two-fold speed up over a conventional implementation of quick sort. However, due to possible anomalous behavior resulting from such factors as the absence of branch prediction, we may actually observe a greater speed up [17]. Similarly, attempts such as [8] to develop a fast worst-case sort for a single SPU starting with bitonic sort are handicapped relative to starting with merge sort because the worst-case complexity of bitonic sort is $O(n \log^2 n)$ while that of merge sort is $O(n \log n)$.

### 2.3.1.1 Shellsort variants

Shellsort [13] sorts a sequence of $n$ numbers in $m$ passes employing a decreasing increment sequence $i_1 > i_2 > \cdots > i_m = 1$. In the $j$th pass, increment $h = i_j$ is used; the sequence is viewed as comprised of $h$ subsequences with the $k$th subsequence comprised of the numbers in positions $k, k + h, k + 2h, \cdots$, of the overall sequence, $0 \leq k < h$; and each subsequence is sorted. The sorting of the subsequences done in
each pass is called an \( h \)-sort. While an \( h \)-sort is typically accomplished using insertion sort, other simple sorting algorithms such as bubble sort may also be used. With the proper choice of increments, the complexity of Shellsort is \( O(n \log^2 n) \) [13]. Shellsort variants replace the \( h \)-sort used in each pass of Shellsort with an \( h \)-pass that only partially sorts the subsequences. For example, in an \( h \)-bubble pass we make only the first pass of bubble sort on each subsequence. Since replacing \( h \)-sort by \( h \)-pass in Shellsort no longer guarantees a complete sort, we follow with some simple sorting algorithm such as bubble sort to complete the sort. So, the \( h \)-passes may be viewed as preprocessing passes done so as to improve the performance of the ensuing sort algorithm. In Shellsort, \( i_m = 1 \) is used to assure that the sequence is sorted following the final \( h \)-sort. However, in a Shellsort variant, this assurance comes from the sort algorithm run following the preprocessing \( h \)-passes. So, the \( h \)-pass with \( h = 1 \) is typically skipped. The general structure of a Shellsort variant is:

**Step 1** [Preprocess] Perform \( h \)-passes for \( h = ij \), \( 1 \leq j < m \).

**Step 2** [Sort] Sort the preprocessed sequence

**Comb and AA sort:** Knuth [13] proposed a Shellsort variant in which each \( h \)-pass is a bubble pass (Figure 2-2). This variant was rediscovered later by Dobosiewicz [14] and Box and Lacey [15] named this variant **comb sort**. The increment sequence used by comb sort is geometric with factor \( s \). Dobosiewicz [14] has shown that the preprocessing step sorts \( a[0 : n - 1] \) with very high probability whenever \( s < 1.33 \). As a result, \( s = 1.3 \) is recommended in practice (note that a larger \( s \) requires a smaller number of \( h \)-passes). With this choice, the outer for loop of the second step (bubble sort) is entered only once with high probability and the complexity of comb sort is \( O(n \log n) \) with high probability. Experiments indicate that the algorithm’s average run time is close to that of quick sort [16]. However, the worst-case complexity of comb sort is \( O(n^2) \) [16].

Inoue et al. [11] have adapted comb sort to the CBE to obtain the sort method AA-sort, which efficiently sorts numbers using all 8 SPUs of a CBE. The single SPU
Algorithm combsort(a,n)
{// sort a[0:n-1]
  // Step 1: Preprocessing
  for (h = n/s; h > 1; h /= s) {
    // h-bubble pass
    for (i = 0; i < n-h; i++)
      if (a[i] > a[i+h]) swap(a[i],a[i+h]);
  }
  sorted = false;
  // Step 2: Bubble sort
  for (pass = 1; pass < n && !sorted; pass++) {
    sorted = true;
    for (i = 0; i < n-pass; i++)
      if (a[i] > a[i+1]) {swap(a[i],a[i+1]); sorted = false;}
  }
}

Figure 2-2. Comb sort

version begins with a vector array \( d[0 : r - 1] \) of numbers; each vector \( d[i] \) has 4 numbers. Hence, \( d \) is an \( r \times 4 \) matrix of numbers. This matrix is first sorted into column-major order and then the numbers permuted so as to be sorted in row-major order. Figure 2-3 gives the algorithm for the column-major sort and Figure 2-5 gives the column-major to row-major reordering algorithm.

Algorithm AA(d,r)
{// sort d[0:r-1] into column-major order
  // Step 1: Preprocessing
  for (i = 0; i < r; i++) sort(d[i]);
  for (h = r; h > 1; h /= s) {
    // h-bubble pass
    for (i = 0; i < r-h; i++)
      mySpu_cmpswap(d[i],d[i+h]);
    for (i = r-h; i < r; i++)
      mySpu_cmpswap_skew(d[i],d[i+h-r]);
  }
  sorted = false;
  // Step 2: Bubble sort
  do {
    for (i = 0; i < r-1; i++)
      mySpu_cmpswap(d[i],d[i+1]);
    mySpu_cmpswap_skew(d[r-1],d[0]);
  } while (not sorted)
}

Figure 2-3. Single SPU column-major AA-sort [11]
The column-major to row-major reordering is done in two steps. In the first step, the numbers in each $4 \times 4$ submatrix of the $r \times 4$ matrix of numbers are transposed so that each vector now has the 4 numbers in some row of the result. For simplicity, we assume that $r$ is a multiple of 4. In the second step, the vectors are permuted into the correct order. For the first step, we collect the first and second numbers in rows 0 and 2 of the $4 \times 4$ matrix being transposed into the vector $\text{row02A}$. The third and fourth numbers of these two rows are collected into $\text{row02b}$. The same is done for rows 1 and 3 using vectors $\text{row13A}$ and $\text{row13B}$. Figure 2-4 shows this rearrangement. Then, the transpose is constructed from the just computed 4 vectors.

Figure 2-4. Collecting numbers from a $4 \times 4$ Matrix. A) Initial. B) Rows 02A, 02B, 03A, 03B, top to bottom. C) Transpose.

**Brick sort:** In brick sort, we replace the $h$-bubble pass of comb sort by an $h$-brick pass [18, 19] in which we first compare-exchange positions $i, i + 2h, i + 4h, \cdots$ with positions $i + h, i + 3h, i + 5h, \cdots$, $0 \leq i < h$ and then we compare-exchange positions $i + h, i + 3h, i + 5h, \cdots$ with positions $i + 2h, i + 4h, i + 6h, \cdots$, $0 \leq i < h$. Figure 2-6 gives our CBE adaptation of the preprocessing step (Step 1) for brick sort. Step 2 is a bubble sort as was the case for AA-sort. The bubble sort needs to be followed by a column-major to row-major reordering step (Figure 2-5). It is known that the preprocessing step of brick sort nearly always does a complete sort when the increment sequence is geometric with shrink factor (i.e., $s$) less than 1.22 [18, 19]. Hence, when we use $s < 1.22$, the do-while loop of Step 2 (bubble sort) is entered only once (to verify the data are sorted) with high probability.

**Shaker sort:** Shaker sort differs from comb sort in that $h$-bubble passes are replaced by $h$-shake passes. An $h$-shake pass is a left-to-right bubble pass as in comb
Algorithm transpose(d,r)
{
// Column major to row major reordering
// Step 1: Transpose 4 x 4 submatrices
for (i = 0; i < r; i += 4) {
    // Compute row02A, row02B, row13A, and row13B
    row02A = spu_shuffle(d[i], d[i+2], WAXB);
    row02B = spu_shuffle(d[i], d[i+2], YCZD);
    row13A = spu_shuffle(d[i+1], d[i+3], WAXB);
    row13B = spu_shuffle(d[i+1], d[i+3], YCZD);
    // Complete the transpose
    d[i] = spu_shuffle(row02A, row13A, WAXB);
    d[i+1] = spu_shuffle(row02A, row13A, YCZD);
    d[i+2] = spu_shuffle(row02B, row13B, WAXB);
    d[i+3] = spu_shuffle(row02B, row13B, YCZD);
}
// Step 2: Reorder vectors
for (i = 0; i < r; i++)
    if (!inPlace[i]) {
        current = i;
        next = i/(r/4) + (i mod (r/4))*4;
        temp = d[i];
        while (next != i) { // follow cycle
            d[current] = d[next];
            inPlace[current] = true;
            current = next;
            next = current/(r/4) + (current mod (r/4))*4;
        }
        d[current] = temp;
        inPlace[current] = true;
    }
}

Figure 2-5. Column-major to row-major

sort followed by a right-to-left bubble pass. Figure 2-7 gives our CBE adaptation of shaker sort. The preprocessing step of shaker sort almost always sorts the data when the shrink factor $s$ is less than 1.7.

2.3.1.2 Merge sort

Unlike the Shellsort variants comb, brick, and shaker sort of Section 2.3.1.1 whose complexity is $O(n \log n)$ with high probability, the worst-case complexity of merge sort is $O(n \log n)$. Further, merge sort is a stable sort (i.e., the relative order of elements that have the same key is preserved). While this property of merge sort isn’t relevant when we are simply sorting numbers (as you can’t tell two equal numbers apart), this property is useful in some applications where each element has several fields, only one of which
Algorithm Brick(d,r)

```plaintext
// sort d[0:r-1] into column-major order
// Step 1: Preprocessing
for (i = 0; i < r; i++) sort(d[i]);
for (h = r; h > 1; h /= s) {
    // h-brick pass
    // compare-exchange even:odd bricks
    for (j = i; j < i + h; j++)
        mySpu_cmpswap(d[j], d[j+h]);
    // handle end conditions
    if (j < n - h) {
        // More than 1 brick remains
        end = j + h;
        for (; j < n - h; j++)
            mySpu_cmpswap(d[j], d[j+h]);
    } else end = r;
    while (j < end) {
        mySpu_cmpswap_skew(d[j], d[j+h-n]);
        j++;
    }
    // compare-exchange odd:even bricks beginning with i = h
    // similar to even:odd bricks
    // Step 2: Bubble sort
    // same as for AA-sort
}
```

Figure 2-6. Column-major brick sort

Algorithm Shaker(d,r)

```plaintext
// sort d[0:r-1] into column-major order
// Step 1: Preprocessing
for (i = 0; i < r; i++) sort(d[i]);
for (h = r; h > 1; h /= s) {
    // h-shake pass
    // left-to-right bubble pass
    for (i = 0; i < r-h; i++)
        mySpu_cmpswap(d[i], d[i+h]);
    for (i = r-h; i < r; i++)
        mySpu_cmpswap_skew(d[i], d[i+h-r]);
    // right-to-left bubble pass
    for (i = r-h-1; i > 0; i--)
        mySpu_cmpswap(d[i], d[i+h]);
    for (i = r-1; i >= r - h; i--)
        mySpu_cmpswap_skew(d[i], d[i+h-r]);
}
// Step 2: Bubble sort
// Same as for AA-sort
```

Figure 2-7. Column-major shaker sort
is the sort key. The Shellsort variants of Section 2.3.1.1 are not stable sorts. On the down side, efficient implementations of merge sort require added space. When sorting numbers in the vector array $d[0 : r - 1]$ we need an additional vector array $t[0 : r - 1]$ to support the merge. CBE merge sort adaptation is presented as a stable sort and later we point out the simplifications that are possible when we wish to sort numbers rather than elements that have multiple fields. We again assume that the numbers are in the vector array $d[0 : r - 1]$.

There are 4 phases to our stable merge sort adaptation:

**Phase 1:** Transpose the elements of $d[0 : r - 1]$, which represents a $r \times 4$ matrix, from row-major to column-major order.

**Phase 2:** Sort the 4 columns of the $r \times 4$ matrix independently and in parallel.

**Phase 3:** In parallel, merge the first 2 columns together and the last 2 columns together to get two sorted sequences of length $2r$ each.

**Phase 4:** Merge the two sorted sequences of length $2r$ each into a row-major sorted sequence of length $4r$.

**Merge sort phase 1–transpose:** We note that Phase 1 is needed only when we desire a stable sort. Figure 2-8 shows an initial $8 \times 4$ matrix of numbers and the result following each of the 4 phases of our merge sort adaptation.

The Phase 1 transformation is the inverse of the column-major to row-major transformation done in Figure 2-5 and we do not provide its details. Details for the remaining 3 phases are provided in the following subsections.

**Merge sort phase 2–sort columns:** Phase 2 operates in $\log r$ subphases characterized by the size of the sorted segments being merged. For instance, in the first subphase, we merge together pairs of sorted segments of size 1 each, in the next subphase the segment size is 2, in the third it is 4, and so forth. At any time, the two segments being merged have the same physical locations in all 4 columns. So, for our $8 \times 4$ example, when merging together segments of size 2, we shall first merge, in

parallel, 4 pairs of segments, one pair from each column. The first segment of a pair is in rows 0 and 1 of the $r \times 4$ matrix and the second in rows 2 and 3. Then, we shall merge together the segments of size 2 that are in rows 4 through 7. Following this, the segment size becomes 4.

To merge 4 pairs of segments in parallel, we employ 8 counters to keep track of where we are in the 8 segments being merged. The counters are called $a_0, \ldots, a_3, b_0, \ldots, b_3$. $(a_i, b_i)$ are the counters for the segments of column $i$, $0 \leq i \leq 3$ that are being merged. When the segment size is $s$ and the segments being merged occupy rows $i$ through $i + 2s - 1$, the $a$ counters are initialized to $i$ and the $b$ counters to $i + s$. Although all $a$ counters have the same value initially as do all $b$ counters, as merging progresses, these counters have different values. Figure 2-9 gives the Phase 2 algorithm. For simplicity, we assume that $r$ is a power of 2.

Merge sort phase 3—merge pairs of columns: In Phase 3 we merge the first two and last two columns of the $r \times 4$ matrix together to obtain 2 sorted sequences, each of
Algorithm Phase2(d, r)

```plaintext
{// sort the 4 columns of d[0:r-1], use additional array t[0:r-1]
for (s = 1; s < r; s *= 2)
    for (i = 0; i < r; i += 2*s) {
        A = spu_splats(i); // initialize a counters
        B = spu_splats(i+s); // initialize b counters
        for (k = i; k < i + 2*s; k++) {// merge the segments
            // one round of compares
            aData = mySpu_gather(d,A);
            bData = mySpu_gather(d,B);
            p = spu_cmpgt(aData,bData);
            t[k] = spu_select(aData,bData,p);
            // update counters
            notP = mySpu_not(p);
            A = spu_sub(A,notP);
            B = spu_sub(B,p);
        }
    }
swap(d,t); // swap roles
}
```

Figure 2-9. Phase 2 of merge sort

size $2r$. The first sequence is in columns 0 and 1 and the second in columns 2 and 3 of an output matrix. We do this merging by using 8 counters. Counters $a_0, b_0, a_1, b_1$ start at the top of the 4 columns of our matrix and move downwards while counters $a_2, b_2, a_3, b_3$ start at the bottom and move up (see Figure 2-10(a)). Let $e(c)$ be the matrix element that counter $c$ is at. The comparisons $e(a_i) : e(b_i), 0 \leq i \leq 3$ are done in parallel and depending on the outcome of these comparisons, 4 of the 8 elements compared are moved to the output matrix. When $e(a_0) \leq e(b_0) (e(a_1) \leq e(b_1)), e(a_0) (e(a_1))$ is moved to the output and $a_0 (a_1)$ incremented by 1; otherwise, $e(b_0) (e(b_1))$ is moved to the output and $b_0 (b_1)$ incremented by 1. Similarly, when $e(a_2) \leq e(b_2) (e(a_3) \leq e(b_3)), e(b_2) (e(b_3))$ is moved to the output and $b_2 (b_3)$ decremented by 1; otherwise, $e(a_2) (e(a_3))$ is moved to the output and $a_2 (a_3)$ decremented by 1. The merge is complete when we have done $r$ rounds of comparisons. Figure 2-11 gives the algorithm for Phase 3.

**Theorem 2.1.** Algorithm Phase3 correctly merges 4 sorted columns into 2.

**Proof.** To prove the correctness of Algorithm Phase3 we need to show that each element of the first (last) two columns of the input $r \times 4$ matrix is copied into the first (last) two
columns of the output matrix exactly once and that the elements of the first (third) output column followed by those of the second (fourth) are in sorted order. It is sufficient to show this for the first two columns of the input and output matrices. First, observe that when \( a_0 \leq a_2 \) (\( b_0 \leq b_2 \)), these counters are at input elements that have yet to be copied to the output. Further, when \( a_0 > a_2 \) (\( b_0 > b_2 \)) all elements of the respective column have been copied from the input to the output (note that a counter is updated only when its
Algorithm Phase3(d,r)
{
    // merge the 4 sorted columns of d[0:r-1] into 2 sorted sequences
    // use additional array t[0:r-1]
    A = {0, 0, r-1, r-1}; // initialize a counters
    B = {0, 0, r-1, r-1}; // initialize b counters
    for (k = 0; k < r; i++) {
       aData = mySpu_gatherA(d,A);
        bData = mySpu_gatherB(d,B);
        p = spu_cmpgt(aData,bData);
        e = spu_equal(aData,bData);
        e = spu_and(e, vector(0,0,-1,-1));
        p = spu_or(p, e);
        min = spu_select(aData,bData, p);
        max = spu_select(bData,aData, p);
        t[k] = spu_shuffle(min,t[k],WBXD);
        t[r-k-1] = spu_shuffle(max,t[r-k-1],AYCZ);
    } // update counters
    notP = mySpu_not(p);
    f1 = spu_and(p,vector(-1,-1,0,0));
    s1 = spu_and(p,vector(0,0,-1,-1));
    f2 = spu_and(notP,vector(-1,-1,0,0));
    s2 = spu_and(notP,vector(0,0,-1,-1));
    A = spu_sub(A,f2);
    A = spu_add(A,s2);
    B = spu_sub(B,f1);
    B = spu_add(B,s1);
}

Figure 2-11. Phase 3 of merge sort

element has been copied to the output matrix). We consider 4 cases: \( a_0 < a_2 \), \( a_0 = a_2 \), \( a_0 = a_2 + 1 \), and \( a_0 > a_2 + 1 \).

Case \( a_0 < a_2 \): When \( b_0 < b_2 \) (Figure 2-10A), exactly one of \( e(a_0) \) and \( e(b_0) \) and one of \( e(a_2) \) and \( e(b_2) \) are copied to the output and the corresponding counters are advanced. No element is copied to the output twice.

Next, consider the case \( b_0 = b_2 \) (Figure 2-10B). If \( e(a_0) \leq e(b_0) \), \( e(a_0) \) and one of \( e(a_2) \) and \( e(b_2) \) are copied to the output and the corresponding counters advanced. Again no element is copied to the output twice. If \( e(a_0) > e(b_0) = e(b_2) \), \( e(b_2) < e(a_0) \leq e(a_2) \) and \( e(b_0) \) and \( e(a_2) \) are copied to the output and their counters advanced. Again, no element is copied twice.
The next case we consider has \( b_0 = b_2 + 1 \). Let the values of \( b_0 \) and \( b_2 \) be \( b'_0 \) and \( b'_2 \) just before the update(s) that resulted in \( b_0 = b_2 + 1 \) and let \( a'_0 \) and \( a'_2 \) be the values of the \( a \) counters at this time. One of the following must be true: (a) \( b'_2 = b'_0 + 1 \) (both \( b_0 \) and \( b_2 \) were advanced, Figure 2-10C), (b) \( b'_0 = b'_2 = b_0 \) (only \( b_2 \) was advanced, Figure 2-10D), or (c) \( b'_0 = b'_2 = b_2 \) (only \( b_0 \) was advanced, Figure 2-10E). In (a), it must be that \( b_2 = b'_0 \) and \( b_0 = b'_2 \). So, \( e(a_0) > e(b'_0) \) and \( e(a_2) \leq e(b'_2) \). Hence, \( e(a_0) \leq e(a_2) \leq e(b'_2) = e(b_0) \) and \( e(a_2) \geq e(a_0) > e(b'_0) = e(b_2) \). Therefore, \( e(a_0) \) and \( e(a_2) \) are copied to the output and \( a_0 \) and \( a_2 \) advanced. Again, only previously uncopied elements are copied to the output and each is copied once. For subcase (b), when \( b'_2 \) was decremented to \( b_2 \), \( a'_0 \) was incremented to \( a_0 \), \( e(b'_2) \geq e(a_2) \) and \( e(a'_0) \leq a(b'_0) \).

Since \( b_0 > b_2 \), all elements of the second column have been copied to the output. We see that \( e(a_0) \leq e(a_2) \leq e(b'_2) = e(b_0) \). So, \( e(a_0) \) is copied and \( a_0 \) is advanced. Further, as a result of some previous comparison, \( b_0 \) was advanced to its current position from the present position of \( b_2 \). So, there is an \( a''_0 \leq a_0 \) such that \( e(b_0) < e(a''_0) \leq e(a_0) \leq e(a_2) \). Therefore, \( e(a_2) \) is copied and \( a_2 \) advanced. Again, no previously copied element is copied to the output and no element is copied twice. Subcase (c) is symmetric to subcase (b).

The final case has \( b_0 > b_2 + 1 \) (Figure 2-10F). From the proof of subcases \( b_0 = b_2 \) and \( b_0 = b_2 + 1 \), it follows that this case cannot arise.

**Case** \( a_0 = a_2 \): There are 4 subcases to consider—(a) \( b_0 < b_2 \), (b) \( b_0 = b_2 \), (c) \( b_0 = b_2 + 1 \), and (d) \( b_0 > b_2 + 1 \) (Figures 2-10(G–J)). Subcase (a) is symmetric to the case \( a_0 < a_2 \) and \( b_0 = b_2 \) considered earlier. In subcase (b), independent of the outcome of the comparison \( e(a_0) : e(b_0) \), which is the same as the comparison \( e(a_2) : e(b_2) \), \( e(a_0) \) (equivalently \( e(a_2) \)) and \( e(b_0) \) (equivalently \( e(b_2) \)) are copied to the output. For subcase (c), we notice that when \( a_0 = a_2 \), these two counters have had a cumulative advance of \( r - 1 \) from their initial values and when \( b_0 = b_2 + 1 \) these two
counters have together advanced by \( r \). So, the 4 counters together have advanced by \( 2r - 1 \) from their initial values. This isn’t possible as the 4 counters advance by a total of 2 in each iteration of the for loop. So, subcase (c) cannot arise. Next, consider subcase (d). From the proof for the case \( a_0 < a_2 \), we know that we cannot have \( b_0 > b_2 + 1 \) while \( a_0 < a_2 \). So, we must have got into this state from a state in which \( a_0 = a_2 \) and \( b_0 \leq b_2 \). It isn’t possible to get into this state from subcase (a) as subcase (a), at worst increases \( b_0 \) by 1 and decreases \( b_2 \) by 1 each time we are in this subcase. So, it is possible to get into this subcase only from subcase (b). However, subcase (b) only arises at the last iteration of the for loop. Even otherwise, subcase (b) either increments \( b_0 \) by 1 or decrements \( b_2 \) by 1 and so cannot result in \( b_0 > b_2 + 1 \).

**Case** \( a_0 > a_2 + 1 \): From the proofs of the remaining cases, it follows that this case cannot arise.

From the proof of Theorem 2.1, it follows that when we are sorting numbers rather than records with numeric keys, algorithm Phase3 works correctly even with the statements

\[
\begin{align*}
e &= \text{spu\_equal}(aData, bData); \\
e &= \text{spu\_and}(e, \text{vector}(0,0,-1,-1)); \\
p &= \text{spu\_or}(p, e);
\end{align*}
\]

omitted.

**Merge sort phase 4–final merge:** For the Phase 4 merge, we employ 4 counters. Counters \( a_0 \) and \( a_1 \), respectively begin at the first and last element of the first sorted sequence (i.e., at the top of the first column and bottom of the second column, respectively) while \( b_0 \) and \( b_1 \) begin at the first and last elements of the second sequence (Figure 2-12). In each round, the comparisons \( a_0 : b_0 \) and \( a_1 : b_1 \) are done in parallel. \( e(a_0) \) (\( e(b_1) \)) is moved to the output if \( e(a_0) \leq e(b_0) \) (\( e(b_1) \geq e(a_1) \)). Otherwise, \( e(b_0) \)
$(e(a_1))$ is moved to the output. The sorted output is assembled in row-major order in the vector array $t$. We use the variables $k$ and $pos$ to keep track of the row and column in $t$ in which to place the output element from the comparison $e(a_0) : e(b_0)$. The output element from $e(a_1) : e(b_1)$ goes into row $(r - k - 1)$ and column $(3 - pos)$ of $t$.

Figure 2-13 gives the algorithm for the case when the counters remain within the bounds of their respective columns. $mask[pos], 0 \leq pos \leq 3$ is defined so as to change only the number in position $pos$ of a $t[]$ vector.

\[
\begin{array}{c|c|c|c}
\text{Column} & 0 & 1 & 2 & 3 \\
\hline
a_0 & & & & \\
\hline
b_0 & & & & \\
\hline
a_1 & & & & \\
\hline
b_1 & & & & \\
\end{array}
\]

Figure 2-12. Phase 4 counters

As was the case in Phase 3, the statements
\[
e = \text{spu\_equal(aData,bData)};
\]
\[
e = \text{spu\_and(e, vector(0,0,-1,-1))};
\]
\[
p = \text{spu\_or(p, e)};
\]
may be omitted when we are sorting numbers rather than records with numeric keys.

### 2.3.1.3 Comparison of single SPU sorting algorithms

We programmed our merge sort, brick sort, and shaker sort adaptations using the CBE SDK Version 3.0. For comparison purposes, we used an AA sort code developed by us, the Cell sort code of [8], a non-vectorized merge sort code developed by us, and the quick sort routine available in the CBE SDK. The codes were first debugged and optimized using the CBE simulator that is available with the SDK. The optimized codes were run on the Georgia Tech-STI Cellbuzz cluster to obtain actual run times. Table 2-1 gives the average time required to sort $n$ 4-byte integers for various values of $n$. The average for each $n$ is taken over 5 randomly generated sequences. The variance in the
Algorithm Phase4(d,r)
{// partial algorithm to merge 2 sorted sequences of d[0:r-1]
// into 1 sorted sequence
// use additional array t[0:r-1]
A = {0, r-1, 0, 0}; // initialize a counters
B = {0, r-1, 0, 0}; // initialize b counters
k = 0; pos = 0;
while (no column is exhausted) {
  aData = mySpu.gather12(d,A);
  bData = mySpu.gather34(d,B);
  p = spu_cmpgt(aData,bData);
  e = spu_equal(aData,bData);
  e = spu_and(e, vector(0,-1,0,0));
  p = spu_or(p, e);
  min = spu_select(aData, bData, p);
  max = spu_select(bData, aData, p);
  max = spu_slqubyter(max, 4);
  t[k] = spu_shuffle(min, t[k], mask[pos]);
  t[r-k-1] = spu_shuffle(max, t[r-k-1], mask[3-pos]);
  // update counters
  notP = mySpu_not(p);
  f1 = spu_and(p, vector(-1,0,0,0));
  s1 = spu_and(p, vector(0,-1,0,0));
  f2 = spu_and(notP, vector(-1,0,0,0));
  s2 = spu_and(notP, vector(0,-1,0,0));
  A = spu_sub(A,f2);
  A = spu_add(A,s1);
  B = spu_sub(B,f1);
  B = spu_add(B,s2);
  k += (pos+1)/4;
  pos = (pos+1)%4;
}
}

Figure 2-13. Phase 4 of merge sort with counters in different columns

sort time from one sequence to the next is rather small and so the reported average is
not much affected by taking the average of a larger number of random input sequences.
Figure 2-14 is a plot of the average times reported in Figure 2-1. The shown run times
include the time required to fetch the data to be sorted from main memory and to store
the sorted results back to main memory.

2.3.2 Hierarchical Sort

For sorting a large set of numbers, a hierarchical approach similar to external
sort is employed where first each SPU sorts a local memory load of data to generate
sorted sequences called runs. These runs are then merged by the SPUs to produce
Table 2-1. Comparison of various SPU sorting algorithms

<table>
<thead>
<tr>
<th># of Integers</th>
<th>AASort</th>
<th>Shaker sort</th>
<th>Brick Sort</th>
<th>Bitonic Sort</th>
<th>Merge Sort</th>
<th>Merge Sort (Sequential)</th>
<th>Quick Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>52.0</td>
<td>53.6</td>
<td>53.0</td>
<td>47.8</td>
<td>50.8</td>
<td>146.6</td>
<td>145.6</td>
</tr>
<tr>
<td>256</td>
<td>62.4</td>
<td>65.4</td>
<td>63.4</td>
<td>65.6</td>
<td>57.6</td>
<td>178.6</td>
<td>206.8</td>
</tr>
<tr>
<td>512</td>
<td>81.8</td>
<td>86.4</td>
<td>81.4</td>
<td>72.6</td>
<td>70.4</td>
<td>272.2</td>
<td>332.0</td>
</tr>
<tr>
<td>1024</td>
<td>123.8</td>
<td>142.2</td>
<td>116.8</td>
<td>125.4</td>
<td>97.0</td>
<td>315.4</td>
<td>605.6</td>
</tr>
<tr>
<td>2048</td>
<td>222.8</td>
<td>262.0</td>
<td>190.2</td>
<td>165.8</td>
<td>142.0</td>
<td>543.0</td>
<td>1164.0</td>
</tr>
<tr>
<td>4096</td>
<td>438.6</td>
<td>494.8</td>
<td>332.6</td>
<td>297.8</td>
<td>268.4</td>
<td>989.8</td>
<td>2416.6</td>
</tr>
<tr>
<td>8192</td>
<td>912.4</td>
<td>1033.6</td>
<td>663.8</td>
<td>609.6</td>
<td>508.0</td>
<td>2011.2</td>
<td>4686.6</td>
</tr>
<tr>
<td>16384</td>
<td>1906.4</td>
<td>2228.0</td>
<td>1361.0</td>
<td>1331.2</td>
<td>1017.0</td>
<td>4103.0</td>
<td>9485.2</td>
</tr>
</tbody>
</table>

the final sorted sequence. Depending on how many runs are merged at a time there will be multiple rounds of merging before generating the final sorted sequence. The PPU dispatches the runs to the SPUs which do the merging and return the results back to the PPU. In the run merging phase, each SPU independently merges a different set of runs. So, one needs to develop only a run merging algorithm for a single SPU. Inoue et al. [11] propose a single SPU merging algorithm that merges runs in pairs (i.e., a 2-way merge) using an adaptation of odd-even merge. Odd-even merge of two four-number sorted sequences is done in 3 stages. First, the two sorted sequences are concatenated to get an 8-number sequence where each half is in sorted order. During the first stage, numbers that are 4 positions apart are compare-exchanged \(^1\). In the second stage, numbers that are 2 positions apart are compare-exchanged and in the last stage alternate numbers are compare-exchanged if needed. This scheme

\(^1\) In a compare-exchange, two numbers are compared and swapped if the first number is larger than the second
can be effectively SIMDized by beginning with two vectors, each containing one of the two sorted 4-number sequences. Vector compare instructions are used so that 4 compare-exchange may be done at a time. Figure 2-15 shows the process with two sorted vectors $A$ and $B$ and Figure 2-16 gives the pseudocode for this adaptation.

As the runs are too long to fit in the local memory of an SPU, buffers are used to hold a portion of each of the runs currently being merged. Multi-buffering techniques are employed to overlap the computation with the data transfer. Figure 2-17 is the plot of the average times taken for sorting 1 to 67M integers with different SPU sorting algorithms followed by SIMD odd-even 2-way merge to merge the runs except in the case of Bitonic Sort [8] where bitonic merge is used to merge the runs and in the case of sequential merge sort where a textbook merge is done to combine the runs. Similar to single SPU sorting, Brick sort is the fastest among shell sort like algorithms taking 82% of the time taken by AA Sort for sorting 67M numbers while Shaker sort is the slowest of the bunch running 21% slower than Brick sort for sorting 67M numbers. Merge sort is the fastest of
Figure 2-15. SIMD odd-even merge of two vectors \[11\]

all algorithms tested taking 84% of the time taken by Brick sort and runs 24% faster than Bitonic sort for sorting 67M numbers. Compared to sequential sorts, the SIMD version of merge sort runs 3 times faster than the textbook merge sort and 24 times faster than the SDK quick sort.

### 2.4 Sorting Records

#### 2.4.1 Record Layouts

A record \( R \) is comprised of a key \( k \) and \( m \) other fields \( f_1, f_2, \ldots, f_m \). For simplicity, we assume that the key and each other field occupies 32 bits. Hence, a 128-bit CBE vector may hold up to 4 of these 32-bit values. Although the development in this section relies heavily on storing 4 keys in a vector (each key occupying 32 bits), the size of the other fields isn’t very significant. Let \( k_i \) be the key of record \( R_i \) and let \( f_{ij}, 1 \leq j \leq m \) be this record’s other fields. With our simplifying assumption of uniform size fields, we may view the \( n \) records to be sorted as a two-dimensional array \( \text{fieldsArray}[][] \) with \( \text{fieldsArray}[i][0] = k_i \) and \( \text{fieldsArray}[i][j] = f_{ij}, 1 \leq j \leq m, 1 \leq i \leq n \). When this array
Algorithm oddEvenMerge(v1, v2)
{// Merge two vectors v1 and v2
   vector f1, f2;
   vector f3, f4;
   vector p;  // for storing pattern
   p = spu_cmpgt(v1, v2);
   f1 = spu_select(v1, v2, pattern);
   f2 = spu_select(v2, v1, pattern);
   // Stage 2
   f3 = spu_rotate(f1, 8);
   f1 = spu_select(f3, f2, p);
   f4 = spu_select(f2, f3, p);
   f2 = spu_shuffle(f1, f4, WACY);
   f3 = spu_shuffle(f1, f4, ZXBD);
   // Stage 3
   p = spu_cmpgt(f2, f3);
   p = spu_shuffle(p, vZero, WXYA);
   f1 = spu_select(f2, f3, p);
   f4 = spu_select(f3, f2, p);
   // Output
   v1 = spu_shuffle(f1, f4 ZWAX);
   v2 = spu_shuffle(f1, f4, BYCD);
}

Figure 2-16. SIMD 2-way merge of 2 vectors $v_1$ and $v_2$

is mapped to memory in column-major order, we get the first layout considered in [11].
We call this layout the ByField layout as, in this layout, the $n$ keys come first. Next we
have the $n$ values for the first field of the records followed by the $n$ second fields, and so
on. When the fields array is mapped to memory in row-major order, we get the second
layout considered in [11]. This layout, which is a more common layout for records, is
called the ByRecord layout as, in this layout, all the fields of $R_1$ come first, then we have
all fields of $R_2$ and so on. When the sort begins with data in the ByField (ByRecord)
layout, the result of the sort must also be in the ByField (ByRecord) layout.

2.4.2 High-level Strategies for Sorting Records

There are two high-level strategies to sort multifield records. In the first, we strip
the keys from the records and create $n$ tuples of the form $(k_i, i)$. We then sort the tuples
by their first component. The second component of the tuples in the sorted sequence
defines a permutation of the record indexes that corresponds to the sorted order for the
initial records. The records are rearranged into this permutation by either copying from
Figure 2-17. Plot of average time to sort 1 to 67 million integers

fieldsArray to a new space or inplace using a cycle chasing algorithm as described for a table sort in [20]. This strategy has the advantage of requiring only a linear number of record moves. So, if the size of each record is \( s \) and if the time to sort the tuples is \( O(n \log n) \), the entire sort of the \( n \) records can be completed in \( O(n \log n + ns) \) time.

The second high-level strategy is to move all the fields of a record each time its key is moved by the sort algorithm. In this case, if the time to sort the keys alone is \( O(n \log n) \), the time to sort the records is \( O(ns \log n) \). For relatively small \( s \), the first strategy outperforms the second when the records are stored in uniform access memory.

However, since reordering records according to a prescribed permutation with a linear number of moves makes random accesses to memory, the second scheme outperforms the first (unless \( s \) is very large) when the records to be rearranged are in relatively slow memory such as disk or the main memory of the CBE. For this reason, we focus, in this
chapter, on using the second strategy. That is, the sort algorithm moves all the fields of a record whenever its key is moved.

2.4.3 Single SPU Record Sorting

Two SIMD vector operations used frequently in number sorting algorithms are findmin and shuffle. The findmin operation compares corresponding elements in two vectors and returns a vector min that contains, for each compared pair, the smaller. For example, when the two vectors being compared are (4, 6, 2, 9) and (1, 8, 5, 3), the min is (1, 6, 2, 3). Suppose that \( v_i \) and \( v_j \) are vectors that, respectively, contain the keys for records \( R_{i:i+3} \) and \( R_{j:j+3} \). Figure 2-18 shows how we may move the records with the smaller keys to a block of memory beginning at \( \text{minRecords} \).

\[
\text{pattern} = \text{spu_cmpgt}(v_i, v_j);
\]
\[
\text{minRecords} = \text{fields_select}(v_i, v_j, \text{pattern});
\]

Figure 2-18. The findmin operation for records

When the ByField layout is used, fields_select takes the form given in Figure 2-19.

\[
\text{for}(p = 1; p <= m; p++) \{
\quad \text{minRecords}[p] = \text{spu_select}(\text{fieldsArray}[i][p], \text{fieldsArray}[j][p], \text{pattern});
\}
\]

Figure 2-19. The fields_select operation in ByField layout

Notice that in the ByField layout, the elements \( \text{fieldsArray}[i : i + 3][p] \) are contiguous and define a 4-element vector. However, in the ByRecord layout, these elements are not contiguous and a different strategy (Figure 2-20) must be employed. The function \( \text{memcpy}(\text{dest, src, size}) \) moves \( \text{size} \) number of bytes from the local store location beginning at \( \text{src} \) to the local store location beginning at \( \text{dest} \); recSize is the length of a record including its key (i.e., it is the number of bytes taken by a row of \( \text{fieldsArray} \)).

The shuffle operation defined by \( \text{spu_shuffle} \) for the case of sorting numbers may be extended to the case of multifield records using the code of Figure 2-21 for the ByField layout and that of Figure 2-22 for the ByRecord layout. Both codes are for the case
for(p = 0; p < 4; p++) {
    memcpy(minRecords + p * recSize, (pattern[p] == 1) ? fieldsArray[i+p] : fieldsArray[j+p], recSize);
}

Figure 2-20. The *fields_select* operation in *ByRecord* layout

when the shuffle pattern is *WYAC*. Other shuffle patterns are done in a similar way. This extension of *spu_shuffle* to records is referred as *fields_shuffle*. Other vector operations like *spu_slqwbyte* can be thought of as a *spu_shuffle* operation with a certain pattern and one can define a similar operation *fields_rotate* along those lines for the records in both layouts e.g. *fields_rotate(v, 8)* is equivalent to *fields_shuffle(v, v, CDWX)*.

for(p = 0; p <= m; p++) {
    resultRecords[p] = spu_shuffle(fieldsArray[i][p], fieldsArray[j][p], WYAC);
}

Figure 2-21. Shuffling two records in *ByField* layout

memcpy(resultRecords, fieldsArray[i], recSize);
memcpy(resultRecords + recSize, fieldsArray[i + 2], recSize);
memcpy(resultRecords + 2 * recSize, fieldsArray[j], recSize);
memcpy(resultRecords + 3 * recSize, fieldsArray[j + 2], recSize);

Figure 2-22. Shuffling two records in *ByRecord* layout

We observe that when the *ByField* layout is used the *findMin* and *shuffle* operations perform $O(m)$ vector operations and that when the *ByRecord* layout is used, a constant number (4) of *memcpy* operations are done. However, the time for each *memcpy* operation increases with *m*.

2.4.4 Hierarchical Sorting for Records - 4way Merge

In Section 2.4, we saw the adaptations needed to the run generation algorithms of [8, 11, 12] so that these may be used to generate runs for multifield records rather than for numbers. In the run merging phase, each of the SPUs independently merges a different set of runs. Inoue et al. [11] propose a single SPU merging algorithm that merges runs in pairs (i.e., a 2-way merge) using an adaptation of odd-even merge. It takes 3 *spu_cmpgt* instructions, 6 *spu_shuffle* and 6 *spu_select* instructions to merge two
SPU vectors using this scheme. The run merging strategy of [11] may be adapted to the case of multifield records using the methods of Section 2.4. A higher-order merge in the run merging phase reduces IO time and has little impact on computation time [20]. Moreover, when sorting multifield records, the IO time increases with the size of a record and for suitably large records, this IO time will exceed the computation time and so cannot be effectively hidden using a 2-way merge and double buffering. So, for multifield records, there is merit to developing a higher-order merge. Correspondingly, two 4-way merge algorithms are proposed in [21]. One is a scalar algorithm and the other is a vectorized SIMD algorithm. Both algorithms are based on the high-level strategy shown in Figure 2-23.

![4-way merge diagram](image)

Figure 2-23. 4-way merge

The 4-way merge strategy involves performing 3 2-way merges in a single SPU using two buffers (main and alt) for each of the 4 input streams A, B, C, and D as well as 2 buffers for the output stream O. An additional buffer is used for the output (E and F, respectively) of each of the two left 2-way merge nodes of Figure 2-23. So, we employ a total of 12 buffers. Runs A and B are pairwise merged using the top left 2-way merge node while runs C and D are pairwise merged using the bottom left 2-way merge node. The former 2-way merge generates the intermediate run E while the latter generates the intermediate run F. The intermediate runs E and F are merged by the right 2-way merge node to produce the output run O, which is written back to main memory. Run generation is done one block or buffer load at a time. Double buffering is employed for
the input of A, B, C, and D from main memory and the output of O to main memory. By using double buffering and asynchronous DMA transfers to and from main memory, we are able to overlap much of the IO time with computation time.

**Scalar 4-way merge:** Figure 2-24 gives the pseudocode for the scalar 4-way merge algorithm. For simplicity, algorithm 4wayPipelinedMerge assumes that we have an integral number of blocks of data in each run. So, if each of the runs A, B, C, and D is (say) 10 blocks long, the output run O will be \( n = 40 \) blocks long. 4wayPipelinedMerge generates these output \( n \) blocks one block at a time. Even blocks are accumulated in one of the output buffers and odd blocks in the other. When an output buffer becomes full, we write the block to memory using an asynchronous DMA transfer (dmaOut) and continue output run generation using the other output buffer. So, other than when the first output block is being generated and the last being written to main memory, one of the output blocks is being written to main memory while the other one is being filled with records for the next block. At the end of each iteration of the outer for loop, we switch the roles of the two output buffers—the one that was being written to main memory becomes the buffer to place records for the next block and the one that was being filled is written out. Of course, this switch may entail some delay as we must wait for the ongoing (if any) dmaOut to complete before we use this buffer for the records of the next block. When generating a block of the output run, we merge from the buffers bufferE and bufferF to the output buffer bufferO that is currently designated for this purpose. The number of records in a full buffer (i.e., the block size) is bSize. In case either bufferE or bufferF is empty, the generation of the output block is suspended and we proceed to fill the empty buffer using the method mergeEF, which merges from either input streams A and B to bufferE or from streams C and D to bufferF. The algorithm mergeEF merges for either the input streams A and B to bufferE or from E and F to bufferF. It uses double buffering on the streams A, B, C, and D and ensures that there is always an active dmaIn for these four input streams. Since the pseudocode is similar to that
for 4wayPipelinedMerge, we do not provide this pseudocode here. Records are moved between buffers using the memcpy instructions when the ByRecord layout is used and moved one field at a time when the layout is ByField.

Algorithm 4wayPipelinedMerge(A, B, C, D, O, n)
{ // Merge runs/streams A, B, C, and D to produce O
  // with n blocks of size bSize
  // bufferA is a buffer for A
  initiate a dmaIn for bufferA, bufferB, bufferC and bufferD;
  for (i = 0; i < n; i++) {
    for (j = 0; j < bSize; j++) { // do block i
      if(bufferE is empty)
        mergeEF(A, B, E);
      if(bufferF is empty)
        mergeEF(C, D, F);
      move smaller record from front of bufferE
      and bufferF to bufferO
    }
    dmaOut(buffer0, O);
    switch the roles of the output buffers;
  }
}

Figure 2-24. 4-way merge

**SIMD 4-way merge:** The SIMD version differs from the scalar version only in the way each of the three 2-way merges comprising a 4-way merge works. These 2-way merges move 4 records at a time from input buffers to the output buffer using odd-even merge scheme on the keys of those records. Two sorted vectors each consisting of keys from 4 sorted records are merged using odd-even merge. The fields are also moved correspondingly using the fields operations introduced in the previous sections. The odd-even merge of two vectors is essentially the same process as in case of merging numbers described in Section 2.3.2. Figure 2-25 gives the pseudocode of the adaptation for merging records.

In Algorithm oddEvenMerge, \( v_1 \) and \( v_2 \) are two vectors each containing the keys of the next 4 records in the input buffers for the two streams being merged. It is easy to see that the next four records in the merged output are a subset of these 8 records and

48
Algorithm oddEvenMerge(v1, v2)
{// Merge records whose keys are in v1 and v2
    fields f1[], f2[];
    fields f3[], f4[];
    vector p; // for storing pattern
    p = spu_cmpgt(v1, v2);
    f1 = fields_select(v1, v2, pattern);
    f2 = fields_select(v2, v1, pattern);
    // Stage 2
    f3 = fields_rotate(f1, 8);
    f1 = fields_select(f3, f2, p);
    f4 = fields_select(f2, f3, p);
    f2 = fields_shuffle(f1, f4, WACY);
    f3 = fields_shuffle(f1, f4, ZXBD);
    // Stage 3
    p = spu_cmpgt(f2, f3);
    p = spu_shuffle(p, vZero, WXYA);
    f1 = fields_select(f2, f3, p);
    f4 = fields_select(f3, f2, p);
    // Output
    v1 = fields_shuffle(f1, f4 ZWAX);
    v2 = fields_shuffle(f1, f4, BYCD);
}

Figure 2-25. SIMD 2-way merge of 2 vectors v1 and v2

in fact are the 4 records (of these 8) with the smallest keys. Algorithm oddEvenMerge
determines these 4 smallest records and moves these to the output buffer.

2.4.5 Comparison of Record Sorting Algorithms

We programmed several multifield record sorting algorithms using Cell BE SDK 3.1. Specifically, the following algorithms were implemented and evaluated:

1. 2-way AA Sort ... this is the multifield record sorting algorithm of Inoue et al. [11].
    This uses a comb sort variant for run generation and 2-way odd-even merge for
    run merging.

2. 4-way AA Sort ... this uses a comb sort variant for run generation as in [11] and
    our 4-way odd-even merge for run merging (Section 2.4.4).

3. 2-way Bitonic Sort ... this is an adaptation of the CellSort algorithm of Gedik et
    al. [8] to multifield records (Section 2.4). It uses bitonic sort for run generation and
    2-way bitonic merge for run merging.

4. 4-way Bitonic Sort ... this uses bitonic sort for run generation as in [8] and our
    4-way odd-even merge for run merging (Section 2.4.4).
5. 2-way Merge Sort ... this uses an adaptation of the SPU merge sort algorithm of Bandyopadhyay and Sahni [12] to multifield records (Section 2.4) for run generation and the 2-way odd-even merge of [11] for run merging.

6. 4-way Merge Sort ... this uses an adaptation of the SPU merge sort algorithm of Bandyopadhyay and Sahni [12] to multifield records (Section 2.4) for run generation and our 4-way odd-even merge for run merging (Section 2.4.4).

7. 2-way Scalar Merge Sort ... this uses an adaptation of the SPU merge sort algorithm of Bandyopadhyay and Sahni [12] to multifield records (Section 2.4) for run generation. Run merging is done using a 2-way scalar merging algorithm derived from the 4-way scalar merging algorithm of Section 2.4.4 by eliminating the bottom left and the right 2-way merge nodes.

8. 4-way Scalar Merge Sort ... this uses an adaptation of the SPU merge sort algorithm of Bandyopadhyay and Sahni [12] to multifield records (Section 2.4) for run generation and our 4-way scalar merge for run merging (Section 2.4.4).

We experimented with the above 8 multifield sorting algorithms using randomly generated input sequences. In our experiments, the number of 32-bit fields per record is varied from 5 to 15 (in addition to the key field) and the number of records varied from 4K to 1M. Also, we tried both layouts—ByField and ByRecord. For each combination of number of fields, number of records, and layout type, the time to sort 10 random sequences was obtained. The standard deviation in the observed run times was small and we report only the average times.

2.4.5.1 Run times for ByField layout

Figures 2-26A through 2-27D give the average run times for our 8 sorting algorithms using the ByField layout and Figures 2-28A through 2-28D compare the average run times for the 2-way and 4-way versions of each of our sorting algorithms for the case when the number of records to be sorted is 1M. For all our data, the 4-way version outperformed the 2-way version. For 1M records with 5 32-bit fields (in addition to a 32-bit key), the 4-way versions of AA Sort, Bitonic Sort, Merge Sort, and Scalar Merge Sort, respectively, took 5%, 4%, 7%, and 4% less time than taken by their 2-way counterparts and these percentages for 15 fields were 9%, 6%, 9%, and 6% respectively.
Figure 2-26. 2-way sorts (ByField). A) 2-way AA sort (ByField). B) 2-way bitonic sort (ByField). C) 2-way merge sort (ByField). D) 2-way scalar merge sort (ByField).

Figure 2-29 shows the run times for the 4 4-way sort algorithms for 1M records. As can be seen, 4-way Bitonic Sort is the slowest, followed by 4-way Scalar Merge Sort, followed by 4-way AA Sort; 4-way Merge Sort was the fastest. In fact, across all our data sets, 4-way Bitonic Sort took between 17% and 23% more time than taken by 4-way Scalar Merge Sort, which in turn took between 18% and 19% more time than taken by 4-way AA Sort. The fastest 4-way sort algorithm, 4-way Merge Sort took, respectively, between 40% and 35%, 26% and 25%, 13% and 10% less time than taken by 4-way Bitonic Sort, 4-way Scalar Merge Sort, and 4-way AA Sort.
Figure 2-27. 4-way sorts (ByField). A) 4-way AA sort (ByField). B) 4-way bitonic sort (ByField). C) 4-way merge sort (ByField). D) 4-way scalar merge sort (ByField).

2.4.5.2 Run times for ByRecord layout

Figures 2-30A through 2-31D give the average run times for sorting algorithms using the ByRecord layout and Figures 2-32A through 2-32D present the comparison of average run times for the 2-way and 4-way versions of each sorting algorithm when the number of records to be sorted is 1M. In this layout as well, the 4-way version outperformed the 2-way version for all the data sets. For 1M records with 5 32-bit fields (in addition to a 32-bit key), the 4-way versions of AA Sort, Bitonic Sort, Merge Sort, and
Scalar Merge Sort, respectively, took 5%, 4%, 7%, and 0.1% less time than taken by their 2-way counterparts and these percentages for 15 fields were 9%, 6%, 9%, and 4% respectively.

Figure 2-33 shows the run times for the 4 4-way sort algorithms for 1M records. As we can observe, 4-way Bitonic Sort is the slowest, followed by 4-way AA Sort, followed by 4-way Merge Sort; 4-way Scalar Merge Sort was the fastest. In fact, across all our data sets, 4-way Bitonic Sort took between 16% and 17% more time than taken by 4-way AA Sort, which in turn took between 24% and 35% more time than taken by 4-way Merge Sort. The fastest of them in ByRecord format, 4-way Scalar Merge Sort took, respectively, 88%, 86%, and between 81% and 88% less time than taken by 4-way Bitonic Sort, 4-way AA Sort, and 4-way Merge Sort.
2.4.5.3 Cross layout comparison

Although in a real application one may not be able to choose the layout format for the data to be sorted, it is worthwhile to compare the relative performance of the 8 sort methods using the better layout for each. This means that we use the ByField layout for AA Sort and Merge Sort and the ByRecord layout for Merge Sort and Scalar Merge Sort. Figure 2-34 gives the run times for the 4-way versions using these formats for the case of 1M records. Although Figure 2-34 is only for the case of 1M records, 4-way Scalar Merge Sort was the fastest for all of our data sets. For 5 32-bit fields (in addition to the key field) 4-way Scalar Merge Sort (ByRecord) ran 81% faster than 4-way Merge Sort (ByRecord), 30% faster than 4-way AA Sort (ByField), and 20% faster than 4-way Merge Sort (ByField). When the number of fields was 15, these percentages were 88%, 64% and 60% respectively.
2.5 Summary

We have developed SPU sorting algorithms based on merge sort, shaker sort, and brick sort. Our merge sort adaptation is also a stable sort. Our experiments reveal that while sorting numbers, a standard non-vectorized textbook implementation of merge sort takes about 4 times the time taken by the vectorized merge sort adaptation. Further, the quick sort method that is part of the CBE SDK takes about 9 times the time taken by our merge sort adaptation. Brick sort is the fastest of the shell sort like algorithms—AA sort, shaker sort and brick sort—considered in this chapter, taking about 71% the time taken.
by AA sort to sort 16384 integers. Although cell (bitonic) sort is slightly faster than brick sort, it takes about 31% more time to sort 16384 integers than taken by merge sort. On the down side, merge sort requires $O(n)$ additional space to sort $n$ numbers while the remaining methods require only $O(1)$ added space.

We have shown how to adapt number sorts to sort multifield records on the Cell Broadband Engine. We have also developed two 4-way merge algorithms for the run merging phase. One of these is a scalar version and the other is an SIMD version. Our
Figure 2-32. 2-way and 4-way sorts (ByRecord), 1M records. A) 2-way and 4-way AA sort (ByRecord). B) 2-way and 4-way bitonic sort (ByRecord). C) 2-way and 4-way merge sort (ByRecord). D) 2-way and 4-way scalar merge sort (ByRecord).

Experiments indicate that the 4-way Scalar Merge Sort developed in this paper is the fastest method (from among those tested) to sort multifield records on the CBE.
Figure 2-33. 4-way sorts (ByRecord), 1M records

Figure 2-34. 4-way sorts using the best algorithm for different layouts
CHAPTER 3
SORTING ON A GRAPHICS PROCESSING UNIT (GPU)

3.1 Sorting Numbers on GPUs

One of the very first GPU sorting algorithms, an adaptation of bitonic sort, was developed by Govindraju et al. [22]. Since this algorithm was developed before the advent of CUDA, the algorithm was implemented using GPU pixel shaders. Zachmann et al. [23] improved on this sort algorithm by using Bitonic Trees to reduce the number of comparisons while merging the bitonic sequences. Cederman et al. [24] have adapted quick sort for GPUs. Their adaptation first partitions the sequence to be sorted into subsequences, sorts these subsequences in parallel, and then merges the sorted subsequences in parallel. A hybrid sort algorithm that splits the data using bucket sort and then merges the data using a vectorized version of merge sort is proposed by Sintron et al. [25]. Satish et al. [5] have developed an even faster merge sort. In this merge sort, two sorted sequences A and B are merged by a thread block to produce the sequence C when A and B have less than 256 elements each. Each thread reads an element of A and then does a binary search on the sequence B with that element to determine where it should be placed in the merged sequence C. When the number of elements in a sequence is more than 256, A and B are divided into a set of subsequences by using a set of splitters. The splitters are chosen from the two sequences in such a way that the interval between successive splitters is small enough to be merged by a thread block. The fastest GPU merge sort algorithm known at this time is Warpsort [26]. Warpsort first creates sorted sequences using bitonic sort; each sorted sequence being created by a thread warp. The sorted sequences are merged in pairs until only a small number of sequences remain. The remaining sequences are partitioned into subsequences that can be pairwise merged independently and finally this pairwise merging is done with each warp merging a pair of subsequences. Experimental results reported in [26] indicate that Warpsort is about 30% faster than
the merge sort algorithm of [5]. Another comparison-based sort for GPUs–GPU sample sort–was developed by Leischner et al. [27]. Sample sort is reported to be about 30% faster than the merge sort of [5], on average, when the keys are 32-bit integers. This would make sample sort competitive with Warpsort for 32-bit keys. For 64-bit keys, sample sort is twice as fast, on average, as the merge sort of [5].

[5, 28–32] have adapted radix sort to GPUs. Radix sort works in phases where each phase sorts on a digit of the key using, typically, either a count sort or a bucket sort. The counting to be done in each phase may be carried out using a prefix sum or \textit{scan} [33] operation that is quite efficiently done on a GPU [28]. Harris et al.'s [29] adaptation of radix sort to GPUs uses the radix 2 (i.e., each phase sorts on a bit of the key) and uses the \textit{bitsplit} technique of [33] in each phase of the radix sort to reorder numbers by the bit being considered in that phase. This implementation of radix sort is available in the CUDA Data Parallel Primitive (CUDPP) library [29]. For 32-bit keys, this implementation of radix sort requires 32 phases. In each phase, expensive scatter operations to/from the global memory are made. Le Grand et al. [30] reduce the number of phases and hence the number of expensive scatters to global memory by using a larger radix, $2^b$, for $b > 0$. A radix of 16, for example, reduces the number of phases from 32 to 8. The sort in each phase is done by first computing the histogram of the $2^b$ possible values that a digit with radix $2^b$ may have. Satish et al. [5] further improve the $2^b$-radix sort of Le Grand et al. [30] by sorting blocks of data in shared memory before writing to global memory. This reduces the randomness of the scatter to global memory, which, in turn, improves performance. The radix-sort implementation of Satish et al. [5] is included in NVIDIA's CUDA SDK 3.0. Bandyopadhyay and Sahni [32] developed the radix sort algorithm, GRS, which is suitable for sorting records with many fields. GRS outperforms the SDK radix sort algorithm while sorting numbers by reducing the number of steps and using an additional storage in the global memory. Merrill and Grimshaw [31] have developed an alternative radix sort, SRTS, for GPUs that
is based on a highly optimized algorithm, developed by them, for the scan operation and co-mingling of several logical steps of a radix sort so as to reduce accesses to device/global memory. Presently, SRTS is the fastest GPU radix sort algorithm for 32-bit integers.

The results of [26, 27] indicate that the radix sort algorithm of [5] outperforms both Warpsort [26] and sample sort [27] on 32-bit keys. These results together with those of [31] imply that the radix sort of [31] is the fastest GPU sort algorithm for 32-bit integer keys.

Most of the GPU sorting routines perform some functions repetitively during their operations. These primitives are designed to be easily parallelized on a many-core architecture like a GPU. Two such important primitives that are frequently used in GPU sorting are scan and reduce. Scan is a function that takes a list of \( n \) elements \( [x_0 \ldots x_{n-1}] \) and a binary operator \( \oplus \) as input. The output is also a list of \( n \) elements \( [y_0 \ldots y_{n-1}] \). There are two variants of scan, exclusive and inclusive. For exclusive scan, \( y_i := x_0 \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{i-1} \) for \( i > 0 \) and \( y_0 = \ominus_0 \) (\( \ominus_0 \) is the identity element defined for operator \( \ominus \)) while in the case of inclusive scan, \( y_i := x_0 \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_i \). The reduce operation, on the other hand, produces a single value \( y = x_0 \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1} \). Many useful operations can be designed using these primitives. For example, prefix sum is an exclusive scan with \( + \) as the binary operator. Finding the maximum of a sequence is a reduce with \( \ominus \) as maximum operator that returns the larger of two elements.

### 3.1.1 GPU Sample Sort

Sample sort [27] is a multi-way divide and conquer sorting algorithm that performs better when the memory bandwidth is an issue as the data transferred to and from the global memory is less than in a two-way approach. The serial version of sample sort works by first choosing a set of splitters randomly from the input data. The splitters are then sorted and arranged in increasing order of their values. The input data set is divided into buckets delimited by successive splitters. The elements in a particular
bucket have values that are bounded by the guarding splitters. Each bucket is sorted by recursive application of sample sort. This recursion continues until the size of the bucket becomes less than a certain threshold. At this point a base sorting algorithm is used to sort the small bucket. Figure 3-1 shows the steps of serial sample sort.

SampleSort(a[])
if sizeof(a) ≤ M // a threshold
{
    Sort(a); // Use a sorting method to sort a
    return;
}
Select k elements randomly from a and put them in samples[];
Sort(samples[]);
for (each element e in a[])
{
    find i such that samples[i] ≤ e ≤ samples[i+1];
    Put e in bucket b[i];
}
for (each bucket b[i])
{
    SampleSort(b[i]);
}

Figure 3-1. Serial Sample Sort

To obtain an efficient parallel version of sample sort, it is necessary to balance the size of buckets assigned to thread blocks. This is done by choosing the splitters from a large randomly selected sample from the input. Once the splitters are selected, the numbers are partitioned into buckets by first dividing the data into equal sized tiles with each tile being assigned to a block of threads. A thread block examines its tile of data and assigns numbers in this tile to buckets whose boundaries are the previously chosen splitters. Finally the buckets produced for the tiles are combined to obtain global buckets. The steps in a particular iteration of GPU sample sort of [27] are described below.

3.1.1.1 Step 1–Splitter selection

During this step, the splitters are chosen. First, random samples are taken out of the elements in the buckets. A set of splitters is then chosen from these random samples. Finally, splitters are sorted using odd-even merge sort [34] in shared memory.
and a Binary Search Tree of splitters is created to facilitate the process of finding the bucket for an element.

3.1.1.2 Step 2–Finding buckets

Each thread block is assigned a part of the input data. Threads in a block load the Binary Search Tree into shared memory and then calculate the bucket index for each element in the tile. At the end threads store the number of elements in each bucket as a $k$-entry histogram in the global memory.

3.1.1.3 Step 3–Prefix sum

The per block $k$-entry histograms are prefix-summed to obtain the global offset for each bucket.

3.1.1.4 Step 4–Placing elements into buckets

Each thread block in this phase again calculates bucket index for all keys in its tile. They also calculate the local offset within the buckets. The local offsets are added to the global offsets from the previous phase to get the final position of the numbers.

Figure 3-2 depicts the steps in an iteration of the GPU sample sort.

Figure 3-2. An Iteration of GPU Sample Sort
3.1.2 Warpsort

Warpsort [26] is another comparison based sorting algorithm along the lines of merge sort. It uses bitonic merge to sort the input in a number of stages. Warpsort consists of the following 4 steps.

3.1.2.1 Step 1–Bitonic sort by warps

The input is first divided into a set of tiles and each tile is sorted by a thread warp using bitonic merge sort. A bitonic network for \( n \) elements comprises of \( \log(n) \) phases with each phase \( i (0 \leq i \leq \log(n) - 1) \) having \( (i + 1) \) stages. Figure 3-3 shows the bitonic sorting network for 8 elements with each arrow indicating a compare-exchange operation where two elements are compared and swapped if the first one is greater (or less depending on the direction of the arrow) than the second.

![Figure 3-3. Bitonic Merge Sort of 8 Elements](image)

All the compare-exchanges of a stage needs to be performed before moving to the next stage. This requires a global synchronization among the threads. So, bitonic sort by a thread block will require a `syncthreads()` function call after each stage. In warpsort, as the bitonic sort is done by a warp of threads there is no need for synchronization as threads in a warp are executed in lock-step fashion. However, in each stage, the threads in a warp will be doing compare-exchanges in ascending or descending direction.
This will cause a divergence among the threads in a warp and hence will lead to serial execution of threads. It can be seen that for an $n$ element bitonic network only $n/2$ compare-exchanges are performed and half of them are to form ascending pairs while the other half are to form descending pairs (except during the last stage when all the compare-exchanges are in the same order). Warpsort uses this fact to set the number of elements handled by a 32-thread warp to be 128. Each thread then performs two operations during each stage of bitonic sort and operations performed by the all threads in a warp are either for an ascending pair or a descending pair. Thread divergence is thus avoided by each thread performing compare-exchanges to generate results in a particular sorted order during each step of its execution.

3.1.2.2 Step 2–Bitonic merge by warps

The sorted sequences are merged in this step by a warp using bitonic merge until the number of sequences falls below a threshold. As the threads in a warp only merge a fixed number of elements $t$ in a step, sequences are divided into buffers of size $t/2$. To merge two sorted sequences A and B, $t/2$ elements from each of the sequences are fetched into the buffers in shared memory and then they are merged using the bitonic merge algorithm (bitonic merge is same as the last step of bitonic sort outlined in Figure 3-3). The smallest $t/2$ elements of the merged sequence are output and the remaining $t/2$ elements are kept in the buffer. Then the last elements of the $t/2$ element buffers of A and B are checked to find out from which sequence the next buffer load of $t/2$ elements should be fetched.

3.1.2.3 Step 3–Splitting long sequences

As the number of the sequences decreases geometrically during merging, after some point there are not enough sequences to be merged on different SMs of the GPU. In this step, the long sequences are split into small sequences which can be merged using all the SMs in the GPU. To split $l$ sequences of size $n$ into subsequences of size $s$, a random sample of $s*k$ elements are chosen from the sequences where $k$ is a constant.
whose value depends on the trade-off between choosing more random samples for a good set of splitters and the time needed for choosing those random samples. The set of random samples is then sorted and the elements in positions that are a multiple of $k$ form the global set of splitters. These $s$ splitters are then used to partition each of these $l$ sequences into $s$ subsequences.

### 3.1.2.4 Step 4–Final merge by warps

This step is essentially the same as Step 2. The Smaller sequences produced in Step 3 can now be merged independently by thread warps to produce the final output. Figure 3-4 gives all the steps in Warpsort.

![Figure 3-4. Warpsort Steps](image)

### 3.1.3 SRTS Radix Sort

SRTS employs a highly optimized version of the scan kernel developed by Merrill and Grimshaw [31] to perform radix sort. The scan process consists of three kernels, bottom level reduce, top level scan, and bottom level scan.

**Bottom Level Reduce** In this step, the threads in a thread block read inputs and produces a final output per thread block. The scan does not use the traditional method
of assigning a unique thread to each input element. Instead, it fixes the number \( C \) of thread blocks performing the reduction. Merrill and Grimshaw [31] experimented with different configurations of empty kernels doing just reading the numbers (one, two or four at a time) from the global memory and then writing out a single value to the global memory to find the optimum \( C \) and the number of threads in a block. Threads in a block loop over the input elements in batches to perform the reduction. The dependencies between different batches are carried over in the SM registers or local shared memory. Threads in a block operate in three different phases while reducing the entire input assigned to it. Firstly, threads read multiple input elements from the current batch and do a serial reduce using the registers. The next phase is the serial reduction in the shared memory by groups of threads until the number of outputs is small enough to be reduced by a single warp of threads. In the last phase, the warp reduces using the strategy of Figure 3-8 to produce the final value for the current batch. The value is also stored in the shared memory to be used when reducing the elements from the next batch. Using this strategy for reducing the input increases the number of arithmetic operations performed by the threads and hence increases the number of instructions per memory transaction. This helps in offsetting the idle time of an SM that is waiting for data from global memory for a predominantly memory-bound operation like reduction. Figure 3-5 gives a schematic of the entire process. The circles indicates a thread working on two elements.

At the end of bottom level reduce, there are \( C \) partial reductions.

**Top Level Scan** A single thread block scans the \( C \) values from the previous phase along with some residual input elements, if any, that are not reduced in case the number of input elements is not a perfect multiple of number of blocks. As only a few elements are scanned, this phase does not contribute much to the overall runtime.

**Bottom Level Scan** This is essentially the same as the first phase but the scans are seeded with the values obtained from Top Level Scan. Although similar steps are
Figure 3-5. Bottom Level Reduction

carried out, the operation is scan instead of reduce. The number of inputs and outputs
are same for the scan operation. Merrill and Grimshaw [31] experimented with empty
kernels that read and write values to and from the global memory to determine the
optimal values of $C$ and number of threads in a block.

SRTS radix sort starts with the three-step scan process. The scan and the reduce
kernels use the summation operation calculating the prefix sum and sum of all elements,
respectively. SRTS augments the kernels with the operations required to radix sort a
number of integers and hence only 3 kernel launches are needed to do a radix sort with
fewer number of intermediate values passed around compared to SDK radix sort [5]. As
with the other radix sort strategies SRTS progressively radix sorts on 4 bits per phase
phase. Hence, SRTS requires 8 phases to completely sort 32-bit integers. Each phase,
as mentioned earlier consists of 3 steps.

3.1.3.1 Step 1–Bottom level reduce

This step adds an extract phase before the bottom level reduce operation. In the
extract phase, $r = 16$ flag vectors corresponding to every possible value of 4 bits are
constructed from the input. For example, if a thread block inputs 64 elements at a time,
16 64-element bit vectors are constructed from the input elements. For the $i$th input
element having a value of $j (0 \leq j \leq 15)$ in the current 4 bits being considered, the
ith bit of the jth vector is set of 1 while the ith bit of other vectors are set to 0. In the next phase, bottom level prefix sum is done on these 16 vectors serially one at a time. The total for each digit is carried over for the next batch of input to be processed by the thread block.

3.1.3.2 Step 2–Top level scan

In this phase a single block of threads operates over the prefix sums to compute the global prefix sum. The scan is modified to handle the scanning of 16 different sets of partial sums one at a time.

3.1.3.3 Step 3–Bottom level scan

Bottom level scan is augmented with two different operations, one performed before the scan and one after the scan. The first one is reading the input and extracting the bits as done in the first step. During the second phase, a bottom level scan is performed seeded with the values from top level scan of Step 2. As in the first phase, 16 bottom level scans are performed each for 16 possible digit values for 4 bits. After the scan, the sorted position of the input elements with the current batch is known. This position is used in the next phase, when the elements are placed into their sorted order in the final output. First, depending on the prefix sum positions obtained, a shared memory exchange is performed to put the elements within the batch in sorted order. After this the threads read consecutive elements from the shared memory and write them to global memory depending on the global offset and the local position. As with other radix sort methods described earlier, this strategy generates larger global memory transactions as consecutive elements within the shared memory are written to nearby positions in the global memory. As with the first phase, the radix counters are accumulated and are carried over to the next batch of input processed by this block of threads.

Figure 3-6 depicts the three stream kernels augmented with extract and scatter operations.
3.1.4 SDK Radix Sort Algorithm

The SDK radix sort algorithm [5] uses a radix of $2^b$ with $b = 4$ ($b = 4$ was determined, experimentally, to give best results). With $b = 4$ and 32-bit keys, the radix sort runs in 8 phases with each phase sorting on 4 bits of the key. Each phase of the radix sort is accomplished in 4 steps as below. These steps assume the data is partitioned (implicitly) into tiles of size 1024 numbers each. There is a separate kernel for each of these 4 steps.

Step 1: Sort each tile on the $b$ bits being considered in this phase using the bit-split algorithm of [33].

Step 2: Compute the histogram of each tile. Note that the histogram has $2^b$ entries with entry $i$ giving the number of elements in the tile for which the $b$ bits considered in this phase equal $i$.

Step 3: Compute the prefix sum of the histograms of all tiles.
Step 4: Use the prefix sums computed in Step 3 to rearrange the numbers into sorted order of the $b$ bits being considered.

### 3.1.4.1 Step 1–Sorting tiles

For Step 1, each SM inputs a tile into shared memory using 256 threads (8 warps). So, each thread reads 4 consecutive numbers facilitating a coalesced read from the global memory. Each thread inputs the 4 numbers using a variable of type `int4`, which is 128 bits long. The input data are stored in registers. Next, the 256 threads collaborate to do $b$ rounds of the bit split scheme of [33]. In each round, all the numbers having 0 in that bit place are moved forward followed by the ones having 1 in the bit position being considered as shown in Figure 3-7.

![Figure 3-7. Bit-split Scheme for Sorting Numbers on 4 Bits](image)

To move the numbers having 0s ahead of the ones having 1s, the position of each number in the output is determined and then the number is written into the shared memory in that position. For the next round, threads read the numbers in the sorted order of the previous bit from the shared memory. A prefix scan based algorithm called warp-scan is executed by each thread to find out the rank of the 4 numbers it has read. First, each thread calculates the number of zeroes in the current bit position of the 4 numbers read by it. It then writes the value in a particular location in the shared memory depending on its thread id ($tid$) constructing a count vector per warp. Additional storage is allocated in the shared memory adjacent to the count vector and zeored out before the calculation begins. An exclusive prefix scan of the count vector is performed by the
32 threads in a warp in \( \log(32) = 5 \) steps. During step \( i \) \((i = 0...4)\), each thread adds its own count with the count which is \( 2^i \) away as shown in the algorithm of Figure 3-8 and corresponding schematic in Figure 3-9. This algorithm has no divergence within a warp. However, spurious additions are performed by the threads during each step other than the first step when each addition is necessary.

Algorithm warp-scan()
{
    // Compute the prefix sum of count vector
    // warpId is the warp Id = tid >> 5
    // warpSize is 32  idx = 2 * warpId * warpSize + tid;
    shared[idx] = 0 // zero out the location
    idx += warpSize;
    shared[idx] = val; // Copy the count here
    shared[idx] += shared[idx - 1];
    shared[idx] += shared[idx - 2];
    shared[idx] += shared[idx - 4];
    shared[idx] += shared[idx - 8];
    shared[idx] += shared[idx - 16];
    // Convert inclusive to exclusive
    return (shared[idx - val]);
}

Figure 3-8. Divergence Free Warp Scan Algorithm

Figure 3-9. Warp Scan of 8 Numbers

A final scan is performed over the last values of 8 warps to find out the final position of the 4 numbers read by a thread. Each thread then writes the numbers out to the shared memory according to the determined position. Following these \( b \) rounds of the bit-split algorithm, the tile is in sorted order (of the \( b \) bits being considered) in registers. The sorted tile is then written out to global memory.
3.1.4.2 Step 2–Calculating histogram

In this step, the numbers for each half tile are input (in sorted order) using 256 threads per SM and the histogram of the $b$ bits being considered is computed for the half tile input. Note that, one can instead input the entire tile and remaining steps would work fine but, empirically, working with half-tiles from this point onwards performs better. For this computation, each thread inputs two numbers from global memory and writes these to shared memory. The threads then determine the up to 15 places in the input half tile where the 4 bits being considered change. For this, each thread simply checks its own number with the next one. If the bits differ, it records the position in a shared memory array. Once all the threads are done, all positions in the half-tile where the bits change are recorded in the array as shown by the broken arrows in Figure 3-10

![Figure 3-10. Calculating Histogram Offsets](image)

Once the positions are determined, the size of each bucket (i.e., number of elements having same bit values) is determined by finding out the intervals between the positions by the same threads which found the positions during the previous step. The histogram is written out to global memory in column major order, i.e., bucket 0 of all half-tiles followed by bucket 1 of all half-tiles all the way up to bucket 15 of all half-tiles. The positions in a half tile where the bits differ are also written out to global memory as this information is needed in the final step.

3.1.4.3 Step 3–Prefix sum of histogram

In Step 3, the prefix sum of the half-tile histograms is computed using the prefix sum code available in CUDPP [29] and written to global memory as shown in the Figure 3-11.
3.1.4.4 **Step 4—Rearrangement**

In Step 4, each SM inputs a half tile of data, which is sorted on the \( b \) bits being considered, and uses the computed prefix sum histogram for this half tile to write the numbers in the half tile to their correct position in global memory. Each thread in the SM reads 2 numbers as before corresponding to its id. The tile offsets calculated in Step 2 and the prefix sum histogram from Step 3 are also read. As the half tiles are already sorted following Step 1, the thread id is the same as the position of the number read in the tile. If the value of the \( b \) bits being considered for the current number is \( r \), the prefix sum histogram gives how many numbers there are with \( b \) bit value less than \( r \) and also how many numbers in the tiles before it have bit value exactly equal to \( r \). However, one also needs to find out how many of the numbers in the current tile have bit value exactly equal to \( r \) and occur before the number in the sorted order. As per Step 2, the tile offset for bit value \((r - 1)\) read from the global memory indicates the place where the bit value change occurs from \((r - 1)\) to \( r \) i.e. the place from where numbers with bit value \( r \) starts. Hence, there are \((tid - tileOffset[r - 1])\) numbers before the current number having bit values exactly equal to \( r \). Hence, the final position of the number will be the sum of this local offset and the prefix sum histogram. As the numbers are already in sorted order when read by threads, consecutive numbers in the tile read by consecutive threads in a half-warp are written to consecutive locations in global memory. This reduces the random scattering by a significant amount as most of the writes produce coalesced...
memory transactions. Following this, all numbers are in sorted order, in global memory, with respect to the $b$ bits being considered in this phase of the radix sort.

3.1.5 GPU Radix Sort (GRS)

Like the SDK radix sort algorithm of [5], GRS accomplishes a radix sort using a radix of $2^b$ with $b = 4$. Each phase of the radix sort is done in 3 steps with each step using a different kernel. For purposes of discussion, we assume a tile size of $t$ ($t = 1024$ in the SDK implementation). We define the rank of number $i$ in a tile to be the number of integers in the tile that precede number $i$ and have the same value as of number $i$. Since we compute ranks in each phase of the radix sort, number equality (for rank purposes) translates to equality of the $b$ bits of the number being considered in a particular phase. Note that when the tile size is 1024, ranks lie in the range 0 through 1023. The three steps in each phase of GRS are:

Step 1: Compute the histogram for each tile as well as the rank of each number in the tile. This histogram is the same as that computed in Step 2 of the SDK radix sort.

Step 2: Compute the prefix sums of the histograms of all tiles.

Step 3: Use the ranks computed in Step 1 to sort the data within a tile. Next, use the prefix sums computed in Step 2 to rearrange the numbers into sorted order of the $b$ bits being considered.

Step 1 requires us to read the numbers in a tile from global memory, compute the histogram and ranks, and then write the computed histogram and ranks to global memory. Step 2 is identical to Step 3 of the SDK algorithm. In Step 3, numbers are again read from global memory. The numbers in a tile are first reordered in shared memory to get the sorted arrangement of Step 1 of the SDK algorithm and then written to global memory so as to obtain the sorted order following Step 4 of the SDK algorithm. This writing of numbers from shared memory to global memory is identical to that done in Step 4 of the SDK algorithm. The following subsections provide implementation
details for the 3 steps of GRS. Note that the sorting done by the SDK radix sort during the very first step is now done using the ranks calculated in the first step and the sorting is merged into the last step of GRS where the numbers are finally put into their sorted order. As the rank calculation is done without any overhead while calculating the histogram, GRS is doing less work compared to SDK radix sort. However, GRS uses an additional storage for storing the ranks.

3.1.5.1 Step 1–Histogram and Ranks

An SM computes the histograms and ranks for 64 tiles at a time employing 64 threads. Figure 3-12 gives a high-level description of the algorithm used by us for this purpose. Our algorithm processes 32 numbers from each tile in an iteration of the for loop. So, the number of for loop iterations is the tile size \( t \) divided by 32. In each iteration of the for loop, the 64 threads cooperate to read 32 numbers from each of the 64 tiles. This is done in such a way (described later) that global memory transactions are 128 bytes each. The data that is read is written to shared memory. Next, each thread reads the 32 numbers of a particular tile from shared memory and updates the tile histogram, which itself resides in shared memory. Although we have enough registers to accommodate the 64 histograms, CUDA relegates a register array to global memory unless it is able to determine, at compile time, the value of the array index. To maintain the histograms in registers, we need an elaborate histogram update scheme whose computation time exceeds the time saved over making accesses to random points in an array stored in shared memory. When a number is processed by a thread, the thread extracts the \( b \) bits in use for this phase of the radix sort. Suppose the extracted \( b \) bits have the value 12, then the current histogram value for 12 is the rank of the number. The new histogram value for 12 is 1 more than the current value. The determined rank is written to shared memory using the same location used by the number (i.e., the rank overwrites the number). Note that once a number is processed, it is no longer needed by Algorithm HR. Once the ranks for the current batch of 32
numbers per tile have been computed, these are written to global memory and we proceed to the next batch of 32 numbers. To write the ranks to global memory, the 64 threads cooperate ensuring that each transaction to global memory is 128 bytes. When the for loop terminates, we have successfully computed the histograms for the 64 tiles and these are written to global memory.

Algorithm HR()
{
  // Compute the histograms and ranks for 64 tiles
  itrs = t / 32; // t = tile size
  for(i = 0; i < itrs; i++)
  {
    Read 32 numbers from each of the 64 tiles;
    Determine the ranks and update the histograms;
    Write the ranks to global memory;
  }
  Write the histograms to global memory;
}

Figure 3-12. Algorithm to Compute the Histograms and Ranks of 64 Tiles

To ensure 128-byte read transactions to global memory, we use an array that is declared as:

shared int4 sKeys4[512];

Each element of sKeys4 is comprised of 4 4-byte integers and the entire array is assigned to shared memory. A thread reads in 1 element of sKeys4 at a time from global memory and in so doing, 4 numbers are input. It takes 8 threads to cooperate so as to read in 32 numbers (or 128 bytes) from a tile. The 16 threads in a half warp read 32 numbers from each of 2 tiles. This read takes 2 128-byte memory transactions. With a single read of this type, the 64 threads together are able to read 32 numbers from a total of 8 tiles. So, each thread needs to repeat this read 8 times (each time targeting a different tile) in order for the 64 threads to input 32 numbers from each of 64 tiles. Besides maximizing bandwidth utilization from global memory, we need also be concerned about avoiding shared memory bank conflicts when the threads begin to process the numbers for their assigned tile. Since shared memory is divided into 16 banks of 4-byte words, storing the numbers in a natural way results in the first number
of each tile residing in the same bank. Since in the next step, each thread processes its
tile’s numbers in the same order, we will have shared memory conflicts that will cause
the reads of numbers to be serialized within each half warp. To avoid this serialization,
we use a circular shift pattern to map numbers to the array sKeys4. The CUDA kernel
code to do this read is given in Figure 3-13.

```c
// tid is the thread id and bid is the block id
// Determine the first tile handled by this thread
startTile = (bid * 64 + tid / 8) * (t / 4);
// starting number position in the tile
// keyOffset is the offset for current 32 keys
keyPos = keyOffset + tid % 8;
// shared memory position to write the keys with
// circular shift
sKeyPos = (tid / 8) * 8 + (((tid / 8) % 8) +
(tid % 8)) % 8;
// some constants
tileSize8 = 8 * (t / 4); tid4 = tid * 4;
// Initialize the histogram counters
for(i = 0; i < 16; i++)
{
    sHist[tid * 16 + i] = 0;
}
// Wait for all threads to finish
syncthreads();
curTileId = startTileId;
for(i = 0; i < 8; i++)
{
    sKeys4[sKeyPos + i * 64] = keysIn4[keyPos +
startTile];
curTileId += tileSize8;
}
syncthreads();
```

Figure 3-13. Reading the Numbers from Global Memory

As stated earlier, to compute the histograms and ranks, each thread works on the
numbers of a single tile. A thread inputs one element (4 numbers) of sKeys4, updates
the histogram using these 4 numbers and writes back the rank of these 4 numbers (as
noted earlier the equals the histogram value just before it is updated). Figure 3-14 gives
the kernel code to update the histogram and compute the ranks for the 4 numbers in one
element of sKeys4.
// Update the histograms and calculate the rank
// startbit is the starting bit position for
// this phase
int4 p4, r4;
for(i = 0; i < 8; i++)
{
    p4 = sKeys4[(tid4) + (i + tid) % 8];
    r4.x = sHist[((p4.x >> startbit) & 0xF) * 64 +
                 tid]++;
    r4.y = sHist[((p4.y >> startbit) & 0xF) * 64 +
                 tid]++;
    r4.z = sHist[((p4.z >> startbit) & 0xF) * 64 +
                 tid]++;
    r4.w = sHist[((p4.w >> startbit) & 0xF) * 64 +
                 tid]++;
    sKeys4[tid4 + (i + tid) % 8] = r4;
}
syncthreads();

Figure 3-14. Processing an Element of sHist4[]

Once the ranks have been computed, they are written to global memory using a
process similar to that used to read in the numbers. Figure 3-15 gives the kernel code.

Here, ranks4[] is an array in global memory; its data type is int4.

curTileId = startTileId;
for(i = 0; i < 8; i++)
{
    ranks4[keyOffset + keyPos + startTileId] =
    sKeys4[sKeyPos + i * 64] ;
    curTileId += tileSize8;
}
syncthreads();

Figure 3-15. Writing the Ranks to Global Memory

When an SM completes the histogram computation for the 64 tiles assigned to
it, it writes the computed 64 histograms to the array counters in global memory. If we
view the 64 histograms as forming a $16 \times 64$ array, then this array is mapped to the one
dimensional array in column-major order. Figure 3-16 gives the kernel code for this.

3.1.5.2 Step 2–Prefix sum of tile histograms

As in Step 3 of the SDK algorithm, the prefix sum of the tile histograms is computed
with a CUDPP function call. The SDK radix sort computes the prefix sum of half tiles
// calculate Id of threads amongst all threads
// nTiles is total number of tiles
globalTid = bid * 64 + tid;
for(i = 0; i < 16; i++)
{
    counters[i * nTiles + globalTid] = sHist[i * 64 + tid];
}

Figure 3-16. Writing the Histograms to Global Memory

while we do this for full tiles. Assuming both algorithms use the same tile size, the prefix sum in Step 2 of GRS involves half as many histograms as does the prefix sum in Step 3 of the SDK algorithm. This difference, however, results in a negligible reduction in run time for Step 2 of GRS versus Step 3 of SDK.

3.1.5.3 Step 3–Positioning numbers in a tile

To move the numbers in a tile to their correct overall sorted position with respect to the $b$ bits being considered in a particular phase, we need to determine the correct position for each number in a tile. The correct position is obtained by first computing the prefix sum of the tile histogram. This prefix sum may be computed using the warp scan algorithm used in the SDK radix sort code corresponding to the algorithm of [5]. The correct position of a number is its rank plus the histogram prefix sum corresponding to the $b$ bits of the number being considered in this phase. As noted earlier, moving numbers directly from their current position to their final correct positions is expensive because of the large number of memory transactions to global memory. Better performance is obtained when we first rearrange the numbers in a tile into sorted order of the $b$ bits being considered and then move the numbers to their correct position in the overall sorted order. Figure 3-17 gives the kernel code. One SM reorders the numbers in a single tile. Since each thread handles 4 numbers, the number of threads used by an SM for this purpose (equivalently, the number of threads per thread block) is $t/4$. 

80
kernel reorderData(keysOut, keysIn4, counters, countersSum, ranks4)
{
    // Read the numbers from keysIn4 and put them in
    // sorted order in keysOut
    // sTileCnt stores tile histogram
    // sGOffset stores global prefix-summed histogram
    shared sTileCnt[16], sGOffset[16];
    // storage for numbers
    shared int sKeys[t];
    int4 k4, r4;
    // Read the histograms from the global memory
    if(tid < 16)
    {
        sTileCnt[tid] = counters[tid * nTiles + bid];
        sGOffset[tid] = countersSum[tid * nTiles + bid];
    }
    syncThreads();
    // Perform a warp scan on the tile histogram
    sTileCnt = warpScan(sTileCnt);
    syncThreads();
    // read the numbers and their ranks
    k4 = keysIn4[bid * (t / 4) + tid];
    r4 = ranks4[bid * (t / 4) + tid];
    // Find the correct position and write to the
    shared mem
    r4.x = r4.x + sTileCnt[(k4.x >> startbit) & 0xF];
    sKeys[r4.x] = k4.x;
    // Similar code for y, z and w components of
    // r4 and k4 comes here
    syncThreads();
    // Determine the global rank
    // Each thread places 4 numbers at positions
    // tid, tid + 1/4, tid + 1/2 and tid + 3/2
    radix = (sKeys[tid] >> startbit) & 0xF;
    globalOffset.x = sGOffset[radix] + tid - sTileCnt[radix];
    keysOut[globalOffset.x] = sKeys[tid];
}

Figure 3-17. Rearranging Data

### 3.1.6 Comparison of Number Sorting Algorithms

According to the results reported in [27] and [26], radix sort based algorithms perform better than sample sort and warpsort. Hence, here, we compare SDK radix sort, GRS and SRTS\(^1\). These algorithms were coded and run using NVIDIA CUDA SDK

\(^1\) We are grateful to the authors of SRTS [31] for making their code available.
3.0 on a Tesla C1060 GPU. For each of the experiments, the run time reported is the average run time for 100 randomly generated sequences. While comparing SDK, GRS and SRTS same random sequences are used for the 3 algorithms.

For the first set of comparisons, SDK, GRS and SRTS are used to sort from 1M to 10M numbers. As shown in Figure 3-18A, SDK runs 20% to 7% faster than GRS for 1M to 3M numbers respectively. However, GRS outperforms SDK for sorting 4M numbers and more. It runs 11% to 21% faster than SDK for 4M to 10M numbers, respectively. SRTS is the best performing algorithm among the three by running 53% to 57% faster than GRS for 1M to 10M numbers. This performance differential is also observed while sorting even larger sets of numbers. Figure 3-18B shows the run times of SDK, GRS and SRTS starting from 10M numbers with an increment of 10M. For 100M numbers, GRS runs 21% faster than SDK and 53% slower than SRTS.

![Figure 3-18. Sorting numbers using radix sorts. A) Sorting 1M to 10M Numbers. B) Sorting 10M to 100M Numbers.](image)

3.2 Sorting Records on GPUs

3.2.1 Record Layouts

A record $R$ is comprised of a key $k$ and $m$ other fields $f_1, f_2, \cdots, f_m$. For simplicity, we assume that the key and each other field occupies 32 bits. Let $k_i$ be the key of record $R_i$ and let $f_{ij}, 1 \leq j \leq m$ be this record’s other fields. With our simplifying assumption
of uniform size fields, we may view the $n$ records to be sorted as a two-dimensional array $\text{fieldsArray}[][]$ with $\text{fieldsArray}[i][0] = k_i$ and $\text{fieldsArray}[i][j] = f_{ij}$, $1 \leq j \leq m$, $1 \leq i \leq n$. When this array is mapped to memory in column-major order, we get the $\text{ByField}$ layout of [21]. This layout was used also for the AA-sort algorithm developed for the Cell Broadband Engine in [11] and is essentially the same as that used by the GPU radix sort algorithm of [5]. When the fields array is mapped to memory in row-major order, we get the $\text{ByRecord}$ layout of [21]. A third layout, $\text{Hybrid}$, is employed in [31]. This is a hybrid between the $\text{ByField}$ and $\text{ByRecord}$ layouts. The keys are stored in an array and the remaining fields are stored using the $\text{ByRecord}$ layout. Essentially then, in the $\text{Hybrid}$ layout, we have two arrays. Each element of one array is a key and each element of the other array is a structure that contains all fields associated with an individual record. In this paper, we limit ourselves to the $\text{ByField}$ and $\text{Hybrid}$ layouts. We do not consider the $\text{ByRecord}$ layout as it appears that most effectively way to sort in this layout is to first extract the keys, sort (key, index) pairs and then reorder the records into the obtained sorted permutation. The last two steps are identical to the steps in an optimal sort for the $\text{Hybrid}$ layout. So, we expect that good strategies to sort in the $\text{Hybrid}$ layout will also be good for the $\text{ByRecord}$ layout. When the sort begins with data in a particular layout format, the result of the sort must also be in that layout format.

### 3.2.2 High level Strategies for Sorting Records

At a high level, there are two very distinct approaches to sort multifield records [35]. In the first, we construct a set of tuples $(k_i, i)$, where $k_i$ is the key of the $i$th record. Then, these tuples are sorted by extending a number sort algorithm so that whenever the number sort algorithm moves a key, the extended version moves a tuple. Once the tuples are sorted, the original records are rearranged by copying records from the $\text{fieldsArray}$ to a new array placing the records into their sorted positions in the new array or in-place using a cycle chasing algorithm as described for a table sort in [20]. The second strategy is to extend a number sort so as to move an entire record every time its
key is moved by the number sort. We call the first strategy as indirect and the second strategy as direct strategy for sorting multifield records. There are advantages and disadvantages to each strategy. Indirect strategy seems to perform much less work than the direct during sorting as the satellite data that needs to be moved with the key is only an integer index while in the direct strategy its the entire record. On the flip side, the indirect strategy has a very costly random global memory access phase at the end when records are moved to their sorted positions whereas the direct strategy does not have this phase.

Our focus, is to extend GPU number sorting algorithms to handle records with multiple fields using different layouts. Specifically, we extend the algorithms which are the fastest comparison and non-comparison based sorting algorithms. Warpsort and sample sort are both comparison based algorithms with similar reported performance. So, we selected one of these, sample sort, to extend to the sorting of multifield records. Among non-comparison based sorting methods SRTS is the fastest for sorting numbers while GRS is the fastest for sorting large records in the Hybrid layout (Section 3.2) using the direct strategy (Section 3.2). So, we extend SRTS for sorting records. and GRS to sort records in other layouts.

### 3.2.3 Sample Sort for Sorting Records

Sample sort \[27\] is a multi-way divide and conquer sorting algorithm which performs better when the memory bandwidth is an issue as the data transferred to and from the global memory is less than two-way approach. Figure 3-1 shows the steps of serial sample sort.

To obtain an efficient parallel version of sample sort, it is necessary to balance the size of buckets assigned to thread blocks. This is done by choosing the splitters from a large randomly selected sample of keys. Once the splitters are selected, the records are partitioned into buckets by first dividing the data into equal sized tiles with each tile being assigned to a block of threads. A thread block examines its tile of data and assigns
records in this tile to buckets whose boundaries are the previously chosen splitters. Finally the buckets produced for the tiles are combined to obtain global buckets. The step in the GPU sample sample sort of [27] are described below.

**Phase 1:** During this phase, the splitters are chosen. First, a set of random samples are taken out of the elements in the buckets. A set of splitters are then chosen from these random samples. Finally, splitters are sorted using odd-even merge sort [34] in shared memory and a Binary Search Tree of splitters is created to facilitate the process of finding the bucket for an element.

**Phase 2:** Each thread block is assigned a part of the input data. Threads in a block load the Binary Search Tree into shared memory and then calculate the bucket index for each element in the tile. At the end threads store the number of elements in each bucket as a $k$-entry histogram in the global memory.

**Phase 3:** The per block $k$-entry histograms are prefix-summed to obtain the global offset for each bucket.

**Phase 4:** Each thread block in this phase again calculates bucket index for all keys in its tile. They also calculate the local offset within the buckets. The local offsets are added to the global offsets from the previous phase to get the final position of the records.

When sample sorting records using the direct strategy outlined in Section 3.2 the records need to be moved only during the fourth phase as in all other phases only the keys are required to be moved. This distribution of the records from the large bucket to small buckets is repeated multiple times till the size of the bucket is below a specified threshold. Finally, quicksort is done on the records when the bucket size is small. Records are also moved during the partitioning phase of the quicksort within a small bucket. The fourth phase and the quick sort part of sample sort can be extended to handle records in *ByField* and *ByRecord* format. In *ByField* layout, while moving
fieldsArray[i] to outfieldsArray[j], threads can move the corresponding fields as shown in Figure 3-19.

```cpp
outKey[j] = key[i];
//Move the fields
for(f = 1; f <= m; f++) {
    outfieldsArray[j][p] = fieldsArray[i][p];
}
```

Figure 3-19. Moving records in ByField layout

Similarly, in ByRecord (Hybrid) format fields can be moved by a thread while moving the keys. Figure 3-20 shows the code to move records assuming that fieldsArray[i] and outfieldsArray[j] are structures that contain the fields.

```cpp
outKey[j] = key[i];
//Move the fields
outfieldsArray[j] = fieldsArray[i];
```

Figure 3-20. Moving records in ByRecord layout

We observe that in the ByField layout, threads in a warp access adjacent elements in global memory resulting in coalescing of a memory access while in the ByRecord layout, threads in a warp access words in the global memory that are potentially far apart generating global memory transactions of size at most 16 bytes. We employ a strategy of grouping the threads together so that we can generate larger memory transactions. Rather than a single thread reading and writing the entire record, we employ a group of threads to read and write the records into the global memory cooperatively. Then this same group of threads iterates to read and write other records co-operatively. This ensures larger global memory transactions. As an example, lets say the record is of 64 bytes in length and as each thread can read in 16 bytes of data using an int4 datatype, we can group 4 threads together so that they can read the entire record together. Then this thread group iterates over to read other records until all the records assigned to the thread group has been read. Let numThreads denote the number of threads in a block and suppose that each thread is to read in one record
and put it into proper place in the output array. Assume that records from `startOffset` to 
`((startOffset + numThreads)` are processed by this thread block. For sake of clarity of the 
 pseudocode we assume that there is a map `mapInToOut` which determines the proper 
position in the output array. In case of sample sort, it would be the Binary Search tree 
constructed out of the splitters which would determine the position of a particular record 
in the output array. Figure 3-21 outlines the optimized version of moving records using 
coalesced read and write.

```c
// Determine the number of threads required to 
// read the entire record
numThrdsInGrp = sizeof(Rec) / 16;
// Total number of records to be read = number of 
// threads in the group
numItrs = numThrdsInGrp;
// Number of records read in a single iteration 
// by all threads
nRecsPerItr = numThrds / numThrdsInGrp;
// Convert Record arrays to int4 arrays
recInt4 = (int4 *)rec; outRecInt4 = (int4 *)outRec;
// Determine the starting record and position in the group for this thread
startRec = startOffset + threadId / numThrdsInGrp;
posInGrp = threadId % numThrdsInGrp;
for(i = 0; i < numItrs; i++)
{
    outRecInt4[mapInOut(startRec) + posInGrp] =
    recInt4[startRec + posInGrp];
    startRec += numThrdsInGrp;
}
```

Figure 3-21. Optimized version of moving records in ByRecord layout

### 3.2.4 SRTS for Sorting Records

SRTS employs a highly optimized version of the scan kernel developed by Merrill 
and Grimshaw[31] to perform radix sort. As with the other radix sort strategies 
it progressively radix sorts on 4 bits a phase. Hence, SRTS requires 8 phases to 
completely sort 32-bit integers. SRTS focuses on reducing the total number of reads 
and writes to the global memory by combining different functions done in separate 
kernels. The performance of most of the kernels in earlier radix sort implementation is 
limited by the bandwidth between SM and global memory. The technique introduced
in SRTS increases the arithmetic intensity of these memory bound operations and eliminates the need for additional kernels for sorting as performed in SDK radix sort by [5]. SRTS further brings parity between computation and memory access by only having a fixed number of thread blocks in the GPU. Each thread loops over to process the data in batches and hence the amount of computation done per thread increases substantially. SRTS uses a fixed number of thread blocks enough to occupy all SMs in the GPU. The input data is divided into tiles and a set of tiles is assigned to a thread block. It uses a radix of $2^b$ with $b = 4$. With $b = 4$ and 32-bit keys, the radix sort runs in 8 phases with each phase sorting on 4 bits of the key. Each thread block while processing a tile finds out the number of keys with a particular radix value. The radix counters indicating number of keys with a particular radix value are accumulated over the tiles assigned to that thread block. SRTS consists of following steps [31].

**Phase 1- Bottom Level Reduction:** This phase consists of two sub-phases. During the first phase, each thread reads in an element from the input data and extracts the $b$ bits being considered and increases the histogram counts correspondingly. The threads in the thread block loop over the tiles assigned to the thread block and the radix counters are accumulated in the local registers as there are only 16 different radix values for 4 bits. After the last tile of input data is processed, the threads within a block perform a local prefix sum cooperatively to prefix sum the sequence of counters and the result is written to the global memory as a set of prefix sums.

**Phase 2 - Top Level Scan:** In this phase a single block of threads operates over the prefix sums to compute the global prefix sum. The block-level scan is modified to handle a concatenation of partial prefix sums.

**Phase 3 - Bottom Level Scan:** Lastly, the threads in a block sort the elements by first reading in the prefix sum calculated during the top-level scan phase. The thread block reads in the elements again and extracts the $b$ bits being considered. A local parallel scan is done to find the local prefix sum. These local offsets are seeded with the
global prefix sums calculated earlier to get the final position of the element in the output. The input elements are next scattered in shared memory using the local offsets to put them in sorted order within the tile. Finally, those elements are read in sorted order from the shared memory and written onto the global memory. This strategy ensures better memory coherence and generates larger global memory transactions. As with the first phase, the radix counters are accumulated and are carried over to the next tile of input processed by this block of threads using local registers.

The final scatter of input elements happens during the very last phase. Only keys of the records are required during other phases. So, the fields of the record can also be moved during the third phase while scattering the keys. We can use the strategies outlined in Figures 3-19 and 3-20 to scatter the fields in ByField and ByRecord layouts respectively. However, due to the way SRTS is implemented using generic programming it is difficult to use the an optimized version of record moving (Figure 3-21) in ByRecord format. The third phase of record scattering occurs only 8 times for 32-bit keys during the entire sorting process and does not depend on the number of records being sorted. This indicates there is a possibility, for SRTS, direct strategy of moving records while sorting might actually perform better than the indirect strategy of sorting \((key, index)\) pairs followed by rearrangement.

3.2.5 GRS for Sorting Records

GRS is developed specifically for sorting records [32] along the lines of the SDK radix sort [5] but the focus is to reduce the number of times a record is read from or written into the global memory. As with the other strategies, the input data is divided into tiles and we define the rank of an element as the number of elements having the same digit value before the element in the input data tile. GRS employs a additional memory to store rank of the elements and eliminates the sorting phase proposed in SDK sort [5]. This helps to find the local offset of records within a input data tile and helps to sort them in the shared memory much like SRTS before writing them out to the global memory.
GRS also processes 4 bits per pass and hence has 8 passes in total to sort records with 32-bit keys. The three phases in each pass of GRS are:

**Phase 1:** Compute the histogram for each tile as well as the rank of each record in the tile. In this phase, a block of 64 threads operate on an input tile to cooperatively read the keys from global memory to the shared memory. The global memory reads are made coalesced by ensuring consecutive threads access the consecutive keys in global memory. The writing on the shared memory is performed with an offset to avoid bank conflicts. Threads in read the keys from the shared memory with an offset and calculate the histogram and the rank using the digit counters. The rank overwrites the keys in the shared memory as we don’t need them after their ranks are calculated. Finally, the histogram and the ranks are written out to the global memory. As the rank can not exceed the size of a tile which is typically set to the 1024 records, a full integer is not required to store the rank.

**Phase 2:** The prefix sums of the histograms of all tiles are computed.

**Phase 3:** Lastly, in this phase the entire record is first read from the global memory to the shared memory. We then use the ranks, prefix-summed local histograms to find out the local offset for each record in the tile. We put the records in the shared memory according to the offset so that we get a sorted tile of records. The threads then read the records from shared memory in order and use the global prefix sum to put them in their final place in the output.

Much like SRTS, the final scatter of the records is done in the last phase. As the last phase caters to a very simple implementation, we can efficiently read and write records in this phase. This simplicity enables us to use the algorithms of Figures 3-19 and 3-21 to move records in **ByField** and **ByRecord** layouts respectively. As with SRTS, we only move records 8 number of times during the sorting of records with 32-bit keys. Hence, GRS also has a fair chance of outperforming the first strategy of sorting records by using (key, index) pairs.
3.3 Experimental Results

We implemented and evaluated the record sorting algorithms mentioned in the previous sections using Nvidia CUDA SDK 3.2. Specifically, we evaluated two versions of sample sort, GRS and SRTS each corresponding to the direct and indirect strategies for sorting records mentioned in Section 3.2. We evaluated the following algorithms

1. SampleSort-direct...samplesort algorithm of [27] extended for sorting records.
2. SampleSort-indirect...We form \((key, index)\) pair for each record and then sort them using samplesort. Finally a rearrangement is done to put the entire record in the sorted order.
3. SRTS-direct...SRTS algorithm [31] extended for sorting records.
4. SRTS-indirect... We form \((key, index)\) pair for each record and then sort them using SRTS. Finally a rearrangement is done to put the entire record in the sorted order.
5. GRS-direct...GRS algorithm for sorting records [32].
6. GRS-indirect... We form \((key, index)\) pair for each record and then sort them using GRS. Finally a rearrangement is done to put the entire record in the sorted order.

We implemented the above 6 multifield sorting algorithms on an Nvidia Tesla C1060 which has 240 cores and 4 GB of global memory\(^2\). The algorithms are evaluated using randomly generated input sequences. In our experiments, the number of 32-bit fields per record is varied from 2 to 20 (in addition to the key field) and the number of records was 10 million. Also, the algorithms ate implemented for both ByField and Hybrid layout. For each combination of number of fields and layout type, the time to sort 10 random sequences was obtained. The standard deviation in the observed run times was small and we report only the average times.

\(^2\) We are grateful to the authors of sample sort [27] for making their code available.
3.3.1 Run Times for *ByField* Layout

Figures 3-22 through 3-23 show the comparison of SampleSort, GRS and SRTS using direct and indirect strategies for sorting 10M records with 2 to 9 fields in the *ByField* layout. During each run, we have used the same set of records while comparing these algorithms. SampleSort-indirect runs 36% faster than SampleSort-direct when sorting 10M records with 2 fields while it runs 66% faster for records with 9 fields.

![SampleSort-direct and SampleSort-indirect for 10M records (ByField)](image)

SRTS-indirect runs 37% slower than SRTS-direct sorting 10M records with 2 fields while it runs 27% slower for records with 9 fields.

![SRTS-direct and SRTS-indirect for 10M records (ByField)](image)

Figure 3-22. SampleSort-direct and SampleSort-indirect for 10M records (*ByField*)

Figure 3-23. SRTS-direct and SRTS-indirect for 10M records (*ByField*)
GRS-indirect runs 27% slower than GRS-direct when sorting 10M records with 2 fields while it runs 33% slower for records with 9 fields.

Figure 3-24. GRS-direct and GRS-indirect for 10M records (ByField)

Figure 3-25 shows the comparison between the faster version of each of these three algorithms. SRTS-direct is the fastest algorithm to sort records in the ByField layout when records have between 2 to 11 fields. GRS-direct is the fastest algorithm for sorting records with more than 11 fields. SRTS-direct runs 35% faster than GRS-direct when sorting 10M records with 2 fields while GRS-direct runs 38% faster than SRTS-direct when records have 20 fields. SampleSort-indirect is the slowest, running 63% slower than GRS when sorting records with 2 fields and 48% slower when sorting records with 20 fields.

3.3.2 Run Times for Hybrid Layout

Figures 3-26 through 3-27 show the comparison of SampleSort, GRS and SRTS using the direct and indirect strategies for sorting 10M records with 2 to 9 fields in the Hybrid layout. SampleSort-indirect runs 12% faster than SampleSort-direct when sorting 10M records with 2 fields while it runs 72% faster for records with 9 fields.

SRTS-indirect runs 38% faster than SRTS-direct sorting 10M records with 2 fields while it runs 79% faster for records with 9 fields.
GRS-indirect runs 13% slower than GRS-direct sorting 10M records with 2 fields while it runs 41% faster for records with 9 fields.

Figure 3-29 shows the comparison between the faster version of each of these three algorithms. SRTS-indirect is the fastest algorithm to sort records in the Hybrid layout. SRTS-indirect runs 27% faster than GRS-indirect and 71% faster than SampleSort-indirect when sorting records with 2 fields while it runs 3% faster than GRS-indirect and 16% faster than SampleSort-indirect when sorting records with 20 fields.
3.4 Summary

We have developed a new radix sort algorithm, GRS, to sort numbers and records in a GPU using different layouts. GRS reads and writes records from/to global memory only once per radix sort phase. We have considered two extensions—direct and indirect of the GPU number sort algorithms SampleSort, SRTS, and GRS. We showed how each extension could be implemented optimally on a GPU maximizing device memory and SM bandwidth. Experiments conducted on the NVIDIA C1060 Tesla indicate that, for the ByField layout, GRS-direct is the fastest sort algorithm for 32-bit keys when records have at least 12 fields. When records have fewer than 12 fields, SRTS-direct is the fastest. This happens due to the better memory coalescing achieved by GRS-direct during
the last scatter phase compared to SRTS-direct. In the *Hybrid* layout SRTS-indirect is the fastest algorithm although the performance gap between SRTS-indirect and GRS-indirect narrows once records have more number of fields. At this point, the last global rearrangement phase in both GRS-indirect and SRTS-indirect dominates over other phases of the algorithms and both GRS-indirect and SRTS-indirect have similar run times.

Intuitively, one would expect the indirect method to be faster than the direct method for large records. This is because the indirect method moves each record only once while the direct method moves records many times (O(log n) times on average for sample sort and 8 times for GRS and SRTS). This intuition is borne out in all cases other than when the records are in *ByField* layout and radix sort is used. SRTS-direct and GRS-direct move each field 8 times affording some opportunity for coalescing of device memory accesses. Although SRTS-indirect and GRS-indirect move each field only once, coalescing isn’t possible as the fields of a record in the *ByField* layout are far apart in device memory.
CHAPTER 4
MOTIF SEARCH

4.1 Introduction

Motifs are approximate patterns found in gene sequences. The discovery of motifs helps in finding transcription factor-binding sites, which are useful for understanding various gene functions, drug design, and so on. Many versions of the motif search problem have been studied extensively. We focus on Planted Motif Search (PMS), also known as \((l, d)\) motif search. PMS takes \(n\) sequences and two numbers \(l\) and \(d\) as input. The problem is to find all sequences \(M\) of length \(l\) (also called an \(l\)-mer) that appear in every input sequence with at most \(d\) mismatches. More precisely, for an \(l\)-mer \(M\), we define its \(d\)-neighborhood to be all \(l\)-mers that differ from \(M\) in at most \(d\) positions. \(M\) is a motif for the given set of \(n\) strings iff each of these \(n\) strings has a substring that is in the \(d\)-neighborhood of \(M\).

As an example, consider three input sequences CATACGT, ACAAGTC and AATCGTG. Suppose that \(l = 3\) and \(d = 1\). The 3-mer CAT is one of the motifs of the given 3 input sequences as CAT appears in first sequence at the first position with 0 mismatch, at the second position in the second sequence with 1 mismatch, and at fourth position in third sequence with 1 mismatch. Alternatively, CAT is a motif of the given 3 input sequences because the substrings CAT of sequence 1, CAA of sequence 2, and CGT of sequence 3 are in the 1-neighborhood of CAT.

PMS, which is well studied in the literature, is known to be NP-hard [36]. Known algorithms for PMS are divided into exact and approximation algorithms depending on whether they are guaranteed to produce every motif always or not. Approximation algorithms for PMS are generally faster than exact algorithms, which have an exponential worst-case complexity. MEME, which is one of the popular approximation algorithms for finding motifs [37], uses the expectation minimization technique to output a set of probabilistic models for each motif indicating the probability of appearance of
different characters in each position of the motif. Pevzner and Sze [38] proposed the WINNOWER algorithm, which maps PMS to a problem of finding large cliques in a graph where each node is an \( l \)-mer and two nodes are connected iff the number of mismatches between them is less than \( 2d \). Buhler and Tompa [39] used random projections by taking only \( k \) positions out of the entire \( l \)-mer to group \( l \)-mers together based on the similarity of the projections. Groups that have a large number of \( l \)-mers have a high probability of having the desired motif as well. Price et al. [40] proposed an algorithm based on performing a local search on the \( d \)-neighborhood of some of the \( l \)-mers from input sequences. GibbsDNA [41] employs Gibbs sampling while CONSENSUS [42] uses statistical measures to align sequences and finds potential motifs from the alignment. Two other examples of approximation algorithms for the motif search problem are MULTIPROFILER [43] and ProfileBranching [40].

While exact algorithms for the motif problem take longer to complete than approximation algorithms, they guarantee to find all motifs. Due to their exponential complexity it is impractical to run these algorithms on very large instances. But, for many instances of practical interest, they are able to run within a reasonable amount of time. Many of these algorithms use the suffix tree or other tree data structures to progressively generate motifs one character at a time. MITRA [44] uses a data structure called Mismatch Tree while SPELLER [45], SMILE [46], RISO [47], and RISOTTO [48] use suffix trees. RISOTTO [48] is the fastest suffix tree based algorithm so far. CENSUS [49] makes a trie of all \( l \)-mers from each of the input sequences. The nodes in the trie then store the Hamming distance (i.e., number of mismatches) from the motif as it is being generated, potentially pruning many branches of the trie. Voting [50] uses hashing but needs space to store all possible strings of length \( l \). Hence the space required by this method is too large for large instances. Kauksa and Pavlovic [51] have proposed an algorithm to generate a superset of motifs (i.e., a set of motif stems). Since they do not provide any data on how difficult it may be to extract the true motifs from this superset,
one cannot assess the value of this algorithm. The recently proposed PMS series of algorithms are both fast and relatively economical on space. PMS1, PMS2 and PMS3 [52] are based on radix sorting and then efficiently intersecting the $d$-neighborhood of all $l$-mers in the input sequences (every length $l$ substring of a string is an $l$-mer of that string). PMS4 [53] proposes a generic speed-up technique to improve the run time of any exact algorithm. PMSP [54] computes the $d$-neighborhood of all $l$-mers of the first input sequence and then performs an exhaustive search with the remaining input sequences to determine which of the $l$-mers in the computed $d$-neighborhood are motifs. PMSPPrune [54] is a branch-and-bound algorithm, which uses dynamic programming to improve upon PMSP. Pampa [55] further improves PMSPPrune by using wildcard characters to find approximate motif patterns and then performing an exhaustive search within possible mappings of the pattern to find the actual motifs. PMS5 [56] is the most recent algorithm in the series. This algorithm, which is described in detail in Section 4.2, efficiently computes the intersection of the $d$-neighborhood of $l$-mers without generating the entire neighborhood. PMS5 is faster than the algorithm for Pampa for challenge instances (15, 5) and larger [56]. Although no direct comparison between PMS5 and the stemming algorithm of [51] has been reported in the literature, both have been compared to PMSPPrune [51, 56]. From these comparisons, we infer that stemming is faster than PMS5 for challenge instances of size up to (19, 7) and slower for larger challenge instances. For example, the run time ratio stemming/PMS5 is approximately 0.12 for (13,4) instances and approximately and 0.91 for (19,7) instances. Although run times for stemming have not been reported for larger challenge instances, we can project that PMS5 is faster for larger instances.

We propose a motif algorithm PMS6 [57] that is faster than PMS5. It's relative speed comes from a faster algorithm for $d$-neighborhood generation and intersection. In Section 4.2 we introduce notations and definitions and also describe the PMS5 algorithm in detail and in Section 4.3 we describe our proposed algorithm for $d$-neighborhood
intersection. The performance of PMS5 and PMS6 is compared experimentally in Section 4.4.

4.2 PMS5

We use the same notations and definitions as in [56].

4.2.1 Notations and Definitions

Definition 1. A string \( s = s[1] \cdots s[l] \) of length \( l \) is called an \( l \)-mer.

Definition 2. The Hamming distance, \( d_H(s, t) \), between two strings \( s \) and \( t \) of equal length is defined to be the number of places where they differ.

Definition 3. The \( d \)-neighborhood, \( B_d(s) \), of a string \( s \), is defined to be \( \{ x | d_H(x, s) \leq d \} \).

Note that the number of strings, \( N(l, d) \), in \( B_d(s) \) is uniquely determined by the length \( l \) of \( s \) and \( d \). It is easy to see that \( N(l, d) = |B_d(s)| = \sum_{i=0}^{d} \binom{|\Sigma|}{i} (|\Sigma| - 1)^i \), where \( \Sigma \) is the alphabet from which the characters in our strings are drawn.

Definition 4. The notation \( r \in_l s \) denotes a substring \( r \) of length \( l \) of string \( s \). We define the Hamming distance between a length \( l \) string \( x \) and a string \( s \) of any length \( \geq l \) to be \( \bar{d}_H(x, s) = \min_{r \in_l s} d_H(x, r) \).

Definition 5. Given a set \( S = \{s_1 \cdots s_n\} \) of strings and a string \( x \) of length \( l \), we define \( \bar{d}_H(x, S) = \max_{1 \leq i \leq n} \bar{d}_H(x, s_i) \).

We note that \( x \) is a motif of a set \( S \) of strings if and only if \( \bar{d}_H(x, S) \leq d \). We denote the set of \( (l, d) \) motifs of set \( S \) by \( M_{l, d}(S) \).

4.2.2 PMS5 Overview

PMS5, which is presently the fastest exact algorithm to compute \( M_{l, d}(S) \), was proposed by Dinh, Rajasekaran, and Kundeti [56]. They observed that \( M_{l, d}(S) \) could be determined by computing the intersection of the \( d \)-neighborhood of the \( l \)-mers in one string of \( S \) with that of \( l \)-mers from the remaining strings. That is, if \( S^* = S \setminus \{s_1\} \), then \( M_{l, d}(S) = \bigcup_{x \in S} [B_d(x) \cap M_{l, d}(S^*)] \). For simplicity, assume that \( |S^*| = 2p \) for some integer \( p \) (the case when \( |S^*| \) is odd is a simple extension). We may partition the strings in \( S^* \) into pairs \( S_1 \cdots S_p \) where \( S_k = \{s_{2k}, s_{2k+1}\} \). The following observations
can be made: \( B_d(x) \cap M_{l,d}(S^*) = \cap_{1 \leq k \leq p}[B_d(x) \cap M_{l,d}(S_k)] \) and \( B_d \cap M_{l,d}(S_k) = \cup_{y \in S_{2k}, z \in S_{2k+1}} [B_d(x) \cap B_d(y) \cap B_d(z)] \). Hence, the motifs of \( S \) may be found by making a series of calls to a function that computes the intersection of \( d \)-neighborhoods, \( B_d(x, y, z) = B_d(x) \cap B_d(y) \cap B_d(z) \) as shown in Figure 4-1.

**PMS5**

\[
\text{for each } x \in s_1 \\
\quad \text{for } k = 1 \text{ to } \left\lfloor \frac{n-1}{2} \right\rfloor \\
\quad \quad Q \leftarrow \emptyset \\
\quad \quad \text{for each } y \in s_{2k} \text{ and } z \in s_{2k+1} \\
\quad \quad \quad Q \leftarrow Q \cup B_d(x, y, z) \\
\quad \quad \text{if } k = 1 \\
\quad \quad \quad Q' = Q \\
\quad \quad \text{else} \\
\quad \quad \quad Q' = Q' \cap Q \\
\quad \quad \text{if } |Q'| < \text{threshold} \\
\quad \quad \quad \text{break;} \\
\quad \]  

\[
M_{l,d}(S) \leftarrow \emptyset \\
\text{for each } r \in Q' \\
\quad \text{if } \bar{d}_H(r, S) \leq d \\
\quad \quad \text{Add } r \text{ to } M_{l,d}(S) 
\]

**Figure 4-1. PMS5 [56]**

The time complexity of PMS5 is \( O(nm^3 lN(l, d)) \), where \( m \) is the length of each input string \( s_i \) [56].

### 4.2.3 Computing \( B_d(x, y, z) \)

The basic idea in the algorithm of [56] to compute \( B_d(x, y, z) \) is to generate \( B_d(x) \) one \( l \)-mer at a time and include it in \( B_d(x, y, z) \) only if it is in \( B_d(y) \cap B_d(z) \). To facilitate this, \( B_d(x) \) is represented as a tree \( T_d(x) \). The root, which is at level 0, of \( T_d(x) \) is \( x \). The nodes at the next level represent \( l \)-mers that are at a Hamming distance of 1 from \( x \). Hence, if the length of \( x \) is \( m \) and the alphabet is \( \Sigma = \{0, 1\} \), the root will have \( m \) children and the \( i \)th child will represent the \( l \)-mer \( x' \) that differs from \( x \) only at position \( i \). That is, \( x'[i] = 0 \) if \( x[i] = 1 \) and \( x'[i] = 1 \) if \( x[i] = 0 \). It is easy to see that \( l \)-mers at level
$k$ of $T_d(x)$ have a Hamming distance of $k$ from $x$ and the depth of $T_d(x)$ is $d$. Figure 4-2 shows the tree $T_2(1010)$ with $\Sigma = \{0, 1\}$.

Figure 4-2. 2-Neighborhood tree [56]

The tree $T_d(x)$ is created dynamically in depth first manner. An $l$-mer $t$ in a node $(t, p)$ ($T$ is the $l$-mer represented by the node and $p$ is the position at which this $l$-mer differs from the $l$-mer in the parent node) is added to $B_d(x, y, z)$ if $t$ is in $B_d(y) \cap B_d(z)$. As we know the current distance of $t$ from each of $x$, $y$ and $z$, we can check if there is possibly an $l$-mer in the as yet ungenerated subtree of $(t, p)$ that is a distance $\leq d$ from each of $x$, $y$ and $z$. The subtree rooted at $(t, p)$ is pruned if there is no such $l$-mer. The criterion used in [56] for pruning is as follows. Let $t_1 = t[1] \cdots t[p], t_2 = t[p + 1] \cdots t[l]$, $x_1 = x[1] \cdots x[p]$ and $x_2 = x[p + 1] \cdots x[l]; y_1, y_2, z_1, z_2$ are defined similarly. Due to the way $T_d(x)$ is generated, $x_2 = t_2$. Also, for any descendent $t'$ of $t$, the first $p$ characters are equal. So, $t'$ can be written as $t' = t_1w$ where $w$ is an $(l - p)$-mer. For $t'$ to be in $B_d(y) \cap B_d(z)$, $w$ has to satisfy the following Hamming distance constraints [56].

1. $d_H(x, t') = d_H(x_1, t_1) + d_H(x_2, w) \leq d.$
2. $d_H(y, t') = d_H(y_1, t_1) + d_H(y_2, w) \leq d.$
3. $d_H(z, t') = d_H(z_1, t_1) + d_H(z_2, w) \leq d.$

Note that each position $i$ of $x_2$, $y_2$ and $z_2$ is one of five different types [56].

Type 1(aaa): $x_2[i] = y_2[i] = z_2[i]$.
Type 2(aab): $x_2[i] = y_2[i] \neq z_2[i]$.
Type 3(aba): $x_2[i] = z_2[i] \neq y_2[i]$.
Type 4(abb): $x_2[i] \neq y_2[i] = z_2[i]$.
Type 5(abc): $x_2[i] \neq y_2[i], x_2[i] \neq z_2[i], y_2[i] \neq z_2[i]$.

Let $n_1, n_2, n_3, n_4$ and $n_5$ denote the number of positions of Type 1, 2, 3, 4 and 5, respectively. Then, $n_1 + n_2 + n_3 + n_4 + n_5 = (l - p)$. For an $(l - p)$-mer $w$, the following variables may be defined [56].

1. $N_{1,a}$ is the number of positions $i$ of Type 1 for which $w[i] = x_2[i]$. By definition, $N_{1,a} \leq n_1$.
2. $N_{2,a}(N_{2,b})$ is the number of positions $i$ of type 2 such that $w[i] = x_2[i](w[i] = z_2[i])$. Also, $N_{2,a} + N_{2,b} \leq n_2$.
3. $N_{3,a}(N_{3,b})$ is defined to be the number of positions $i$ of type 3 such that $w[i] = x_2[i](w[i] = y_2[i])$. By definition, $N_{3,a} + N_{3,b} \leq n_3$.
4. $N_{4,a}(N_{4,b})$ is the number of positions $i$ of type 4 for which $w[i] = y_2[i](w[i] = x_2[i])$. By definition, $N_{4,a} + N_{4,b} \leq n_4$.
5. $N_{5,a}(N_{5,b}, N_{5,c})$ is the number of positions $i$ of type 5 such that $w[i] = x_2[i](w[i] = y_2[i], w[i] = z_2[i])$. Also, $N_{5,a} + N_{5,b} + N_{5,c} \leq n_5$.

The number of mismatches between $x_2$ and $w$ in type 1(2,3,4,5) positions is $n_1 - N_{1,a}(n_2 - N_{2,a}, n_3 - N_{3,a}, n_4 - N_{4,b}, n_5 - N_{5,a})$. Hence, $d_H(x_2, w) = n_1 - N_{1,a} + n_2 - N_{2,a} + n_3 - N_{3,a} + n_4 - N_{4,b} + n_5 - N_{5,a}$. Similarly, $d_H(y_2, w) = n_1 - N_{1,a} + n_2 - N_{2,a} + n_3 - N_{3,b} + n_4 - N_{4,a} + n_5 - N_{5,b}$ and $d_H(z_2, w) = n_1 - N_{1,a} + n_2 - N_{2,b} + n_3 - N_{3,a} + n_4 - N_{4,a} + n_5 - N_{5,c}$.

The Hamming distance constraints given above may now be expressed as the following integer linear program (ILP) [56].

1. $n_1 - N_{1,a} + n_2 - N_{2,a} + n_3 - N_{3,a} + n_4 - N_{4,b} + n_5 - N_{5,a} \leq d - d_H(x_1, t_1)$
2. $n_1 - N_{1,a} + n_2 - N_{2,a} + n_3 - N_{3,b} + n_4 - N_{4,a} + n_5 - N_{5,b} \leq d - d_H(y_1, t_1)$
3. $n_1 - N_{1,a} + n_2 - N_{2,b} + n_3 - N_{3,a} + n_4 - N_{4,a} + n_5 - N_{5,c} \leq d - d_H(z_1, t_1)$
4. $N_{1,a} \leq n_1$
5. $N_{2,a} + N_{2,b} \leq n_2$
6. $N_{3,a} + N_{3,b} \leq n_3$
7. $N_{4,a} + N_{4,b} \leq n_4$
8. $N_{5,a} + N_{5,b} + N_{5,c} \leq n_5$
9. All variables are non-negative integers.

When traversing \( T_d(x) \) in depth-first fashion, we may prune the tree at every node \((t, p)\) for which the above ILP has no solution (note that \( d_H(x_1, t_1), d_H(y_1, t_1), d_H(z_1, t_1) \) and \( n_1 \cdots n_5 \) are readily determined for \( x, y, z, t, \) and \( p \)).

As the possible values for \( n_1, \cdots, n_5 \) and \( d_H(x_1, t_1), d_H(y_1, t_1), d_H(z_1, t_1) \) are \([0, \cdots, l]\) and \([0, \cdots, d]\), respectively, only \((l + 1)^5(d + 1)^3\) distinct ILPs are possible. For a given pair \((l, d)\), we may solve these distinct ILPs in a preprocessing step and store, in an 8-dimensional table, whether each has a solution. Once these preprocessing has been done, we can use the results stored in the 8-dimensional table to find motifs for many \((l, d)\) instances.

Figure 4-3 gives the pseudocode for the algorithm to compute \( B_d(x, y, z) \). Its time complexity is \( O(l + d|B_d(x, y, z)|) \) [56].

\[
\begin{align*}
B_d(x, y, z) \\
\text{Compute } d_H(x, y) \text{ and } d_H(x, z). \\
\text{Determine } n_1, n_2, n_3, n_4, n_5 \text{ for each } p = 0, \cdots, (l - 1). \\
\text{Do a depth-first traversal of } T_d(x). \text{ At each node } (t, p) \text{ do the following:} \\
\enspace & \text{Incrementally compute } d_H(x, t), d_H(y, t) \text{ and } \\
\enspace & d_H(z, t) \\
\enspace & \text{Incrementally compute } d_H(x_1, t_1), d_H(y_1, t_1) \text{ and } \\
\enspace & d_H(z_1, t_1) \text{ from its parent.} \\
\enspace & \text{If } d_H(x, t) \leq d, \ d_H(y, t) \leq d, \text{ and } d_H(z, t) \leq d \\
\enspace & B_d(x, y, z) \leftarrow B_d(x, y, z) \cup t. \\
\enspace & \text{Look-up ILP table with parameters } n_1, n_2, n_3, \\
\enspace & n_4, n_5, d_H(x, t), d_H(y, t), d_H(z, t). \\
\enspace & \text{If } d_H(x, t) \leq d \text{ and the ILP has a solution,} \\
\enspace & \quad \text{explore the children of } (t, p); \text{ otherwise prune at } (t, p).
\end{align*}
\]

Figure 4-3. Computing \( B_d(x, y, z) \) [56]

4.2.4 Intersection of \( Q_s \)

PMS5 uses several novel techniques to compute a superset of the intersection \( Q' \) of the \( Q_s \). Although a superset of \( Q' \) (Figure 4-1) is computed, PMS5 determines the exact set of motifs because the last loop of the algorithm (Figure 4-1) verifies that each member of the final \( Q' \) is, in fact, a motif. One of the novel techniques applied for challenge instances of size (19,7) and larger is a Bloom filter [20] with 2 hash functions.
The $l$-mer to be hashed uses $\lceil l/4 \rceil$ bytes (recall that the alphabet size is 4). The first hash function is bytes 0-3 of the $l$-mer and the second is bytes 1-4.

4.3 PMS6

4.3.1 Overview

PMS6 differs from PMS5 only in the way it determines the motifs corresponding to an $l$-mer $x$ of $s_1$ and the strings $s_{2k}$ and $s_{2k+1}$ [57]. Recall that PMS5 does this by computing $B_d(x, y, z)$ independently for every pair $(y, z)$ such that $y \in I_{s_{2k}}$ and $z \in I_{s_{2k+1}}$ (Figure 4-1); the computation of $B_d(x, y, z)$ is done by performing a depth-first search of the tree $T_d(x)$ using an ILP to prune subtrees. In PMS6 we determine the motifs corresponding to an $l$-mer $x$ of $s_1$ and the strings $s_{2k}$ and $s_{2k+1}$ using the following 2-step process:

Form Equivalence Classes: In this step, the triples $(x, y, z)$ of $l$-mers such that $y \in I_{s_{2k}}$ and $z \in I_{s_{2k+1}}$ are partitioned into classes $C(n_1, \cdots, n_5)$. For this partitioning, for each triple $(x, y, z)$, we compute $n_1, \cdots, n_5$ using the definitions of Section 4.2.3 and $p = 0$, $x_1 = y_1 = z_1 = \epsilon$, $x_2 = x$, $y_2 = y$, and $z_2 = z$. Each triple is placed in the class corresponding to its computed $n_1, \cdots, n_5$ values.

Compute $B_d$ for all triples by classes: For each class $C(n_1, \cdots, n_5)$, the union, $B_d(C)$, of $B_d(x, y, z)$ for all triples in that class is computed. We note that the union of all $B_d(C)$s is the set of all motifs of $x$, $s_{2k}$, and $s_{2k+1}$.

Figure 4-4 gives the pseudocode for PMS6.

4.3.2 Computing $B_d(C(n_1, \cdots, n_5))$

Let $(x, y, z)$ be a triple in $C(n_1, \cdots, n_5)$ and let $w$ be an $l$-mer in $B_d(x, y, z)$. Let $p = 0$ and let $N_{1,a}, N_{2,a}, N_{2,b}, N_{3,a}, N_{3,b}, N_{4,a}, N_{4,b}, N_{5,a}, N_{5,b}, N_{5,c}$ be as in Section 4.2.3.

We observe that the 10-tuple $(N_{1,a}, \cdots, N_{5,c})$ satisfies the ILP of Section 4.2.3 with $d_H(x_1, t_1) = d_H(y_1, t_1) = d_H(z_1, t_1) = 0$. In fact, every $l$-mer of $B_d(x, y, z)$ has a 10-tuple $(N_{1,a}, \cdots, N_{5,c})$ that is a solution to this ILP with $d_H(x_1, t_1) = d_H(y_1, t_1) = d_H(z_1, t_1) = 0$. 

105
Given a 10-tuple solution to the ILP, we may generate all $l$-mers $w$ in $B_d(x, y, z)$ as follows:

1. Each of the $l$ positions in $w$ is classified as being of Type 1, 2, 3, 4, or 5 depending on the classification of the corresponding position in the $l$-mers $x$, $y$, and $z$ (see Section 4.2.3).

2. Select $N_{1,a}$ of the $n_1$ Type 1 positions of $w$. If $i$ is a selected position, then, from the definition of a Type 1 position, it follows that $x[i] = y[i] = z[i]$. Also from the definition of $N_{1,a}$, this many Type 1 positions have the same character in $w$ as in $x$, $y$, and $z$. So, for each selected Type 1 position $i$, we set $w[i] = x[i]$. The remaining Type 1 positions of $w$ must have a character different from $x[i]$ (and hence for $y[i]$ and $z[i]$). So, for a 4-character alphabet there are 3 choices for each of the non-selected Type 1 positions of $w$. As, there are $\binom{n}{N_{1,a}}$ ways to select $N_{1,a}$ positions out of $n_1$ positions, we have $3^q \binom{n}{N_{1,a}}$ different ways to populate the $n_1$ Type 1 positions of $w$, where $q = n_1 - N_{1,a}$.

3. Select $N_{2,a}$ positions $I$ and $N_{2,b}$ different positions $J$ from the $n_2$ Type 2 positions of $w$. For each $i \in I$, set $w[i] = x[i]$ and for each $j \in J$, set $w[j] = z[i]$. Each of
the remaining \( n_2 - N_{1,a} - N_{1,b} \) Type 2 positions of \( w \) is set to a character different from that in \( x, y, \) and \( z \). So, if \( k \) is one of these remaining Type 2 positions, \( x[k] = y[k] \neq z[k] \). We set \( w[k] \) to one of the 2 characters of our 4-letter alphabet that are different from \( x[k] \) and \( z[k] \). Hence, we have \( 2^r \binom{n_2}{N_{2,a}} \binom{n_2-N_{2,a}}{N_{2,b}} \) ways to populate the \( n_2 \) Type 2 positions in \( w \), where \( r = n_2 - N_{2,a} - N_{2,b} \).

4. Type 3 and Type 4 positions are populated using a strategy similar to that used for Type 2 positions. The number of ways to populate Type 3 positions is \( 2^s \binom{n_3}{N_{3,a}} \binom{n_3-N_{3,a}}{N_{3,b}} \), where \( s = n_3 - N_{3,a} - N_{3,b} \) and that for Type 4 positions is \( 2^u \binom{n_4}{N_{4,a}} \binom{n_4-N_{4,a}}{N_{4,b}} \), where \( u = n_4 - N_{4,a} - N_{4,b} \).

5. To populate the Type 5 Positions of \( w \), we must select the \( N_{5,a} \) Type 5 positions, \( k \), that will be set to \( x[k] \), the \( N_{5,b} \) Type 5 positions, \( k \), that will be set to \( y[k] \), and the \( N_{5,c} \) Type 5 positions, \( k \), that will be set to \( z[k] \). The remaining \( n_5 - N_{5,a} - N_{5,b} - N_{5,c} \) Type 2 positions, \( k \), of \( w \) are set to the single character of the 4-letter alphabet that differs from \( x[k], y[k], \) and \( z[k] \). We see that the number of ways to populate the \( n_5 \) Type 5 positions is \( \binom{n_5}{N_{5,a}} \binom{n_5-N_{5,a}}{N_{5,b}} \binom{n_5-N_{5,a}-N_{5,b}}{N_{5,c}} \).

The preceding strategy to generate \( B_d(x, y, z) \) generates \( 3^a 2^r 2^s 2^u \binom{n_1}{N_{1,a}} \binom{n_2}{N_{2,a}} \binom{n_3}{N_{3,a}} \binom{n_4}{N_{4,a}} \binom{n_5}{N_{5,a}} \binom{n_5-N_{5,a}}{N_{5,b}} \binom{n_5-N_{5,a}-N_{5,b}}{N_{5,c}} \) \( l \)-mers \( w \) for each 10-tuple \((N_{1,a}, \ldots, N_{5,c})\). While every generated \( l \)-mer is in \( B_d(x, y, z) \), some \( l \)-mers may be the same. Computational efficiency is obtained by computing \( B_d(x, y, z) \) for all \((x, y, z)\) in the same class \( C(n_1, \ldots, n_5) \) concurrently by sharing the loop overheads as the same loops are needed for all \((x, y, z)\) in a class. Figure 4-5 gives the pseudocode for our algorithm to compute \( B_d(x, y, z) \) by classes.

As in the case of PMS5, run-time may be reduced by precomputing data that do not depend on the string set \( S \). So, for a given pair \((l, d)\), there are \( O((l + 1)^5) \) 5-tuples \((n_1, \ldots, n_5)\). For each of the 5-tuples, we can precompute all 10-tuples \((N_{1,a}, \ldots, N_{5,c})\) that are solutions to the ILP of Section 4.2.3 with \( d_H(x_1, t_1) = d_H(y_1, t_1) = d_H(z_1, t_1) = 0 \).

The 10-tuple solutions of the ILP are found using an exhaustive search. For each 10-tuple, we can precompute all combinations (i.e., selections of positions in \( w \)). The precomputed 10-tuple solutions for each 5-tuple are stored in a table with \((l + 1)^5\) entries and indexed by \([n_1, \ldots, n_5]\) and the precomputed combinations for the 10-tuple solutions are stored in a separate table. By storing the combinations in a separate table, we can...
ClassBd$\left(C(n_1, n_2, n_3, n_4, n_5)\right)$

$B_d \leftarrow \emptyset$

Find all ILP solutions with parameters $n_1, n_2, n_3, n_4, n_5$

for each solution $(N_{1,a}, \ldots, N_{5,c})$

\{
    \textit{curComb} \leftarrow \text{first combination for this solution}
    \text{for } i = 0 \text{ to (# combinations)}
    \{
        \text{for each triplet } (x, y, z) \text{ in } C(n_1, \ldots, n_5)
        \{
            \text{Generate } w's \text{ for curComb}.
            \text{Add these } w's \text{ to } B_d.
        \}
        \textit{CurComb} \leftarrow \text{next combination in Gray code order}.
    }\}
\}

\text{return } B_d$

\textbf{Figure 4-5. Computing } B_d(n_1, \ldots, n_5)$

ensure that each is stored only once even though the same combination may be needed
by many 10-tuple solutions.

We store precomputed combinations as vectors. For example, a Type 1 combination
for $n_1 = 3$ and $N_{1,a} = 1$ could be stored as \{010\} indicating that the first and third Type
1 positions of $w$ have a character different from what $x$, $y$, and $z$ have in that position
while the character in the second Type 1 position is the same as in the corresponding
position of $x$, $y$, and $z$. A Type 2 combination for $n_2 = 4$, $N_{2,a} = 2$ and $N_{2,b} = 1$
could be stored as \{3011\} indicating that the character in the first Type 2 position of $w$ comes
from the third \textit{l}-mer, $z$, of the triplet, the second type 2 position of $w$ has a character
that is different from any of the characters in the same position of $x$ and $z$ and the third
and fourth Type 2 positions of $w$ have the same character as in the corresponding
positions of $x$. Combinations for the remaining position types are stored similarly. As
indicated by our pseudocode of Figure 4-5, combinations are considered in Gray code
order so that only two positions in the \textit{l}-mer being generated change from the previously
generated \textit{l}-mer. Consequently, we need less space to store the combinations in the
combination table and less time to generate the new \textit{l}-mer. An example of a sequence
of combinations in Gray code order for Type 2 positions with $n_2 = 4$, $N_{2,a} = 1$, $N_{2,b} = 1$
is \{0012, 0021, 0120, 0102, 0210, 0201, 1200, 1002, 1020, 2010, 2001, 2100\}. Note that in
going from one combination to the next only two positions are swapped.

4.3.3 The Data Structure \(Q\)

We now describe the data structure \(Q\) that is used by PMS6. This is a reasonably
simple data structure that has efficient mechanisms for storing and intersection. In
the PMS6 implementation of [57], there are three arrays in \(Q\); a character array, \(strs[]\),
for storing all \(l\)-mers, an array of pointers, \(bucketPointers[]\), which points to locations in
the character array and a bit array, \(markBuffer[]\), used for intersection. There is also
a parameter \(bucketIndex\) which determines how many characters of \(l\)-mers are used
for indexing into \(bucketPointers[]\) array. As there are 4 possibilities for a character, for
\(p\) characters \(bucketIndex\) can vary from 0 to \((4^p - 1)\). The number of characters, \(p\),
to be used for indexing into \(bucketPointers[]\), is set when \(Q\) is initialized. During the
first iteration of PMS6, for \(k = 1\), \(l\)-mers in \(B_d(C)\) are stored in \(strs[]\). After all \(B_d(C)\)s
are computed, \(strs[]\) is sorted in-place using Most Significant Digit radix sort. After
sorting, duplicate \(l\)-mers are adjacent to each other. Also, \(l\)-mers that have the same
first \(p\) characters and hence are in the same bucket are adjacent to each other as well
in \(strs[]\). By a single scan through \(strs[]\), duplicates are removed and the pointers in
\(bucketPointers[]\) are set to point to different buckets in \(strs[]\). During the remaining
iterations, for \(k \geq 2\), all \(B_d(C)\)s generated are to be intersected with \(Q\). This is done
by using the bit array \(markBuffer\). First, while computing \(B_d(C)\), each \(l\)-mer is searched
for in \(Q\). The search proceeds by first mapping the first \(p\) characters of the \(l\)-mer to
the corresponding bucket and then doing a binary search inside \(strs[]\) within the region
pointed to by the bucket pointer. If the \(l\)-mer is found, its position is set in \(markBuffer[]\).
Once all \(l\)-mers are marked in the \(markBuffer[]\), \(strs[]\) is compacted by removing the
unmarked \(l\)-mers by a single scan through the array. The bucket pointers are also
updated during this scan.
For larger instances, the size of $B_d(C)$ is such that we don’t have sufficient memory to store $B_d(C)$ in $Q$. For these larger instances, in the $k = 1$ iteration, we initialize a Bloom filter using the $l$-mers in $B_d(C)$ rather than storing these $l$-mers in $Q$. During the next iteration ($k = 2$), we store, in $Q$, only those $l$-mers that pass the Bloom filter test (i.e., the Bloom filter’s response is "Maybe"). For the remaining iterations, we do the intersection as for the case of small instances. Using a Bloom filter in this way reduces the number of $l$-mers to be stored in $Q$ at the expense of not doing intersection for the second iteration. Hence at the end of the second iteration, we have a superset of $Q'$ of the set we would have had using the strategy for small instances. Experimentally, it was determined that the Bloom filter strategy improves performance for challenging instances of size (19,7) and larger. As in [56], PMS6 uses a partitioned Bloom filter of total size 1GB. From Bloom filter theory [58] we can determine the number of hash functions to use to minimize the filter error. However, we need to minimize the run time rather than the filter error. Experimentally, [57] determined that the best performance was achieved using two hash functions with the first one being bytes 0-3 of the key and the second being the product of bytes 0-3 and the remaining bytes (byte 4 for (19,7) instances and bytes 4 and 5 for (21,8) and (23,9) instances).

4.3.4 Complexity

The asymptotic time complexity of PMS56 is the same as that of PMS5, $O(nm^3lN(l, d))$, where $m$ is the length of each input string $s[j]$.

4.4 Experimental Results

We evaluate the performance of PMS6 on challenge instances described in [56]. For each $(l, d)$ that characterizes a challenge instance, we generated 20 random strings of length 600 each. Next, a random motif of length $l$ was generated and planted at random positions in each of the 20 strings. The planted motif was then randomly mutated in exactly $d$ randomly chosen positions. For each $(l, d)$ value up to (19,7), we generated 20 instances and for larger $(l, d)$ values, we generated 5 instances. The
average run times for each \((l, d)\) value are reported in this section. Since the variation in run times across instances was rather small, we do not report the standard deviation. Even though we test our algorithm using only synthetic data sets, several authors (e.g., [56]) have shown that PMS codes that work well on the kind of synthetic data used by us also work well on real data. As PMS5 is the fastest algorithm [56] for large-instance motif search, we compare the run times of PMS6 with PMS5. For PMS5, we used C++ code provided by the authors of [56]. PMS6 was coded by us in C++.

The PMS5 and PMS6 codes were compiled using the -03 flag in gcc under 64-bit Linux and run on a Intel Core i7 system running at 3.3 GHz. Our experiments were limited to challenge instances \((l, d)\) [56]. For each challenge instance multiple datasets of 20 randomly generated strings each of length of 600 characters were generated. For each \((l, d)\), the average time is reported here. The standard deviation in the run times is very small.

4.4.1 Preprocessing

The preprocessing times of PMS5 and PMS6 for all challenge instances are given in Figure 4-1. The space required to save the preprocessed data is given in Figure 4-2. The time taken by PMS5 to build its ILP tables for \(l = 23\) and \(d = 9\) is 883 seconds while PMS6 takes 25.5 seconds to build its ILP tables. For the \((23,9)\) case, PMS5 uses roughly 300MB to store its ILP results while PMS6 uses roughly 350 MB to store its ILP solutions and combinations. Although, for large instances, PMS6 needs more space for its ILP tables than does PMS5, the space required by other components of the PMS5 and PMS6 algorithms dominates. For example, the Bloom filter used by both algorithms occupies 1GB and to solve \((23,9)\) instances, we need approximately 0.5GB for Q. At present, run time, not memory, limits our ability to solve larger challenge instances than \((23,9)\) using either PMS5 or PMS6.
Table 4-1. Preprocessing times for PMS5 and PMS6

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(13, 4)</th>
<th>(15, 5)</th>
<th>(17, 6)</th>
<th>(19, 7)</th>
<th>(21, 8)</th>
<th>(23, 9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS5</td>
<td>24.0s</td>
<td>55.0s</td>
<td>119.0s</td>
<td>245.0s</td>
<td>478.0s</td>
<td>883.0s</td>
</tr>
<tr>
<td>PMS6</td>
<td>0.9s</td>
<td>1.0s</td>
<td>2.0s</td>
<td>3.5s</td>
<td>9.5s</td>
<td>25.5s</td>
</tr>
<tr>
<td>PMS5/PMS6</td>
<td>26.67</td>
<td>55.00</td>
<td>59.50</td>
<td>70.00</td>
<td>50.31</td>
<td>34.63</td>
</tr>
</tbody>
</table>

Table 4-2. Total storage for PMS5 and PMS6 in MegaBytes(MB)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(13, 4)</th>
<th>(15, 5)</th>
<th>(17, 6)</th>
<th>(19, 7)</th>
<th>(21, 8)</th>
<th>(23, 9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS5</td>
<td>5.00</td>
<td>15.00</td>
<td>35.00</td>
<td>50.00</td>
<td>155.00</td>
<td>300.00</td>
</tr>
<tr>
<td>PMS6</td>
<td>1.00</td>
<td>4.00</td>
<td>15.00</td>
<td>45.00</td>
<td>145.00</td>
<td>450.00</td>
</tr>
<tr>
<td>PMS5/PMS6</td>
<td>5.00</td>
<td>3.75</td>
<td>2.33</td>
<td>1.11</td>
<td>1.07</td>
<td>0.67</td>
</tr>
</tbody>
</table>

4.4.2 Computing $B_d$

Next, we compare the time taken to compute all $B_d(x, y, z)$ in PMS5 and time taken to compute all $B_d(C(n_1, \cdots, n_5))$ in PMS6 for different challenge instances. The times are given in the Figure 4-3. Note that since PMS5 and PMS6 use different Bloom filters to intersect the $Q$s, the number of iterations needed for $Q'$ to reach the threshold size (i.e., the number of iterations of the for $k$ loop (Figure 4-1)) may be different in PMS5 and PMS6. If PMS6 is run for the same number of iterations as used by PMS5, the PMS6 times to compute the $B_d$s goes up to 8.89m for the (19,7) instances, to 1.29h for the (21,8) instances, and to 10.83h for the (23,9) instances. The PMS5/PMS6 ratios become 2.5, 2.26 and 1.82, respectively, for these instances.

4.4.3 Total Time

The total time (i.e., time to compute the motifs) taken by PMS5 and PMS6 for different challenge instances is shown in Figure 4-4. The run time ratio PMS5/PMS6 ranges from a high of 2.20 for the (21,8) case to a low of 1.69 for the (17,6) case.

Table 4-3. Time taken to compute $B_d(C(n_1, \cdots, n_5))$ and $B_d(x, y, z)$

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(13, 4)</th>
<th>(15, 5)</th>
<th>(17, 6)</th>
<th>(19, 7)</th>
<th>(21, 8)</th>
<th>(23, 9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS5</td>
<td>12.03s</td>
<td>67.64s</td>
<td>6.86m</td>
<td>22.21m</td>
<td>2.91h</td>
<td>19.73h</td>
</tr>
<tr>
<td>PMS6</td>
<td>4.20s</td>
<td>21.67s</td>
<td>2.31m</td>
<td>7.75m</td>
<td>1.09h</td>
<td>8.84h</td>
</tr>
<tr>
<td>PMS5/PMS6</td>
<td>2.86</td>
<td>3.12</td>
<td>2.97</td>
<td>2.87</td>
<td>2.67</td>
<td>2.23</td>
</tr>
</tbody>
</table>
Table 4-4. Total run time of PMS5 and PMS6

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(13,4)</th>
<th>(15,5)</th>
<th>(17,6)</th>
<th>(19,7)</th>
<th>(21,8)</th>
<th>(23,9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS5</td>
<td>39.0s</td>
<td>130.0s</td>
<td>11.35m</td>
<td>40.38m</td>
<td>4.96h</td>
<td>40.99h</td>
</tr>
<tr>
<td>PMS6</td>
<td>22.0s</td>
<td>75.0s</td>
<td>6.72m</td>
<td>22.75m</td>
<td>2.25h</td>
<td>19.19h</td>
</tr>
<tr>
<td>PMS5/PMS6</td>
<td>1.77</td>
<td>1.73</td>
<td>1.69</td>
<td>1.77</td>
<td>2.20</td>
<td>2.14</td>
</tr>
</tbody>
</table>

Table 4-5. Total + preprocessing time by PMS5 and PMS6

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(13,4)</th>
<th>(15,5)</th>
<th>(17,6)</th>
<th>(19,7)</th>
<th>(21,8)</th>
<th>(23,9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS5</td>
<td>63.0s</td>
<td>185.0s</td>
<td>13.33m</td>
<td>44.47m</td>
<td>5.09h</td>
<td>41.23h</td>
</tr>
<tr>
<td>PMS6</td>
<td>22.9s</td>
<td>76.0s</td>
<td>6.75m</td>
<td>22.81m</td>
<td>2.25h</td>
<td>19.19h</td>
</tr>
<tr>
<td>PMS5/PMS6</td>
<td>2.75</td>
<td>2.43</td>
<td>1.98</td>
<td>1.95</td>
<td>2.26</td>
<td>2.15</td>
</tr>
</tbody>
</table>

4.4.4 Total Time Plus Preprocessing Time

In some application scenarios, the preprocessing time may not be amortized over several instances that have the same \((l, d)\). In these environments one would need to use the time taken for both the preprocessing step as well as the time to find the motifs using the preprocessed data as the actual cost of the computation. The total time plus the preprocessing time for PMS5 and PMS6 is shown in Figure 4-5. When preprocessing time is factored in, the run time ratio PMS5/PMS6 declines from a high of 2.75 for the (13,4) case to a low of 1.95 for the (19,7) case.

4.5 PMS6MC

4.5.1 Overview

PMS6MC exploits the parallelism present in the PMS6 algorithm. First, there is outer-level parallelism where the motif search for many \(x\)'s from \(s_1\) can be carried out in parallel (i.e., several iterations of the outer for loop of Figure 4-4 are run in parallel). Second, there is inner-level parallelism where the individual steps of the inner for loop of Figure 4-4 are done in parallel using multiple threads. Outer-level parallelism is limited by the amount of memory available. We have designed PMS6MC to be flexible in terms of its memory and thread requirements. The total number of threads can be set depending on the number of cores and available memory of the system. The threads are grouped into thread blocks. Each thread block operates on a different \(x\) from \(s_1\).
So, for example, if we use a total of \( t \) threads and 4 thread blocks, then our code does 4 iterations of the outer \texttt{for} loop in parallel with each iteration (or thread block) being assigned to \( t/4 \) threads. The threads assigned to a thread block cooperate to find the motifs corresponding to a particular \( x \). The threads use the \texttt{syncthreads()} primitive function to synchronize. This function can be implemented using the thread library synchronization mechanism available under different operating systems. We denote thread block \( i \) as \( T[i] \) while the threads within thread block \( i \) are denoted by \( T[i][j] \).

### 4.5.2 Outer-Level Parallelism

In this, each thread block processes a different \( x \) from \( s_1 \) and calls the function \texttt{findMotifAtThisX()} (Figure 4-6). Once a thread block is done with its assigned \( x \) it moves on to the next \( x \) from \( s_1 \) which is not processed yet. Threads in a thread block execute the function \texttt{findMotifAtThisX()} to find if there is any motif in the \( d \)-neighborhood of \( x \).

\[
\text{PMS6MC}(S, l, d)
\]

for each idle thread block \( T[i] \) in parallel do
\[
\begin{align*}
&\{ \\
&\text{select an } x \in s_1 \text{ that hasn’t yet been selected;} \\
&\text{if there is no such } x \text{ the thread block stops;} \\
&\text{findMotifAtThisX}(x, i); \\
&\}
\]

Figure 4-6. PMS6MC outer level loop

### 4.5.3 Inner-Level Parallelization

Finding motifs in the \( d \)-neighborhood of a particular \( x \) from \( s_1 \) is done by finding the motifs of \( x \) and the strings \( s_{2k} \) and \( s_{2k+1} \) for \( k = 1 \cdots \lfloor \frac{n-1}{2} \rfloor \). As described in Figure 4-4, this is a 4 step process. These steps are done cooperatively by all threads in a thread block. First, we find the equivalence classes for \( x \) and \( l \)-mers from \( s_{2k} \) and \( s_{2k+1} \). For any triple \((x, y, z)\) from an equivalence class we know the number of \( l \)-mers \( w \) which are at a distance \( d \) from \( x, y \) and \( z \) from pre-computed tables. Hence, by multiplying the number of triples with the number of possible \( w \)'s we determine the total number of \( w \)'s for each equivalence class. We denote this number by \( |B_d(C)| \). Next, we compute \( B_d(C) \) for
these equivalence classes in decreasing order of $|B_d(C)|$ in parallel by the threads in the thread block. This order helps in load balancing between different threads as each will be computing $|B_d(C)|$ in parallel. This is akin to using the LPT scheduling rule to minimize finish time. Then we need to store $B_d(C)$s in $Q$ if $k = 1$; i.e., when finding motifs between $x$, $s_2$ and $s_3$. This can be done during the previous step while computing $B_d(C)$. For $k \geq 2$, we need to intersect the set of all $B_d(C)$s with $Q$. When the size of $Q$ falls below a certain threshold, we need to execute the function outputMotifs to find out which $l$-mers in $Q$ are valid motifs. The different steps for findMotifAtThisX() are given in Figure 4-7.

```java
findMotifAtThisX(String x, Thread block i)
for k = 1 to $\lfloor \frac{n-1}{2} \rfloor$
{
    Classes[] ← ∅
    Q ← ∅
    findEquivalenceClasses(x, T[i], Classes[]);
    Sort Classes[] in decreasing order of $|B_d(C)|$;
    for each class C in order from Classes[] in parallel by threads T[i][j];
    {
        ComputeProcess $B_d(C)$;
    }
    syncthreads(T[i]);
    ProcessQ(Q, k, T[i]);
    syncthreads(T[i]);
    if $|Q| < \text{threshold}$ break;
} outputMotifs(Q) in parallel by threads T[i][j];
```

Figure 4-7. Finding motifs in parallel

The data structure used in PMS6MC for $Q$ is very similar to that used in PMS6. However, we use two character arrays strs1[] and strs2[] instead of one. This helps us to perform many operations on $Q$ in parallel by multiple threads. We now describe the different steps used to finding motifs employing this modified structure for $Q$.

4.5.3.1 Finding equivalence classes in parallel

Each thread works on a segment of the string $s_{2k}$ in parallel. For all $y$ that belong to the thread’s assigned segment of $s_{2k}$ and for all $z$ from $s_{2k+1}$, the thread computes the
number of type 1 through type 5 positions for the triple \((x, y, z)\). Based on the number of type 1 through type 5 positions \((n_1, \cdots, n_5)\), the triple is put into the corresponding equivalence class. Once all the threads finish, the equivalence classes formed by different threads need to be merged. As \(n_i\)s for \(i = 1 \cdots 5\) can only vary from 0 to \(l\), the whole \(l^5\) range of \((n_1, \cdots, n_5)\) is divided among the threads in the thread block. Each thread then finds the equivalence classes present in its assigned range and gathers all the triples of these equivalences classes in parallel. The pseudocode for this step is given in Figure 4-8.

```
findEquivalenceClasses(x, i, Classes[])  
for each \(y \in s_{2k}\) and \(z \in s_{2k+1}\) in parallel by threads \(T[i][j]\)  
{  
    Compute \(n_1, \cdots, n_5\) for \((x, y, z)\);  
    if \(C(n_1, \cdots, n_5) \not\in \text{Classes}[j]\)  
    {  
      Create the class \(C(n_1, \cdots, n_5)\) with \((x, y, z)\);  
      Add \(C(n_1, \cdots, n_5)\) to \(\text{Classes}[j]\);  
    }  
    else add \((x, y, z)\) to class \(C(n_1, \cdots, n_5)\);  
}  
syncthreads(T[i]);  
Merge equivalence classes in \(\text{Classes}[]\) by threads \(T[i][j]\) in parallel;  
syncthreads(T[i]);
```

Figure 4-8. Finding equivalence classes in parallel

### 4.5.3.2 Computing \(B_d(C)\) in parallel

Once equivalence classes are formed, we determine \(|B_d(C)|\) by multiplying the number of triples with the number of solutions for equivalence classes using pre-computed tables. Once the number of \(l\)-mers is known, the offset in \(\text{strs1}[]\) to store \(l\)-mers during the first iteration is also known. Hence, each thread can store \(l\)-mers from the designated offset without conflict with other threads. To ensure that each thread in a thread block is roughly doing the same amount of work, we first order the equivalence classes in terms of decreasing \(|B_d(C)|\). This sorting can be done by a single thread as the number of equivalence classes is typically less than 1000 even for
large instances. Thread $j$ of the thread block selects the $j$th equivalence class to work on; when a thread completes, it selects the next available equivalence class to work on. This strategy is akin to the LPT scheduling strategy and is known to provide good load balance in practice. Each thread computes $B_d(C)$ for the class $C$ it is working on using the same strategy as used by PMS6 (see Section 4.3.2).

For $k = 1$, we store the $l$-mers in $strs1[]$ from the designated offset. We also do some additional work which facilitates sorting $Q$ in parallel during the next step. For each thread, we keep track of the number of $l$-mers having the same first character. This is done by maintaing a 2-D counter array $counter[][]$ indexed by thread number and the first character of the $l$-mer.

For $k \geq 2$, the $l$-mer is searched for and marked in the $markBuffer[]$ when found (see Section 4.3.3). Although there might be a write conflict while setting the bit in the $markBuffer[]$, all threads can carry this step in parallel as threads that write to the same mark bit write the same value. Figure 4-9 gives the steps used to compute and process $B_d(C)$.

```
ProcessBd($B_d(C)$, $Q$, $k$, $i$, $j$)
if $k = 1$
  {
    for each $l$-mer $w \in B_d(C)$ $counter[j][w[0]] +=$;
    Copy $l$-mers in $B_d(C)$ to $strs1[]$ from the offset for this class;
  }
if $k \geq 2$
  {
    For all $l$-mer $w \in B_d(C)$, set $markBuffer[]$ if $w$ is present in $Q$;
  }
```

Figure 4-9. Compute and process $B_d(C)$

4.5.3.3 Processing $Q$ in parallel

The processing of $Q$ depends on the iteration number. When $k = 1$, we sort $strs1[]$, then remove the duplicates and set the bucket pointers. For the remaining iterations, we need to remove all unmarked $l$-mers from $strs1[]$ and update the bucket pointers.
The $k = 1$ sort is done by first computing the prefix sum of counters so that the counter for a particular thread and a particular character equals the total number of $l$-mers processed that have either smaller first character or equal first character but processed by threads with a smaller index. Since the number of counters is small (256 different counters for 8-bit characters) we compute the prefix sums using a single thread. The pseudocode is given in Figure 4-10. Next, each thread in the thread block scans through the $l$-mers in $strs_1[]$ that it had stored while generating $B_d(C)$. Depending on the first character of the $l$-mer, the $l$-mer is moved to $strs_2[]$ starting from the offset indicated by prefix sum counter. This movement of $l$-mers is done by the threads in a thread block in parallel. Once the movement is complete, $strs_2[]$ is divided into segments such that the first characters of all $l$-mers within a segment are the same. Following this segmenting, the threads sort the segments of $strs_2[]$ in parallel using radix sort. Each thread works on a different segment. Since the first character of the $l$-mers in a segment are the same, the radix sort starts from the second character. Once the segments are sorted, we proceed to eliminate duplicates and set the bucket pointers. First the threads count the number of unique $l$-mers in each segment in parallel by checking adjacent $l$-mers. Again, each thread works on a different segment. The determined counts of unique $l$-mers are prefixed summed by a single thread to get the offsets required for moving the unique $l$-mers to their final positions. Using these offsets, the threads move unique $l$-mers with each thread moving the unique $l$-mers of a different segment from $strs_2[]$ to $strs_1[]$ in parallel. While moving an $l$-mer, the threads also check to see if first $p$ characters of the current $l$-mer are the same as those of the previous $l$-mer; if not, the appropriate pointer in bucketPointers[] is set.

When $k \geq 2$, we need to remove from $Q$ all the $l$-mers that are not marked in markBuffer. This is done in two steps. First, the markBuffer is divided into segments and each thread does a prefix sum on different segments in parallel. This gives the number of marked $l$-mers in each segment. Next, we move the marked $l$-mers in each
PrefixCounters(counters[])  
sum = 0;  
for i = 0 to 255 // There are 256 possibilities  
        // for the first character  
{  
    counters[0][i] = counters[0][i] + sum;  
    for j = 1 to threads  
    {  
        counters[j][i] = counters[j][i] + counters[j][i - 1];  
    }  
    sum = counters[threads - 1][i];  
}  

Figure 4-10. Prefix sum of counters

segment from strs1[] to strs2[]. For this, a prefix sum is performed by a single thread on the counters having the number of marked l-mers in different segments to get the offset in strs2[] for moving the l-mers. With these offsets, the threads then move the marked l-mers from different segments in parallel. As before, when moving an l-mer, the thread checks to see if the first p characters of the current l-mer differs from the previous one and update the appropriate pointer in bucketPointers. There might be a problem in updating the bucket pointers in the boundary region of the segments as one bucket can extend across the boundary of two segments and hence two threads might update that bucket pointer. These boundary bucket pointers are fixed by a single thread after all l-mers are moved. Note that there can only be as many boundary buckets as there are segments which are very few in number. The pseudocode for the processing Q for different values of k is given in Figure 4-11.

4.5.3.4 The outputMotifs routine

Once the size of Q drops below a certain threshold, we break out of the loop and call outputMotifs(Q) to determine the set of valid motifs in Q. This step can be done in parallel as checking the validity of l-mers to be motifs can be done independent of one another. So, each thread examines a disjoint set of l-mers from Q exhaustively checking if it is a motif as is done in PMS6; the threads operate in parallel.
Figure 4-11. Processing $Q$

### 4.6 Experimental Results

We evaluated the performance of PMS6MC on the challenging instances described in [56]. For each $(l, d)$ that characterizes a challenging instance, we generated 20 random strings of length 600 each. Next, a random motif of length $l$ was generated and planted at random positions in each of the 20 strings. The planted motif was then randomly mutated in exactly $d$ randomly chosen positions. For each $(l, d)$ value up to $(19,7)$, we generated 20 instances and for larger $(l, d)$ values, we generated 5 instances. The average run times for each $(l, d)$ value are reported in this section. Since the variation in run times across instances was rather small, we do not report the standard deviation. Even though we test our algorithm using only synthetic data sets, several
authors (e.g., [56]) have shown that PMS codes that work well on the kind of synthetic data used by us also work well on real data.

### 4.6.1 PMS6MC Implementation

PMS6MC is implemented using the pthreads (POSIX Threads) library under Linux on an Intel 6-core system with each core running at 3.3GHz. We experimented with different degrees of outer-level (number of thread blocks) and inner-level (number of threads in a thread block) parallelism for different challenging instances. For smaller instances (e.g. (13,4) and (15,5)), the performance is limited by the memory bandwidth of the system. Hence, increasing the degree of inner or outer level parallelism does not have much effect on the run time as most of the threads stall for memory access. For larger instances, the number of thread blocks is limited by the available memory of the system. Figure 4-6 gives the number of thread blocks and the number of threads in a thread block for different challenging instances which produces the optimum performance.

**Table 4-6. Degree of inner and outer level parallelism for PMS6MC**

<table>
<thead>
<tr>
<th>Challenging Instances</th>
<th>Outer-level Blocks</th>
<th>Threads per Block</th>
<th>Total Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>(13,4)</td>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>(15,5)</td>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>(17,6)</td>
<td>8</td>
<td>6</td>
<td>48</td>
</tr>
<tr>
<td>(19,7)</td>
<td>4</td>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>(21,8)</td>
<td>2</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>(23,9)</td>
<td>2</td>
<td>12</td>
<td>24</td>
</tr>
</tbody>
</table>

### 4.6.2 PMS6 and PMS6MC

We compare the run times of PMS6 and PMS6MC on an Intel 6-core system with each core running at 3.3GHz. PMS6 takes 22 seconds on an average to solve (15,5) instances and 19 hours on an average to solve (23,9) instances. PMS6MC, on the other hand, takes 8 seconds on an average to solve (15,5) instances and 3.5 hours on an average to solve (23,9) instances. The speedup achieved by PMS6MC over PMS6 varies from a low of 2.75 for (13,4) instances to a high of 6.62 for (21,8) instances. For
(19,7) and larger instances PMS6MC achieves a speedup of over 5. The run times for various challenging instances are given in Figure 4-7.

Table 4-7. Run times for PMS6 and PMS6MC

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(13,4)</th>
<th>(15,5)</th>
<th>(17,6)</th>
<th>(19,7)</th>
<th>(21,8)</th>
<th>(23,9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS6</td>
<td>22.0s</td>
<td>74.0s</td>
<td>6.82m</td>
<td>22.75m</td>
<td>2.25h</td>
<td>19.19h</td>
</tr>
<tr>
<td>PMS6MC</td>
<td>8.0s</td>
<td>21.0s</td>
<td>1.03m</td>
<td>4.45m</td>
<td>25.5m</td>
<td>3.57h</td>
</tr>
<tr>
<td>PMS6/PMS6MC</td>
<td>2.75</td>
<td>3.52</td>
<td>6.62</td>
<td>5.11</td>
<td>5.29</td>
<td>5.38</td>
</tr>
</tbody>
</table>

4.6.3 PMS6MC and Other Parallel Algorithms

Dasari, Desh and Zubair proposed a voting based parallel algorithm for multi-core architectures using bit arrays [59]. They followed up with an improved algorithm based on suffix trees for GPUs and multi-core CPUs from intel [60]. We estimate the relative performance of PMS6MC and these parallel algorithms using published run times and performance ratios. Figure 4-8 and the first 4 rows of Figure 4-9 give the performance of PMS5 and PMSPrune as reported in [56] and that of PMS6 and PMS5 as reported in [57], respectively.

Table 4-8. Run times for PMS5 and PMSPrune [56]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(13,4)</th>
<th>(15,5)</th>
<th>(17,6)</th>
<th>(19,7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS5</td>
<td>117.0s</td>
<td>4.8m</td>
<td>21.7m</td>
<td>1.7h</td>
</tr>
<tr>
<td>PMSPrune</td>
<td>45.0s</td>
<td>10.2m</td>
<td>78.7m</td>
<td>15.2h</td>
</tr>
<tr>
<td>PMS5/PMSPrune</td>
<td>2.60</td>
<td>0.47</td>
<td>0.28</td>
<td>0.11</td>
</tr>
</tbody>
</table>

We divide the ratio \( PMS5/PMSPrune \) by \( PMS5/PMS6 \) to estimate the ratio \( PMS6/PMSPrune \) (5th row of Figure 4-9). Next, we divide the ratio \( PMS6/PMS6MC \) (row 4 of Figure 4-7) by our estimate of \( PMS6/PMSPrune \) to get an estimate of \( PMSPrune/PMS6MC \) (6th row of Figure 4-9).

The first 4 rows of Figure 4-10 give the run times of gSPELLER-x, mSPELLER-x, and PMSPrune as reported in [60]. The "x" indicates the number of CPU cores for mSPELLER and the number of GPU devices in the case of gSPELLER. We report the times for mSPELLER-16 and gSPELLER-4 as these were the fastest reported in [60]. From this data and that of Figure 4-9 we can estimate the speedups shown in
Table 4-9. Total run time of different PMS Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(13,4)</th>
<th>(15,5)</th>
<th>(17,6)</th>
<th>(19,7)</th>
<th>(21,8)</th>
<th>(23,9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS5</td>
<td>39.0s</td>
<td>130.0s</td>
<td>11.35m</td>
<td>40.38m</td>
<td>4.96h</td>
<td>40.99h</td>
</tr>
<tr>
<td>PMS6</td>
<td>22.0s</td>
<td>75.0s</td>
<td>6.72m</td>
<td>22.75m</td>
<td>2.25h</td>
<td>19.19h</td>
</tr>
<tr>
<td>PMS5/PMS6</td>
<td>1.77</td>
<td>1.73</td>
<td>1.69</td>
<td>1.77</td>
<td>2.20</td>
<td>2.14</td>
</tr>
<tr>
<td>PMS6/PMSPrune</td>
<td>1.46</td>
<td>0.27</td>
<td>0.17</td>
<td>0.06</td>
<td>-.</td>
<td>-.</td>
</tr>
<tr>
<td>PMSPrune/PMS6MC</td>
<td>1.88</td>
<td>13.04</td>
<td>38.94</td>
<td>85.17</td>
<td>-.</td>
<td>-.</td>
</tr>
</tbody>
</table>

rows 5 through 8 of Figure 4-9. We estimate that the speed up of PMS6MC using 6 cores compared to mSPELLER-16 using 16 cores varies from a low of 0.07 for (13,4) instances to 3.58 for (19,7) instances while the speed up for PMS6MC using only one CPU with respect to gSPELLER-4 using 4 GPUs varies from a low of 0.03 for (13,4) instances to a high of 1.97 for (19,7) instances.

Table 4-10. Comparing mSPELLER and gSPELLER with PMS6MC

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(13,4)</th>
<th>(15,5)</th>
<th>(17,6)</th>
<th>(19,7)</th>
<th>(21,8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMSPrune</td>
<td>53.0s</td>
<td>9.0m</td>
<td>69.0m</td>
<td>9.2h</td>
<td>-.</td>
</tr>
<tr>
<td>mSPELLER-16</td>
<td>2.0s</td>
<td>16.5s</td>
<td>2.5m</td>
<td>23.6m</td>
<td>3.7h</td>
</tr>
<tr>
<td>gSPELLER-4</td>
<td>0.8s</td>
<td>7.2s</td>
<td>1.2m</td>
<td>13.0m</td>
<td>2.2h</td>
</tr>
<tr>
<td>PMSPrune/mSPELLER-16</td>
<td>26.50</td>
<td>32.73</td>
<td>27.60</td>
<td>23.38</td>
<td>-.</td>
</tr>
<tr>
<td>PMSPrune/gSPELLER-4</td>
<td>66.25</td>
<td>75.00</td>
<td>57.50</td>
<td>42.46</td>
<td>-.</td>
</tr>
<tr>
<td>mSPELLER-16/PMS6MC</td>
<td>0.07</td>
<td>0.40</td>
<td>1.41</td>
<td>3.64</td>
<td>-.</td>
</tr>
<tr>
<td>gSPELLER-4/PMS6MC</td>
<td>0.03</td>
<td>0.17</td>
<td>0.68</td>
<td>2.00</td>
<td>-.</td>
</tr>
</tbody>
</table>

4.7 Summary

We have developed a new algorithm, PMS6, for the motif discovery problem. The run time ratio PMS5/PMS6 ranges from a high of 2.20 for the (21,8) challenge instances to a low of 1.69 for the (17,6) challenge instances. Both PMS5 and PMS6 require some amount of preprocessing. The preprocessing time for PMS6 is 34 times faster than that for PMS5 for (23,9) instances. When preprocessing time is factored in, the run time ratio PMS5/PMS6 is as high as 2.75 for (13,4) instances and as low as 1.95 for (17,6) instances.

We have further developed a multicore version of PMS6 that achieves a speedup that ranges from a low of 2.75 for (13,4) challenging instances to a high of 6.62 for
(17,6) challenging instances on a 6-core CPU. Our multicore algorithm is able to solve (23,9) challenging instances in 3.5 hours while the single core PMS6 algorithm takes 19 hours. We estimate that our multicore algorithm is faster than other parallel algorithms for the motif search problem on large challenging instances. For example, we estimate that PMS6MC can solve (19,7) instances 3.6 times faster than using our 6-core CPU mSPELLER-16 using the 16-core CPU of [60] and about 2 times faster than gSPELLER-4 using 4 GPU devices.
In this dissertation, we have developed algorithms for sorting and motif search on different parallel architectures. Specifically, we have developed a parallel merge sort algorithm for Cell Broadband Engine Architecture which is the fastest way to sort numbers stored in the local store of an SPE. We then extended this algorithm to sort large records on Cell Broadband Engine. Next, we developed a fast radix sort based algorithm, GRS, for sorting large records on GPUs. We have experimented with different memory layouts for storing records and have shown GRS to be a fast algorithm for sorting records. We then developed a fast sequential algorithm, PMS6 for motif search and implemented it on an intel CPU. We have shown that PMS6 is twice as fast, on average, compared to the fastest sequential algorithm for motif search. We then developed a parallel version of PMS6, PMS6MC, for intel multi-core CPUs. On a 6-core intel core i7 CPU, it achieves a speed up of as high as 6 for some challenging instances of motif search.

PMS6MC can be extended to other multi-core systems. However, the granularity of parallelism should be varied according to the system on which it is being implemented. The natural extension of PMS6MC would be to implement it on GPUs. There are certain differences between CPUs and GPUs that makes just porting the current code to GPU much less optimal. On the CPUs, it is better to run a few threads with each thread doing multiple data elements at a time. As opposed to CPUs, it is essential to spawn thousands of threads in GPU to hide memory latency and run the GPU at its maximum performance. As an example of applying these principles, lets consider the step when $B_d(C)$ for all equivalence classes are generated in parallel by threads. In the CPU implementation, it was enough that each thread processes different classes in parallel but in case of GPUs it is not a good design. Firstly, there is not enough classes to exploit full parallelism of the GPU. Secondly, each thread processing a different class will lead
to divergence. Hence, in case of GPU implementation, each thread in a thread block can work with different triplets \((x, y, z)\) from equivalence class \(C\) as there will be a little divergence on the processing of triplets from the same class. This also increases the number of threads that can be launched on the GPUs. Each thread block can process different classes. However, this approach increases the register and shared memory pressure as each thread needs to keep informations related to that equivalence class. We intend to analyze the various trade-offs in designing GPU kernels by increasing the level of parallelism with efficient memory access.
REFERENCES


BIOGRAPHICAL SKETCH

Shibdas Bandyopadhyay completed his Ph.D. in computer engineering at the University of Florida in the summer of 2012. His research focus is to develop parallel algorithms for many-core architectures. Before starting his Ph.D., he has completed his master's in computer science from Indian Statistical Institute and his bachelor's in information technology from Indian Institute of Information Technology, Kolkata.