Final thesis

Optimized On-chip Software Pipelining On the Cell BE Processor

by

Rikard Hultén

LIU-IDA/LITH-EX-A—10/015—SE

2010-03-12
Examensarbete

Optimized On-chip Software Pipelining On the Cell BE Processor

av

Rikard Hultén

LIU-IDA/LITH-EX-A—10/015—SE

2010-03-12

Handledare: Christoph Kessler
Examinator: Christoph Kessler
Abstract

The special architecture of the Cell BE processor has made scientists revisit the problem of sorting. This paper implements and tests a variant of merge sort where a number of 2-to-1 mergers are connected in a pipelined tree. For large trees there are many more such mergers than processors which means they must be mapped to the processors in some way. Optimized mappings are tested and results show that changing the model used when optimizing might be beneficial. It is also shown that the small size of the local storages on the co-processors is not limiting the performance.
Acknowledgments

I would like to thank supervisor and examiner Christoph Kessler, LiU, for his time. Niklas Dahl, IBM, for letting me use their machine. Mattias Eriksson, LiU, for helping me with the PS3. I would also like to thank my close friend Emil Boström for tips on C programming and for allways picking my spirits up when needed.
## Contents

1 Introduction 1  
  1.1 Intended Audience 1  
  1.2 Purpose and Methodology 1  
  1.3 Parallelism 1  
  1.4 Cell 2  
  1.5 Sorting 2  

2 Background 3  
  2.1 Merge Sort 3  
  2.2 SIMD 3  
  2.3 Bitonic Merge Sort 3  
  2.4 Pipelining 5  
  2.5 Circular Buffer 6  
  2.6 Double- and Multi- Buffering 6  

3 The Cell BE Processor 7  
  3.1 Overview 7  
  3.2 Local Store 7  
  3.3 Element Interconnect Bus 8  
  3.4 Memory Flow Controller 8  
  3.5 Vector 9  
  3.6 Programming 9  

4 Prior Work 11  
  4.1 CellSort 11  
  4.2 Optimized Mappings 11  
    4.2.1 DC-Map 12  

5 Problem Statement 15  

6 Implementation 17  
  6.1 The Merger Node 17  
  6.2 The PPE 17  
  6.3 Mapping Nodes to SPEs 17  
  6.4 Communication 18
6.4.1 Synchronization ................................................. 18
6.4.2 Tag Administration ............................................. 18
6.5 Memory ............................................................... 19
6.6 Sorting Kernel ....................................................... 20

7 Results ................................................................. 21
7.1 Baseline .............................................................. 21
7.2 Setup ................................................................. 21
7.3 Machine Configuration ............................................. 24
  7.3.1 PS3 ............................................................ 24
  7.3.2 QS-20 ........................................................ 24
7.4 Results of Different Mappings .................................... 24
  7.4.1 PS3 ............................................................ 24
  7.4.2 QS-20 ........................................................ 24
7.5 Results of Different Buffer Sizes ............................... 27
  7.5.1 QS-20 ........................................................ 27
7.6 Results of DC Map .................................................. 27
  7.6.1 QS-20 ........................................................ 27
7.7 Compared to CellSort .............................................. 27

8 Discussion ............................................................ 33
8.1 Mappings ............................................................ 33
8.2 Buffer Sizes ......................................................... 33
8.3 Conclusion .......................................................... 33
8.4 Future Work ........................................................ 34

Bibliography ............................................................ 35
Chapter 1

Introduction

1.1 Intended Audience

The reader is assumed to have a basic knowledge of computer architecture and programming.

1.2 Purpose and Methodology

The purpose of this work is to investigate the impact of using optimized on-chip software pipelining of merge sort on the Cell BE processor. This is done by implementing an algorithm on the Cell processor, that can run different optimized pipelines and sorting data of different sizes. The time consumed for sorting is measured and analyzed.

1.3 Parallelism

In recent years clock frequencies have stagnated partly because cooling the processor would be too big of a problem. With higher clock frequency comes higher power consumption, all of which becomes heat. Trying to make the processor bigger or 'smarter' is very hard because of complexity and does not provide much performance gains. Another problem with making the processor ever faster is that the memory does not keep up. There is little use of a fast processor with no data. These problems are sometimes referred to as the power-, ILP- and memory-walls.

The only viable way to provide more performance is to go parallel. This is the direction the industry is going, at time of writing this no new computer (with the exception of small laptops) ships with less than two processors. The problem is that writing programs that benefit from multiple CPUs is much more difficult. In the past programs became faster because the processors were getting faster. Just waiting 18 months could make it twice as fast, because the clock frequency would double in that time. Today, for a program to become twice as fast, it must be able
Parallel computing is not new to the high performance computing, HPC, society. For ordinary PC programmers though it will be a great challenge to take advantage of the new many-core processors the coming years.

1.4 Cell

Sony, Toshiba and IBM joined forces and developed the Cell Broadband Engine in 2001. The Cell processor differs from the regular x86 processor found in PCs in a number of ways. Firstly, all cores are not of the same type, there is a main core and a number of co-processors. The co-processors are optimized for data parallelism, meaning code must be carefully written to take advantage. Secondly each co-processor core has its own memory that must be explicitly managed. Because of this, programming the Cell posts quite a challenge. The most well known Cell system must be the Sony Playstation 3, PS3. Other systems include the IBM QS-20 and the supercomputer Roadrunner, at the time of writing the 2nd fastest supercomputer in the world, it uses no less then 12,960 Cell processors.

1.5 Sorting

Sorting is fundamental to computer science with countless applications in everything from databases to graphics. Because of this much work has been put into sorting and there exist many well known algorithms. In this work sorting is also interesting because it has a high communication to computation ratio, meaning that it is ill suited for computers with limited memory bandwidth between main memory and CPU. Of course most processors have this problem, but the aggregated demand for memory bandwidth on the Cell, with its many cores, is even higher.
Chapter 2

Background

2.1 Merge Sort

Merge sort is a divide-and-conquer algorithm where sorted lists of increasing length are merged to obtain a combined sorted list. It starts with lists of one element, which are trivially in sorted order. Two such lists are then merged into a list of two sorted elements. Two such lists are merged into a list of four elements, and so on. The procedure of merge sort for 10 elements is illustrated in figure 2.1.

Unlike other algorithms merge sort does not sort the elements in place, meaning that it moves the elements and places them in sorted order somewhere else. However, a big advantage is that elements are accessed in a simple serial way that makes it useful when sorting very large data sets that do not fit into working memory. Chunks of data can be brought in memory and merged and then written back.

For further explanation about correctness, complexity and implementation see e.g. Introduction to Algorithms [2].

2.2 SIMD

SIMD operations, Same Instruction Multiple Data, exposes data parallelism by applying the same instruction on multiple data at the same time. The instructions are still used sequentially but data manipulation is parallel. Figure 2.2 shows a SIMD operation adding multiple integers at the same time. SIMD operations require special hardware with wide registers that can hold the multiple data and special arithmetic hardware that can operate on them.

2.3 Bitonic Merge Sort

Bitonic merge sort works like the regular merge sort but utilizes properties of bitonic sequences. In short, the lists to be merged at any step are sorted in alternating order. The key feature of bitonic merge sort, and the reason it is
**Figure 2.1.** Illustration of merge sort on 10 elements.

**Figure 2.2.** Example of a SIMD add operation.
interesting in this work, is that the sequence of comparisons are data independent. This allows for very efficient implementation in hardware or in this case SIMD operations.

A bitonic sequence is a sequence of number that is twofold monotonic, first zero or more monotonically increasing numbers and then zero or more monotonically decreasing numbers. For example \{0, 0, 1, 2, 3, 2\} or \{6, 5, 4, 5, 8\} are both bitonic.

Such sequences can be sorted in a sorting network, as put forward in the paper by K.E. Batcher [1]. A sorting network is a network of comparators connected by wires. A comparator takes two input values on its input wires and outputs the lower value to one of the output wires and the higher value on the other output wire. A network of comparators that sorts its input values is called a sorting network. Figure 2.3 shows a sorting network for 8 input values, where the vertical lines represent comparators. Note that independent comparisons are made in parallel so that the network in figure 2.3 has a depth of 6.

2.4 Pipelining

Sometimes an algorithm can be divided into stages that can be performed in a sequence on data. Ideally each stage can work on part of the data at a time and then send it to the next stage, so that data is streamed through the pipeline. This is called pipelining and is a useful technique in parallel computing because all stages can be run in parallel. The assembly line of a car factory is a good analogy. In order to improve utilization of resources and throughput the manufacturing is broken up in steps such that each station can do useful work most, if not all, of the time.
2.5 Circular Buffer

A buffer is a memory area used to hold data so that communication can be done in chunks instead of per element, reducing overhead of transfer initializations. A circular buffer is a buffer that is written to and read from in a circular fashion, it wraps around at the ends. This is useful when a producer of data and a consumer of data shares a buffer for communication. The next free slot in the buffer is called the head, and the last occupied slot is called the tail.

The producer puts data into the buffer at the head as long as there is an empty slot, and the consumer pulls data out of the buffer at the tail as long as there is something in it. If the buffer is full the producer waits and if the buffer is empty the consumer waits.

2.6 Double- and Multi- Buffering

Double buffering is a technique where two buffers are used, one for computation and writing whilst the other one is read from. A use of this is to mask data transfer latency: while one buffer is being used for computation the other one can simultaneously be transferred to or from somewhere. If the time it takes to transfer a buffer out and put new data in is less than the time used for computation the transfer time is completely masked out.

Multi buffering extends this concept to an arbitrary number of buffers ordered in a circular FIFO queue. This can be useful if multiple transfers can be in flight at the same time.
Chapter 3
The Cell BE Processor

3.1 Overview

Figure 3.1 shows the architecture of the Cell processor having two different types of cores, one main processing unit and eight smaller cores. On the PS3 only seven are activated and six of them are accessible when running Linux, on other systems like the QS-20 all eight are accessible (the QS-20 actually has two Cell processors for a total of 16 SPEs). The main core is a slightly modified PowerPC processor called the PowerPC Processing Element, PPE. The eight smaller cores are identical and called Synergistic Processing Elements, SPE. The SPEs are optimized for vector processing of single precision numbers, and less suited for general purpose program execution. Programs therefore typically run on the PPE and then branch out compute intensive regions to the SPEs.

Each SPE has 256kB of memory called its local store which must house all program code and data that the SPE is to use because it can not directly access main memory. Data must be transferred on and off the local stores explicitly by the programmer using the SPEs communication units called the memory flow controllers. The PPE on the other hand can access main memory in an ordinary fashion.

3.2 Local Store

Each SPE has a bank of on-chip memory called the local store. This is all the memory the SPE sees and is able to work on. Both execution code and data must thus fit onto the 256kB large area. In order to do something useful the SPE must transfer data from main memory onto its local store, do computation on it and then transfer it back to main memory. This is done through asynchronous Direct Memory Access, DMA operations.
3.3 Element Interconnect Bus

The SPEs communicate through the Element Interconnect Bus, EIB, which mainly consists of four unidirectional rings, two in each direction. The bandwidth between participants in the ring is theoretically 25.6GB/s, more importantly the maximum bandwidth to main memory is 25.6GB/s to be compared to the aggregate on-chip bus bandwidth of 204.8GB/s. This means data should be brought on and off the chip with care in order to not congest the bus to main memory. If possible data should be brought on-chip, completely processed and then written back to main memory [8].

3.4 Memory Flow Controller

Each SPE has a memory flow controller, MFC, that handles memory transfers between the local store and main memory or another SPE. It works alongside the rest of the SPE, so the SPE can transfer data and work on some other data in its local store at the same time. Double- or multi-buffering makes this possible and can give great performance gains, especially if time computing is about the same as time transferring.

The MFC can keep track of 32 groups of transfers, when initializing a transfer one must choose a group id for it, called a tag. The MFC can then be queried about the status of any group, mainly if it is completed or not.
3.5 Vector

A vector is a 128 bit block of memory and is the basic unit of arithmetics granularity for the SPEs. It can store 4 32b integers, 16 16b short integers, 2 64b doubles and so forth. The SPE only works on these vectors, while the PPE has a larger instruction set that can handle both vectors and the regular scalar data types. When working on scalars the SPE must first put them in vectors, do the operation and then put the scalar back. This makes operations on single scalars slower than if vectors are used. A scalar also occupies a full vector of memory, even if the data type only needs a part of it. Non-cohesive data may be packed in the same vector to save memory, but then operating on the individual elements becomes trickier. Altogether scalars are to be avoided for both size and speed.

3.6 Programming

When programming the Cell programs are written for the PPE and the SPEs look and feel like Linux threads to the PPE. The code for the thread function is copied to the local store of the SPE before it is started. Only one thread can run on a SPE at a time and is not interrupted which means the thread function runs to completion. For a comprehensive study of Cell programming see Scarpinos book 'Programming the Cell Processor: For Games, Graphics, and Computation' [9], Kesslers chapter 'Programming the Cell Processor' in [7] and IBM’s programming guide [8] available online.
Chapter 4

Prior Work

4.1 CellSort

CellSort [3] uses bitonic sort as its sorting kernel because it lends itself to SIMD operations. It works in three phases. First as much data as can fit onto each SPE is sorted locally, on all participating SPEs in parallel. In the second phase, the SPEs collaborate to sort the data on multiple SPEs. Lastly, if the amount of data can not fit onto the aggregate local storages of the SPEs, it does an out-of-core sort. It only works on $2^n$ SPE’s which limits it to 4 SPEs on the PS3 and 16 on QS-20. Each SPE also only works on $2^n$ elements which limits the local storage utilization to 128KiB plus code out of the available 256KiB. During the out-of-core phase it moves data on and off the processor a number of times which fills up the limited bandwidth to main memory and becomes IO-bound.

4.2 Optimized Mappings

In [5, 4, 6] C. Kessler and J. Keller recognize that sorting on the Cell BE in a streaming fashion during the third step of CellSort [3] better utilizes the bandwidth between the processor and memory. A tree of merger nodes is proposed and the problem of how to best map the nodes onto the SPEs is investigated. A model taking into account the cost of communication, the computational and memory load of each SPE is described.

Assuming all input blocks have the same size, all nodes on the same level in the tree has the same computational load because they process equally much data. The root node must process all data, its children half of it each and so on. Computationally, mapping the nodes of the tree by the level is good, but substituting a node from the level above with two nodes from the lower is equally good.

Communicating on the same SPE is cheaper than cross SPE communication because no actual transfers need to happen, some pointer reordering is potentially all that it takes. Because of this, placing all nodes on the same SPE would be
Prior Work

Figure 4.1. Optimized 5-level tree.

optimal, but then the computational load gets very bad.

In here there is a non-trivial trade off between computation load, communication load and memory usage. C. Kessler and J. Keller [5] uses a ILP optimization solver to generate mappings with optimal trade offs. Figure 4.1 shows a optimized tree for 5 SPEs.

4.2.1 DC-Map

In [5] a simple heuristic for making a node map for a larger tree out of optimal mappings for a smaller tree is described. It is needed because calculating optimal mappings for large trees becomes very time consuming.

It works as follows. Let us say a mapping for a tree with 6 levels, $k = 6$, is sought but only a mapping for $k = 5$ exists. First the root is placed on its own SPE, leaving 5 SPEs. Each subtree of the root is mapped according to the
4.2 Optimized Mappings

mapping for $k = 5$. The SPEs in the subtrees are then sorted by the number of nodes mapped to them. The SPE with fewest nodes in one of the subtrees are combined with the SPE with most nodes in the other tree and so on until the two mappings have been combined into one.
Chapter 5

Problem Statement

The idea is to connect 2-to-1 mergers in a tree-like fashion in order to merge a set of sorted lists into a single sorted list. Three 2-to-1 mergers makes a 4-to-1 merger tree and so on. The numbers to be sorted are communicated in packages so that the tree is pipelined, or streamed. A pipelined system can easily be parallelized since each stage in the pipeline is isolated. For bigger trees there will be more nodes than SPEs on the Cell processor and hence multiple nodes must be placed on the same SPU. Each level in the merger tree processes equally many integers and thus performs equally much work. The naive way to map the nodes to the SPEs would be to simply put each level of the tree on its own SPE, that way the workload is spread out evenly. However, with such a mapping each node must communicate to its parent on a different SPU which ought to be more time consuming than to communicate locally. Maybe there are better ways to map the nodes to the SPEs, this was explored by C. Kessler and J. Keller in [5].

The objective of this work is to implement a merge tree, involving

- An efficient merger node using SIMD instructions
- A communication and synchronization scheme between the nodes
- User-level scheduling of nodes on the same SPE

The tree must be constructed from given mappings, given as matrices in text files as outputted from the optimizer used in [5]. The different trees should then be used to sort data and be timed in order to evaluate and confirm that optimized on-chip pipelining for mergesort does indeed give speedups.

Lastly, new performance bottlenecks and challenges for future work should be identified.
Chapter 6

Implementation

6.1 The Merger Node

A merger node consists of a small control structure, containing addresses to its buffers, its parents in-buffer, its id and so forth. The code for merging is shared for all nodes and parameterized on such a control structure.

6.2 The PPE

The PPE reads in the mapping and constructs the tree by setting up the control structures for the merger nodes. Each SPE has an array of control structures which get transferred from the PPE before the SPE starts executing. The PPE then waits for the SPEs to finish, doing nothing.

6.3 Mapping Nodes to SPEs

Mapping a node to an SPE consists of adding its control structure to the list of control structures on the SPE. The mapping between nodes and SPEs is read in from a text file generated by the ILP optimizer used in [5]. If a mapping file for the tree size being used can not be found, the mapping for a smaller tree is applied in the DC-map heuristic described in section 4.2.1.

Nodes on the same SPE are scheduled in a round-robin order, the list of control structures is looped over and each node gets a chance to run. The first thing each merger does is to check if it is done, in which case it yields. If it is not done it checks if it has any data in its in-buffers and if there is any room in the out-buffer. This involves checking if DMA operations have completed. If the merger has no data, or if the out-buffer is full, it yields and the next merger runs.
6.4 Communication

Data is pushed up the tree, except from the leaf nodes who also pull from main memory. The communication looks different depending on whether the parent node being pushed to is on the same SPE or not. In the case where the parent is local the memory flow controller can not be used because it demands that the receiving address is outside the sender’s local store. But because they are on the same SPE, the child’s out-buffer and its parent’s in-buffer can simply be the same. This eliminates the need for an extra out-buffer and makes more efficient use of the limited amount of memory in the local store.

6.4.1 Synchronization

When the child and parent node are on different SPEs communication must be done through the memory flow controller and synchronization must be carefully thought through. In this case, the child node has an out-buffer that it writes to while merging. When either of the in-buffers are depleted, or the out-buffer filled, as much of the out-buffer that fits in the parents in-buffer gets asynchronously transfered.

This eliminates the need for traditional double buffering on all SPEs except for the one holding the root merger. Transfer time is instead masked by letting the next merger on the SPE run. It can be seen as multi buffering but without the FIFO queue as described in 2.6.

Figure 6.1 illustrates how a node only reads from its in-buffers and thus only updates the tail pointers (see section 2.6) and never writes to the head pointers. A child node only writes to its parents in-buffers, which means it only writes to the head pointer and only read the tail position. Furthermore, when a child reads the tail position of its parents in-buffer, it reads the last known tail position. This means the calculated space available can never be larger than it really is, so it is safe to use even if the parent is using its buffer and is updating the tail position as it goes. The reverse is true for the head position, the child writes and the parent only reads. This means that no locks need to be used in the synchronization between nodes.

6.4.2 Tag Administration

As described in section 3.4 each SPE can keep track of 32 groups of memory transfers. In order for the communication scheme described above to work each node that communicates across SPEs needs its own tag, to know when its transfers are complete. Note that leaf nodes pulls data from main memory and therefore needs one tag for each of its two in-buffers. A leaf node thus needs two or three tags depending on whether its parent is on the same SPE or not. All inner nodes needs one or no tag and the root always one tag.

A node acquires and releases tags from the pool of 32 tags available. Because nodes runs in a round robin order, and tag administration is done by the node itself,
6.5 Memory

Each node needs two input buffers and one output buffer if its parent is residing on a different SPE. Each SPE has a fixed sized pool of memory for buffers that gets shared by the nodes. This means that nodes on less populated SPEs gets bigger buffers. Also, a SPE with high locality needs fewer output buffers and so gets bigger buffers than another SPE with equally many nodes but where more output buffers are needed. The model used in [5] does not take this into account when calculating the memory load during optimization of mappings.

Figure 6.1. Synchronization between child and parent on different SPEs.
6.6 Sorting Kernel

SIMD instructions are being used as much as possible in the innermost loops of the merger node. Merging two vectors is done purely with SIMD instructions as done in CellSort [3]. It is possible to use only SIMD instructions in the entire merge loop, this was tested but it did not reduce the time because in order to eliminate an if-statement it required quite a bit comparisons and moving data around redundantly.
Chapter 7

Results

7.1 Baseline

CellSort [3] is used as baseline for the results of the merger tree. Only phase 3, the merge phase, is later considered when comparing CellSort to this work. Figure 7.1 and 7.2 shows total sorting times for CellSort on PS3 and QS-20 respectively. Note that data sizes are bigger on QS-20 because more SPEs are involved in phase 1 and 2 of CellSort.

Figure 7.3 and 7.4 shows the relative times of the phases of CellSort and it is clear that the third phase scales badly.

7.2 Setup

Blocks of integers, two for each leaf node, were filled with random data and sorted. This mimics the state of the data after phase 2 of CellSort [3] explained in 4.1. Ideally each such block would be the size of the aggregate local storages on the processor. CellSort sorts 32Ki integers per SPE, blocks would be $4 \times 32768$ integers $\times 4B = 512$KiB on the PS3 and $16 \times 32768 \times 4B = 2$MiB on the QS-20. For example a 6 level tree has 32 leaf nodes, so optimal data size on the QS-20 would be $64 \times 2$MiB = 128MiB. However, block sizes of other sizes were used when testing in order to magnify the differences between mappings.

The parameter used to generate different mappings, called $\epsilon$, is varied between 0 and 1 to give preference to memory or communication load. For details see the paper on optimized mappings [5]. Mappings for $\epsilon = 0.1$, 0.5 and 0.9 were tested if available at the time of testing. Not all mappings were available at the time of testing, which means some data points are missing. Also, not all RAM is generally available to an application and on the PS3 about 190MB could be acquired, reducing the possible problem size to test.
Figure 7.1. Total sorting times for CellSort on PS3, showing partial times for the phases.

Figure 7.2. Total sorting times for CellSort on QS-20, showing partial times for the phases.
7.2 Setup

Figure 7.3. The relative time for the phases of CellSort on PS3.

Figure 7.4. The relative time for the phases of CellSort on QS-20.
7.3 Machine Configuration

7.3.1 PS3

The PS3 has 6 SPEs and 256MiB RAM. The code was compiled using gcc version 4.1.1, IBM Cell SDK 3.0 and run on Linux kernel version 2.6.23.

7.3.2 QS-20

The code was compiled using gcc version 4.1.1 and run on Linux kernel version 2.6.18-128.e15. The machine had 16 SPUs on 2 Cell processors and 1GiB RAM.

7.4 Results of Different Mappings

7.4.1 PS3

Mappings generated with $\epsilon = 0.1$, 0.5 and 0.9 was tested on different data and merger tree sizes. Results for 5 level trees are shown in figure 7.5 and for 6 level trees in figure 7.6.

7.4.2 QS-20

Mappings generated with $\epsilon = 0.1$, 0.5 and 0.9 was tested on different data and merger tree sizes. Results for $k = 8$ shown in figure 7.7, $k = 7$ in figure 7.8, $k = 6$ in figure 7.9 and for $k = 5$ in figure 7.10.
Figure 7.6. Merge times, $k = 6$ and different $\epsilon$ on the PS3.

Figure 7.7. Merge times, $k = 8$ and different $\epsilon$ on QS-20.
Figure 7.8. Merge times, $k = 7$ and different $\epsilon$ on QS-20.

Figure 7.9. Merge times, $k = 6$ and different $\epsilon$ on QS-20.
7.5 Results of Different Buffer Sizes

7.5.1 QS-20

The buffer size, number of vectors, was varied with a fixed $\epsilon = 0.5$. Figure 7.11 and 7.12 shows the merge times for a 5 level tree. Figure 7.13 and 7.14 shows merge times for a 7 level tree. There is a clear correlation between available buffer memory and speed of the merge tree. But by using 128KiB for buffers instead of 216KiB, 88KiB can be freed while keeping performance loss down. As a frame of reference the current implementation uses much less than 88KiB of the local store for code.

7.6 Results of DC Map

7.6.1 QS-20

Trees for $k = 8, 7$ and 6 were constructed using optimal smaller trees with $\epsilon = 0.5$ and tested with 64Mi integers. Results are shown in figure 7.15.

7.7 Compared to CellSort

When comparing the merge tree to CellSort one must be careful to compare apples with apples. As said in section 4.1 CellSort only works on $2^n$ SPEs, and was tested on 4 and 16 SPEs. This makes the blocks to merge in phase 3 either 512KiB or 2MiB in size. Each tree should be compared with a data size corresponding to using...
**Figure 7.11.** Merge time, $k = 5$ and $\epsilon = 0.5$, with different total buffer size per SPU on QS-20.

**Figure 7.12.** Normalized merge times, $k = 5$ and $\epsilon = 0.5$, with different total buffer size per SPU on QS-20.
7.7 Compared to CellSort

**Figure 7.13.** Merge times, $k = 7$ and $\epsilon = 0.5$, with different total buffer size per SPU on QS-20.

**Figure 7.14.** Normalized merge times, $k = 7$ and $\epsilon = 0.5$, with different total buffer size per SPU on QS-20.
Figure 7.15. Trees constructed from smaller trees using DC Map.

<table>
<thead>
<tr>
<th>#integers</th>
<th>CellSort [ms]</th>
<th>Optimized Merge Tree [ms]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4Mi</td>
<td>89,98</td>
<td>44,09</td>
<td>2,04</td>
</tr>
<tr>
<td>8Mi</td>
<td>225,92</td>
<td>93</td>
<td>2,43</td>
</tr>
</tbody>
</table>

Table 7.1. Merge times on the PS3.

These block sizes. For example, a 6 level tree has 32 leaf nodes, each connected to two blocks of 512kB or 2MiB. This resulting in total data sizes of 32MiB and 128MiB respectively. These are the data sizes one should compare a 6 level tree against CellSort with. Applying this, figure 7.16 shows merge times for CellSort and the merge tree of a fair size on the PS3. Figure 7.17 show the same thing on the QS-20.

<table>
<thead>
<tr>
<th>#integers</th>
<th>CellSort [ms]</th>
<th>Optimized Merge Tree [ms]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>16Mi</td>
<td>219</td>
<td>174</td>
<td>1,26</td>
</tr>
<tr>
<td>32Mi</td>
<td>565</td>
<td>350</td>
<td>1,61</td>
</tr>
<tr>
<td>64Mi</td>
<td>1316</td>
<td>973</td>
<td>1,35</td>
</tr>
</tbody>
</table>

Table 7.2. Merge times on the QS-20.
7.7 Compared to CellSort

Figure 7.16. CellSort compared to fastest mapping of the merge tree for fair data sizes on the PS3 (k=5,6).

Figure 7.17. CellSort compared to fastest mapping of the merge tree for fair data sizes on the QS-20 (k=5,6,7).
Chapter 8

Discussion

8.1 Mappings

Different mappings give some variation in execution times, it seems like the cost model used in the optimizer is more important than the parameters in it. The implementation does not reflect the cost used in the model very well. In the model each node is said to use equally much buffer memory but in the implementation nodes whose parent is on the same SPE use less buffer memory than nodes with a non-local parent.

8.2 Buffer Sizes

Looking at figure 7.12 and 7.14 it is clear that buffer size may be lowered a little with acceptable performance losses, thus allowing more code in the SPE. The freed space could possibly house the code for phase 1 and 2 of CellSort. Being able to load all code needed to sort would save the time it takes to load in a different program for the merge phase.

8.3 Conclusion

This thesis work has successfully shown the big benefits of pipelining a parallel merge sort. Reducing merge times compared to CellSort [3] by as much as a factor 2.43 on PS3 and 1.61 on QS20, see chapter 7.7. This because of fewer main memory accesses, not by exceptional programming skills or code optimizations. In fact, no profiling of the code was done which might show possibilities for improvement.

Furthermore, noticeable differences between different mappings of merger nodes to SPEs was found. Especially when using a different cost model.
8.4 Future Work

CellSort only works on $2^n$ integers in the local store of each SPE, limiting it to $2^{15} = 32768$ integers when doing local sort. This does not use all available space in the local store. Maybe a different algorithm than bitonic merge sort could be used, which could utilize all available memory. There are little or no performance benefits for CellSort on data sizes that fits the aggregate local stores, this is reported in [3]. Being able to sort more data locally produces bigger blocks to merge, which would reduce the total time for sorting.

Figure 7.7 shows sorting times for the same $\epsilon$ but for two different cost models used during optimization. The '0.5 uw' mapping is much faster than the regular one. The uw-model (unweighted) was not supposed to be used at all in this work and mapping using it was not available for all tree sizes. Reworking the cost model, or maybe reworking the implementation to better reflect the model, seems very beneficial.

When many nodes needs DMA tags, see section 3.4, the current implementation stalls to almost standstill. Maybe this could be avoided by a more dynamic DMA handling and scheduling of nodes. An interrupt handler can be set up to handle the completion of DMA transfers. This would free up DMA tags, allowing more nodes to work on the same SPE. It would also make smarter scheduling of merger nodes possible. Instead of the implemented round-robin scheduler, where each task is started and checked if it has resources enough to work, a ready-queue could be set up with nodes known to have enough resources.

Lastly, merging is only one memory intensive application that benefits from pipelining. Maybe a generalized framework that pipelines arbitrary work could be constructed.
Bibliography


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Rikard Hultén