Hybrid Parallel Sort on the Cell Processor

Jörg Keller[1], Christoph Kessler[2], Kalle König[3] and Wolfgang Heenes[3]

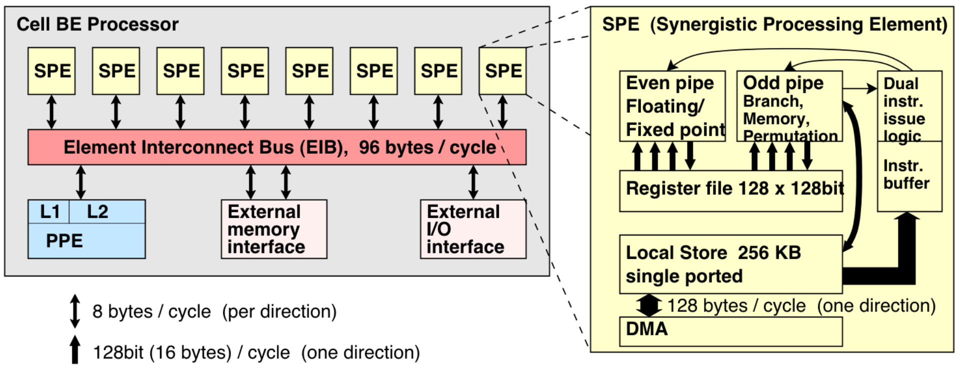
[1]FernUniversität in Hagen, Fak. Math. und Informatik, 58084 Hagen, Germany
joerg.keller@FernUni-Hagen.de
[2]Linköpings Universitet, Dept. of Computer and Inf. Science, 58183 Linköping, Sweden
chrke@ida.liu.se
[3]Technische Universität Darmstadt, FB Informatik, 64289 Darmstadt, Germany
kalle_koenig@gmx.de
heenes@ra.informatik.tu-darmstadt.de

Abstract: Sorting large data sets has always been an important application, and hence has been one of the benchmark applications on new parallel architectures. We present a parallel sorting algorithm for the Cell processor that combines elements of bitonic sort and merge sort, and reduces the bandwidth to main memory by pipelining. We present runtime results of a partial prototype implementation and simulation results for the complete sorting algorithm, that promise performance advantages over previous implementations.

Key words: Parallel Sort, Merge Sort, Cell Processor, Hybrid Sort

1 Introduction

Sorting is an important subroutine in many high performance computing applications, and parallel sorting algorithms have therefore attracted considerable interest continuously for the last 20 years, see e.g. [Akl85]. As efficient parallel sorting depends on the underlying architecture [AISW96], there has been a rush to devise efficient parallel sorting algorithms for every new parallel architecture on the market. The Cell BE processor (see e.g. [GEMN07] and the references therein) presents a challenge in this respect as it combines several types of parallelism within a single chip: each of its 8 parallel processing units (called SPEs) is an SIMD processor with a small local memory of 256 KBytes. The SPEs communicate via one-sided message-passing over a high-speed ring network with each other and with the off-chip main memory, that is shared among the SPEs, but not kept consistent. The chip also contains a multi-threaded Power processor core, which however is not of particular interest for our work. Although the majority of applications for Cell seem to be numerical algorithms, there have been several attempts to efficiently implement sorting on that architecture. We present a sorting algorithm that incorporates those attempts by combining bitonic sort, merge sort, and pipelining to reduce the bandwidth to main memory, which is seen as one of the major bottlenecks in Cell. We report on the performance of a partial prototype implementation, and on simulation results for the



Cell BE Processor
SPE
Element Interconnect Bus (EIB), 96 bytes / cycle
L1
L2
PPE
External memory interface
External I/O interface
8 bytes / cycle (per direction)
128bit (16 bytes) / cycle (one direction)
SPE (Synergistic Processing Element)
Even pipe Floating/ Fixed point
Odd pipe Branch, Memory, Permutation
Dual instr. issue logic
Instr. buffer
Register file 128 x 128bit
Local Store 256 KB single ported
128 bytes / cycle (one direction)
DMA

Figure 1: Cell BE Processor Architecture

full sorting algorithm. The combined results indicate that our algorithm outperforms all previous approaches.

The remainder of this article is organized as follows. In Section 2 we provide all technical information necessary to attack the problem at hand, and discuss related work. In Section 3 we detail our solution, report on the performance results for our pipelined merger components of the overall sorting algorithm on a Play Station 3, and extrapolate from these data for the analysis of the expected performance of the overall sorting algorithm. Section 4 concludes and gives an outlook on further developments.

2 Cell Processor and Parallel Sorting

The Cell BE processor [GEMN07] is a multi-core processor consisting of 8 SIMD processors called SPE and a dual-threaded Power core (PPE), cf. Fig. 1. Each SPE has a small local memory of 256 KBytes that contains code and data. There is no cache, no virtual memory, and no operating system on the SPE. The SPE has datapaths and registers 128 bits wide, and instructions to operate on them as on vector registers, e.g. perform parallel comparisons between two registers, each seen as holding four 32-bit values. Hence, control flow instructions tremendously slow down data throughput of an SPE. The main memory is off-chip, and can be accessed by all SPEs and the Power core, i.e. it is a shared memory. Yet, there is no protocol to automatically ensure coherency between local memories and main memory. Data transfer between an SPE and another SPE or the main memory is performed by DMA. Thus, data transfer and computation can be overlapped, but communications must be programmed at a very low level. The SPEs, the PPE core and the memory interface are interconnected by the Element Interconnect Bus (EIB). The EIB is implemented via four uni-directional rings with an aggregate bandwidth of 306.4 GByte/s. The bandwidth of each unit on the ring to send data over or receive data from the ring is only 25.6 GByte/s. Hence, the off-chip memory tends to become the performance bottleneck. Programming the Cell processor is challenging. The programmer has to strive to use the SPE's SIMD architecture efficiently, and has to take care for messaging and coherency, taking into account the rather small local memories.

The Cell processor seems to have two major application fields: gaming[1] and numerical algorithms. To our knowledge, there are only a few investigations about parallel sorting algorithms for the Cell processor, most notably [GBY07, IMKN07] that use bitonic sort and merge sort, respectively. There is also an implementation of radix sort [RB07], but only with a 4 bit radix, because for an 8 bit radix the array of counters would not fit into the SPE's local store. Also [GBY07] reports to have investigated radix sort but that it "involve(s) extensive scalar updates using indirection arrays that are difficult to SIMDize and thus, degrade the overall performance."

The paper [IMKN07] is quite close to our work and appeared only a few months before this article was written. Both [GBY07] and [IMKN07] work in two phases to sort a data set of size n, with local memories of size m. In the first phase, blocks of data of size $8m$ that fit into the combined local memories of the SPEs are sorted. In the second phase, those sorted blocks of data are combined to a fully sorted data set.

In [GBY07], the first phase is realized in two stages: in the first stage, each SPE sorts data in its local memory sequentially by a variant of Batcher's bitonic sort, then the SPEs perform Batcher's bitonic sort on the sorted data in their local memories. The combined content of their local memories is then written to main memory as a sorted block of data. This is repeated $n/(8m)$ times until the complete data set is turned into a collection of sorted blocks of data. The second phase performs Batcher's bitonic sort on those blocks. Batcher's sort is chosen because it needs no data dependent control flow and thus fully supports the SPE's SIMD architecture. The disadvantage is that $O(n \log^2 n)$ data have to be read from and written to main memory, which makes the main memory link the limiting speed factor, and the reason why the reported speedups are very small.

In [IMKN07], the first phase is also realized in two stages: in the first stage, each SPE performs a variant of combsort that exploits the SIMD capability of the SPE, then the SPEs perform a mergesort on the sorted data in their local memories. As in the first approach, this is repeated $n/(8m)$ times. The second phase is mergesort, that uses a so-called vectorized merge to exploit the SIMD instructions of the SPEs, and employs a 4-to-1-merge to reduce memory bandwidth. Yet, each merge procedure reads from main memory and writes to main memory, so that $n \log_4(n/(8m))$ data are read from and written to main memory during the second phase.

Our approach focuses on the second phase, as the first phase only reads and writes n data from and to the main memory, and thus is not as critical to the overall performance as the second phase. Also, there are known approaches for the first phase. We also implemented a vectorized merge routine, similar to that of [IMKN07] (then unknown to us), only that we perform 2-to-1 merges. The vectorization uses a variant of Batcher's bitonic sort to merge chunks of four successive 32-bit integers, as those will fit into one Cell register. However, there is a notable difference between our approach and that of [IMKN07]. We run mergers of several layers of the merge tree concurrently to form a pipeline, so that

[1]In a variant with 6 SPEs, Cell is deployed in the Play Station 3.

output from one merger is not written to main memory but sent to the SPE running the follow-up merger. Thus, we can realize 16-to-1 or 32-to-1 mergers between accesses to main memory, and reduce the memory bandwidth by a factor of 2 and 2.5, respectively, in relation to [IMKN07]. In order to exploit this advantage we have to ensure that our pipeline runs close to the maximum possible speed, which requires consideration of load balancing. More concretely, if a merger M must provide an output rate of k words per time unit, then the two mergers M_1, M_2 feeding its inputs must provide a rate of $k/2$ words per time unit on average. However, if the values in M_2 are much larger than in M_1, the merger M will only take values from the output of M_1 for some time, so that the merger M_1 must be able to run at a rate of k words for some time, or the output rate of M will reduce!

In principle, we could have mapped each layer of a binary merge tree onto one SPE, each SPE working the mergers of its layer in a time-slot fashion. A time slot is the time that a merger needs to produce one buffer full of output data, provided that its input buffers contain enough data. Thus, with k SPEs, we realize a 2^k-to-1 merge. This will balance load between the layers as the combined rate from one layer to the next is the same for all layers. Also, because of finite size buffers between the mergers, if M only draws from M_1, M_2 will not be able to work further and thus M_1 will get more time slots and be able to deliver faster. The disadvantage of this model is that the larger i, the more mergers SPE i has to host, which severely restricts the buffer size, because there must be one output buffer and two input buffers for each merger, that all must fit into about half the local memory of an SPE (the other half is for code and variables). Therefore, we devised mappings that minimize the maximum amount of mergers that one SPE has to host, and thus maximize the buffer size. We present two such mappings in the next section.

3 Experiments and Simulations

We have implemented the prototype core of a merger routine on a Cell processor from a Play Station 3. Despite including only 6 SPEs, it corresponds to the processor sold in blades by IBM. Our routine provides a bandwidth of 1.5 GByte/s. This indicates that with 8 SPEs concurrently reading and writing data as in [IMKN07], a bandwidth to main memory of $2 \times 8 \times 1.5 = 24$ GByte/s would be needed which would saturate the memory link. Assuming that the fully developed merger in [IMKN07] is more efficient than our prototype, we see that the bandwidth to main memory is the limiting performance factor in their design. Conversely, if we can reduce the memory bandwidth needed, we gain a corresponding factor in performance.

In order to get an impression of the performance of our full algorithm, we implemented a discrete event simulation[2] of the sorting algorithm. As the runtime of the merger core is only dependent on the size of the output buffer, it is more or less constant. As furthermore communication and computation are overlapped, we believe the simulation to accurately reflect the runtime of the full algorithm.

[2]While the merge algorithm is not very complex, implementing the DMA transfers is cumbersome and low-level, thus we decided to postpone the full implementation.

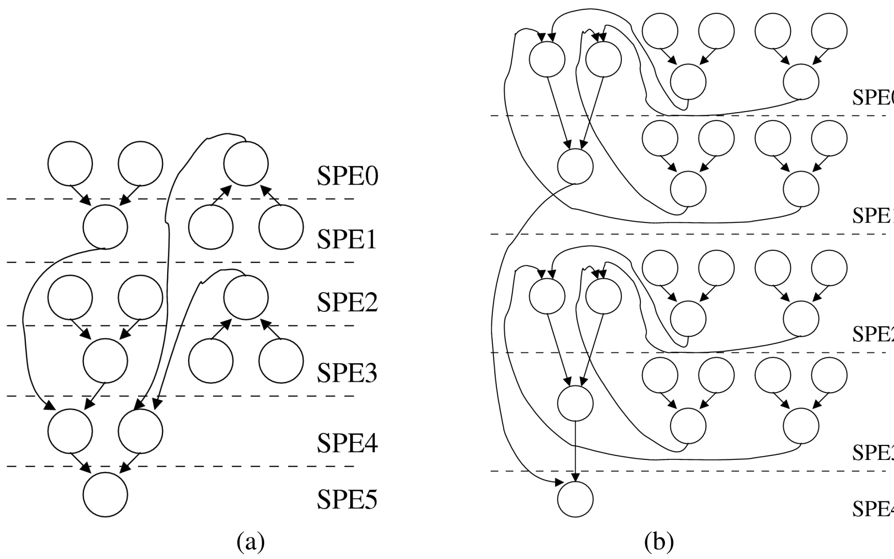


Figure 2: Mapping of merger nodes to SPEs

In each step, each SPE runs one merger with enough input data until it has produced one output buffer full of data. As buffer size, we use 4 KByte for the output buffer (holding 1,024 32-bit integers), and 2×4 KByte for the input buffers, in order to allow concurrent working of a merger and filling of its input with the output of a previous merger. Each merger receives a share of the SPE's processing time at least according to its position in the merge tree. For example, in Fig. 2(b), the merger left in SPE1 receives one half of the processing power, because it is placed in depth 1 of the merge tree, while the other mergers receive 1/8 and 1/16 of the processing power, respectively, because they are in depths 3 and 4, respectively. We use a simple round robin scheduling policy in each SPE, where a merger not ready to run because of insufficient input data or full output buffer is left out.

We have investigated two mappings, depicted in Fig. 2. Both try to place neighboring mergers in one SPE as often as possible, in order to exploit the load balancing discussed in the previous section. In mapping (a), the mergers of SPE0 and SPE1 (similarly SPE2 and SPE3) could have been placed in one SPE, but we decided to give them twice the processor share to be on the safe side and avoid load balancing and pipeline stall problems. We have simulated this mapping with 16 input blocks of 2^{20} sorted integers each. The blocks were randomly chosen and then sorted. In all experiments, the pipeline ran with 100% efficiency as soon as it was filled. As we realize a 16-to-1 merge, we gain a factor of 2 on the memory bandwidth in relation to [IMKN07]. Yet, as we need 6 instead of 4 SPEs to do this, our real improvement is only $2 \cdot 4/6 = 4/3$ in this case.

In mapping (b), we have implemented a 32-to-1 merge, with the restriction that no more than 8 mergers are to be mapped to one SPE. With 20 KByte of buffers for each merger, this seems to be upper limit. Here each merger has a processing share according to its position in the merge tree. We used 32 input blocks of 2^{20} sorted integers each, chosen as before. The pipeline ran with an efficiency of 93%, meaning that in 93% of the time steps, the merger on SPE4 could be run and produced output. In comparison to [IMKN07], our memory bandwidth decreased by a factor of 2.5. Combined with a pipeline efficiency of 93%, we still gain a factor of 1.86 in performance.

4 Conclusion and Future Work

We have provided a new sorting algorithm for the Cell Processor Architecture that uses a vectorized merge sort in a pipelined variant to reduce memory bandwidth. Our simulation results indicate that the performance of a full implementation of our algorithm will show better performance than previous algorithms. Future work will consist of obtaining this implementation.

Note that our algorithm is also able to run on multiple Cell processors, as does [IMKN07]. At the beginning, there will be many blocks, and hence many 16-to-1 or 32-to-1 mergers can be employed. In the end, when nearing the root, we are able to employ a method already known and mentioned in [IMKN07]: we partition the very large data blocks and perform merges on the partitions in parallel.

References

[AISW96] Nancy M. Amato, Ravishankar Iyer, Sharad Sundaresan, and Yan Wu. A Comparison of Parallel Sorting Algorithms on Different Architectures. Technical Report 98-029, Texas A&M University, January 1996.

[Akl85] Selim G. Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.

[GBY07] Bugra Gedik, Rajesh Bordawekar, and Philip S. Yu. CellSort: High Performance Sorting on the Cell Processor. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *VLDB*, pages 1286–1207. ACM, 2007.

[GEMN07] Michael Gschwind, David Erb, Sid Manning, and Mark Nutter. An Open Source Environment for Cell Broadband Engine System Software. *Computer*, 40(6):37–47, 2007.

[IMKN07] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *Proc. 16th Int.l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 189–198. IEEE Computer Society, 2007.

[RB07] N. Ramprasad and Pallav Kumar Baruah. Radix Sort on the Cell Broadband Engine. In *Int.l Conf. High Perf. Comuting (HiPC) – Posters*, 2007.