# Optimized on-chip-pipelined mergesort on the Cell/B.E.

Rikard Hultén[1], Christoph W. Kessler[1], and Jörg Keller[2]

[1] Linköpings Universitet, Dept. of Computer and Inf. Science, 58183 Linköping, Sweden
[2] FernUniversität in Hagen, Dept. of Math. and Computer Science, 58084 Hagen, Germany

**Abstract.** Limited bandwidth to off-chip main memory is a performance bottleneck in chip multiprocessors for streaming computations, such as Cell/B.E., and this will become even more problematic with an increasing number of cores. Especially for streaming computations where the ratio between computational work and memory transfer is low, transforming the program into more memory-efficient code is an important program optimization. In earlier work, we have proposed such a transformation technique: on-chip pipelining.
On-chip pipelining reorganizes the computation so that partial results of subtasks are forwarded immediately between the cores over the high-bandwidth internal network, in order to reduce the volume of main memory accesses, and thereby improves the throughput for memory-intensive computations. At the same time, throughput is also constrained by the limited amount of on-chip memory available for buffering forwarded data. By optimizing the mapping of tasks to cores, balancing a trade-off between load balancing, buffer memory consumption, and communication load on the on-chip bus, a larger buffer size can be applied, resulting in less DMA communication and scheduling overhead.
In this paper, we consider parallel mergesort on Cell/B.E. as a representative memory-intensive application in detail, and focus on the global merging phase, which is dominating the overall sorting time for larger data sets. We work out the technical issues of applying the on-chip pipelining technique for the Cell processor, describe our implementation, evaluate experimentally the influence of buffer sizes and mapping optimizations, and show that optimized on-chip pipelining indeed reduces, for realistic problem sizes, merging times by up to 70% on QS20 and 143% on PS3 compared to the merge phase of CellSort, which was by now the fastest merge sort implementation on Cell.

## 1 Introduction

The new generation of multiprocessors-on-chip derives its raw power from parallelism, and explicit parallel programming with platform-specific tuning is needed to turn this power into performance. A prominent example is the Cell Broadband Engine [1] with a PowerPC core and 8 parallel slave processors called SPEs. Yet, many applications use the Cell BE like a dancehall architecture: the SPEs use their small on-chip local memories (256 KB for both code and data) as explicitly-managed caches, and they all load and store data from/to the external (off-chip) main memory. The bandwidth to the external memory is much smaller than the SPEs' aggregate bandwidth to the on-chip interconnect bus (EIB). This limits performance and prevents scalability.

External memory is also a bottleneck in other multiprocessors-on-chip. This problem will become more severe as the core count per chip is expected to increase considerably in the foreseeable future. Scalable parallelization on such architectures therefore must use direct communication between the SPEs to reduce communication with off-chip main memory.

In this paper, we consider the important domain of memory-intensive computations and consider the global merging phase of pipelined mergesort on Cell as a challenging case study, for the following reasons:

- The ratio of computation to data movement is low.
- The computational load of tasks varies widely (by a factor of $2^k$ for a binary merge tree with $k$ levels).
- The computational load of a merge task is not fixed but only averaged.
- Memory consumption is not proportional to computational load but constant among tasks.
- Communication always occurs between tasks of different computational load.

These factors complicate the mapping of tasks to SPEs. In total, pipelining a merge tree is much more difficult than task graphs of regular problems such as matrix vector multiplication.

The task graph of the global merging phase consists of a tree of merge tasks that should contain, in the lowest layer, at least as many merger tasks as there are SPEs available. Previous solutions like CellSort [2] and AAsort [3] process the tasks of the merge tree layer-wise bottom-up in serial rounds, distributing the tasks of a layer equally over SPEs (there is no need to have more than one task per SPE). Each layer of the tree is then processed in a *dancehall* fashion, where each task operates on (buffered) operand and result arrays residing in off-chip main memory. This organization leads to relatively simple code but puts a high access load on the off-chip-memory interface.

*On-chip pipelining* reorganizes the overall computation in a pipelined fashion such that intermediate results (i.e., temporary stream packets of sorted elements) are not written back to main memory where they wait for being reloaded in the next layer processing round, but instead are forwarded immediately to a consuming successor task that possibly runs on a different core. This will of course require some buffering in on-chip memory and on-chip communication of intermediate results where producer and consumer task are mapped to different SPEs, but multi-buffering is necessary anyway in processors like Cell in order to overlap computation with (DMA) communication. It also requires that all merger tasks of the algorithm be active simultaneously; usually there are several tasks mapped to a SPE, which are dynamically scheduled by a user-level round-robin scheduler as data is available for processing.

However, as we would like to guarantee fast context switching on SPEs, the limited size of Cell's local on-chip memory then puts a limit on the number of buffers and thus tasks that can be mapped to an SPE, or correspondingly a limit on the size of data packets that can be buffered, which also affects performance. Moreover, the total volume of intermediate data forwarded on-chip should be low and, in particular, must not exceed the capacity of the on-chip bus. Hence, we obtain a constrained optimization problem for mapping the tasks of streaming computations to the SPEs of Cell such that the resulting throughput is maximized.

In previous work we developed mappings for merge trees [4,5]. In particular, we have developed various optimal, approximative and heuristic mapping algorithms for optimized on-chip pipelining of merge trees. Theoretically, a tremendous reduction of required memory bandwidth could be achieved, and our simulations for an idealized Cell architecture indicated that considerable speedup over previous implementations are possible. But an implementation on the real processor is very tricky if it should overcome the overhead related to dynamic scheduling, buffer management, synchronization and communication delays. Here, we detail our implementation that actually achieves notable speedup of up to 61% over the best previous implementation, which supports our earlier theoretical estimations by experimental evidence. Also, the results support the hypothesis that on-chip pipelining as an algorithmic engineering option is worthwhile in general because simpler applications might profit even more.

The remainder of this article is organized as follows. In Section 2, we give a short overview of the Cell processor, as far as needed for this article. Section 3 develops the on-chip pipelined merging algorithm, Section 4 gives details of the implementation, and Section 5 reports on the experimental results. Further details will soon be available in a forthcoming master thesis [6]. Section 6 concludes and identifies issues for future work.

## 2 Cell/B.E. Overview

The Cell/B.E. (Broadband Engine) processor [1] is a heterogeneous multi-core processor consisting of 8 SIMD processors called SPE and a dual-threaded PowerPC core (PPE), which differ in architecture and instruction set. In earlier versions of the Sony PlayStation-3™ (PS3), up to 6 SPEs of its Cell processor could be used under Linux. On IBMs Cell blade servers such as QS20 and later models, two Cells with a total of 16 SPEs are available. Cell blades are used, for instance, in the nodes of RoadRunner, which was the world's fastest supercomputer in 2008–2009.

While the PPE is a full-fledged superscalar processor with direct access to off-chip memory via L1 and L2 cache, the SPEs are optimized for doing SIMD-parallel computations at a significantly higher rate and lower power consumption than the PPE. The SPE datapaths and registers are 128 bits wide, and the SPU vector instructions operate on them as on vector registers, holding 2 doubles, 4 floats or ints, 8 shorts or 16 bytes, respectively. For instance, four parallel float comparisons between the corresponding sections of two vector registers can be done in a single instruction. However, branch instructions can tremendously slow down data throughput of an SPE. The PPE should mainly be used for coordinating SPE execution, providing OS service and running control intensive code.

Each SPE has a small local on-chip memory of 256 KBytes. This *local store* is the only memory that the SPE's processing unit (the SPU) can access directly, and therefore it needs to accommodate both SPU code and data. There is no cache and no virtual memory on the SPE. Access to off-chip memory is only possible by asynchronous DMA *put* and *get* operations that can communicate blocks of up to 16KB size at a time to and from off-chip main memory. DMA operations are executed asynchronously by the SPE's memory flow controller (MFC) unit in parallel with the local SPU; the SPU

can initiate a DMA transfer and synchronize with a DMA transfer's completion. DMA transfer is also possible between an SPE and another SPE's local store.

There is no operating system or runtime system on the SPE except what is linked to the application code in the local store. This is what necessitates user-level scheduling if multiple tasks are to run concurrently on the same SPE.

SPEs, PPE and the memory interface are interconnected by the Element Interconnect Bus (EIB) [1]. The EIB is implemented by four uni-directional rings with an aggregate bandwidth of 204 GByte/s (peak). The bandwidth of each unit on the ring to send data over or receive data from the ring is only 25.6 GB/s. Hence, the off-chip memory tends to become the performance bottleneck if heavily accessed by multiple SPEs.

Programming the Cell processor efficiently is a challenging task. The programmer should partition an application suitably across the SPEs and coordinate SPE execution with the main PPE program, use the SPE's SIMD architecture efficiently, and take care of proper communication and synchronization at fairly low level, overlapping DMA communication with local computation where possible. All these different kinds of parallelism are to be orchestrated properly in order to come close to the theoretical peak performance of about 220 GFlops (for single precision).

To allow for overlapping DMA handling of packet forwarding (both off-chip and on-chip) with computation on Cell, there should be at least buffer space for 2 input packets per input stream and 2 output packets per output stream of each streaming task to be executed on an SPE. While the SPU is processing operands from one buffer, the other one in the same buffer pair can be simultaneously filled or drained by a DMA operation. Then the two buffers are switched for each operand and result stream for processing the next packet of data. (Multi-buffering extends this concept from 2 to an arbitrary number of buffers per operand array, ordered in a circular queue.) This amounts to at least 6 packet buffers for an ordinary binary streaming operation, which need to be accommodated in the size-limited local store of the SPE. Hence, the size of the local store part used for buffers puts an upper bound on the buffer size and thereby on the size of packets that can be communicated.

On Cell, the DMA packet size cannot be made arbitrarily small: the absolute minimum is 16 bytes, and in order to be not too inefficient, at least 128 bytes should be shipped at a time. Reasonable packet sizes are a few KB in size (the upper limit is 16KB). As the size of SPE local storage is severely limited (256KB for both code and data) and the packet size is the same for all SPEs and throughout the computation, this means that the maximum number of packet buffers of the tasks assigned to any SPE should be as small as possible. Another reason to keep packet size large is the overhead due to switching buffers and user-level runtime scheduling between different computational tasks mapped to the same SPE. Figure 1 shows the sensitivity of the execution time of our pipelined mergesort application (see later) to the buffer size.

## 3 On-chip pipelined mergesort

Parallel sorting is needed on every modern platform and hence heavily investigated. Several sorting algorithms have been adapted and implemented on Cell BE. The highest

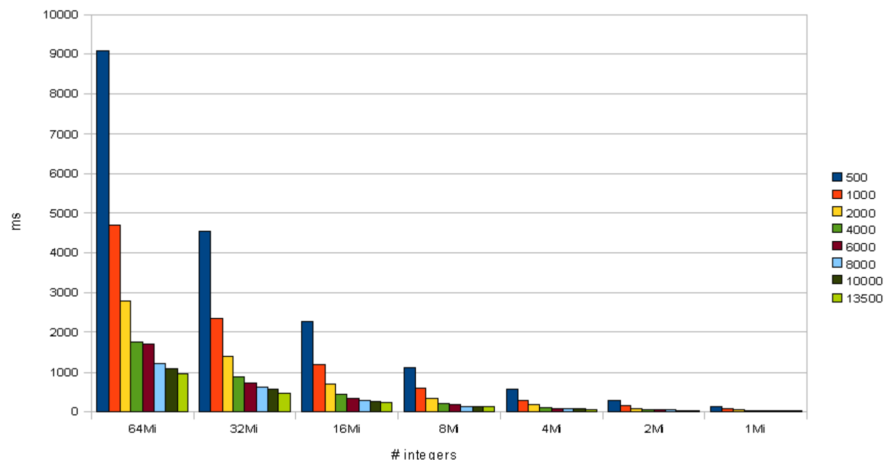


**Fig. 1.** Merge times (here for a 7-level merger tree pipeline), shown for various input sizes (number of 128bit-vectors per SPE), strongly depend on the buffer size used in multi-buffering.

performance is achieved by Cellsort [2] and AAsort [3]. Both sort data sets that fit into off-chip main memory but not into local store. Both implementations have similarities.

They work in two phases to sort a data set of size $N$ with local memories of size $N'$. In the first phase, blocks of data of size $8N'$ that fit into the combined local memories of the 8 SPEs are sorted. In the second phase, those sorted blocks of data are combined to a fully sorted data set. We concentrate on the second phase as the majority of memory accesses occurs there and as it accounts for the largest share of sorting time for larger input sizes. In CellSort [2], this phase is realized by a bitonic sort because this avoids data dependent control flow and thus fully exploits SPE's SIMD architecture. Yet, $O(N \log^2 N)$ memory accesses are needed and the reported speedups are small. In AAsort [3], mergesort with 4-to-1-mergers is used in the second phase. The data flow graph of the merge procedures thus forms a fully balanced merge quadtree. The nodes of the tree are executed on the SPEs layer by layer, starting with the leaf nodes. As each merge procedure on each SPE reads from main memory and writes to main memory, all $N$ words are read from and written to main memory in each merge round, resulting in $N \log_4(N/(8N')) = O(N \log_4 N)$ data being read from and written to main memory. While this improves the situation, speedup still is limited.

In order to decrease the bandwidth requirements to off-chip main memory and thus increase speedup, we use on-chip pipelining. This means that all merge nodes of all tree levels are active from the beginning, and that results from one merge node are forwarded in packets of fixed size to the follow-up merge node directly without usage of main memory as intermediate store. With $b$-to-1 merger nodes and a $k$-level merge tree, we realize $b^k$-to-1 merging with respect to main memory traffic and thus reduce main memory traffic by a factor of $k \cdot \log_4(b)$.

The decision to forward merged data streams in packets of fixed size allows to use buffers of this fixed size for all merge tasks, and also enables follow-up merge tasks

to start work before predecessor mergers have handled their input streams completely, thus keeping as many merge tasks busy as possible, and allowing pipeline depths independent of the lengths of data streams. Note that already the mergers in the AAsort algorithm [3] must work with buffering and fixed size packets.

The requirement to keep all tasks busy is complicated by the fact that the processing of data streams is not completely uniform over all tasks but depends on the data values in the streams. A merger node may consume only data from one input stream for some time, if those data values are much smaller than the data values in the other input streams. Hence, if all input buffers for those streams are filled, and the output buffers of the respective predecessor merge tasks are filled as well, those merge tasks will be stalled. Moreover, after some time the throughput of the merger node under consideration will be reduced to the output rate of the predecessor merger producing the input stream with small data values, so that follow-up mergers might also be stalled as a consequence. Larger buffers might alleviate this problem, but are not possible if too many tasks are mapped to one SPE.

Finally, the merger nodes should be distributed over the SPEs such that two merger nodes that communicate data should be placed onto the same SPE whenever possible, to reduce communication load on the EIB. As a secondary goal, if they cannot be placed onto the same SPE, they might be placed such that their distance on the EIB is small, so that different parts of the EIB might be used in parallel.

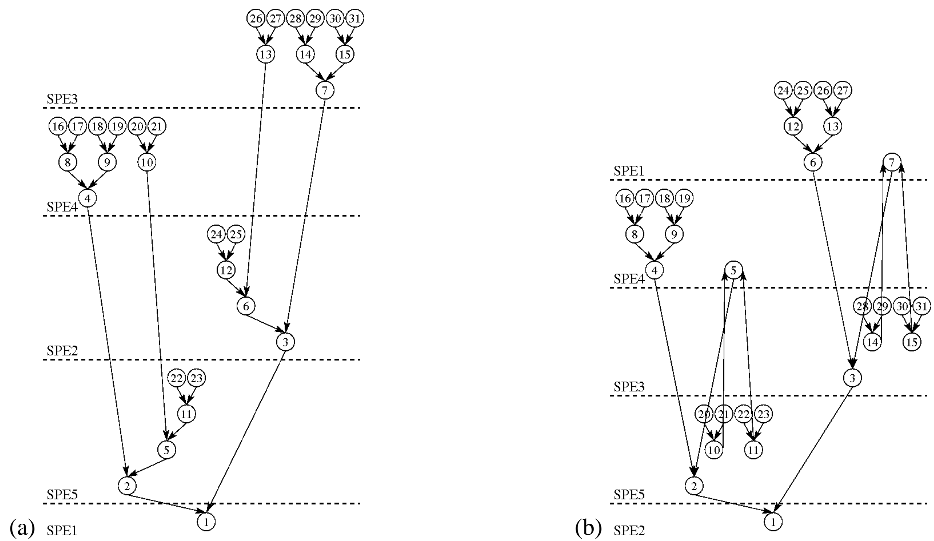


**Fig. 2.** Two Pareto-optimal solutions for mapping a 5-level merge tree onto 5 SPEs, computed by the ILP solver [4]. (a) The maximum memory load is 22 communication buffers (SPE4 has 10 nodes with 2 input buffers each, and 2 of these have output buffers for cross-SPE forwarding) and communication load 1.75 (times the root merger's data output rate); (b) max. memory load 18 and communication load 2.5. The (expected) computational load is perfectly balanced (1.0 times the root merger's load on each SPE) in both cases.

In our previous work [4], we have formulated the above problem of mapping of tasks to SPEs as an integer linear programming (ILP) optimization problem with the constraints given. An ILP solver (we use CPLEX 10.2 [7]) can find optimal mappings for small tree sizes (usually for $k \leq 6$) within reasonable time; for $k = 7$, it can still produce an approximative solution. For larger tree sizes, we used an approximation algorithm [4].

The ILP based mapping optimizer can be configured by a parameter $\epsilon \in (0, 1)$ that controls the priority of different secondary optimization goals, for memory load or communication load; computational load balance is always the primary optimization goal. Example mappings computed with different $\epsilon$ for a 5-level tree are visualized in Fig. 2.

## 4 Implementation details

*Merging kernel* SIMD instructions are being used as much as possible in the innermost loops of the merger node. Merging two (quad-word) vectors is completely done with SIMD instructions as in CellSort [2]. In principle, it is possible to use only SIMD instructions in the entire merge loop, but we found that it did not reduce time because the elimination of an if-statement required too many comparisons and moving data around redundantly.

*Mapping optimizer* The mapping of merger task nodes to SPEs is read in by the PPE from a text file generated by the mapping optimizer. The PPE generates the task descriptors for each SPE at runtime, so that our code in not constrained to a particular merge-tree, but still optimized to the merge-tree currently used. Due to the complexity of the optimization problem, optimal mappings can be (pre-)computed only for smaller tree sizes up to $k = 6$. For larger trees, we use the approximative mapping algorithm DC-map [4] that computes mappings by recursively composing mappings for smaller trees, using the available optimal mappings as base cases.

*SPE task scheduler* Tasks mapped to the same SPE are scheduled by a user-level scheduler in a round-robin order. A task is ready to run if it has sufficient input and an output buffer is free. A task runs as long as it has both input data and space in the output buffer, and then initiates the transfer of its result packet to its parent node and returns control to the scheduler loop. If there are enough other tasks to run afterwards, DMA time for flushing the output buffer is masked and hence only one output buffer per task is necessary (see below). Tasks that are not data-ready are skipped.

As the root merger is always alone on its SPE, no scheduler is needed there and many buffers are available; its code is optimized for this special case.

*Buffer management* Because nodes (except for the root) are scheduled round-robin, the DMA latency can, in general, be masked completely by the execution of other tasks, and hence double-buffering of input or output streams is not necessary at all, which reduces buffer requirements considerably. An output stream buffer is only used for tasks whose parents/successors reside on a different SPE. Each SPE has a fixed sized pool of

memory for buffers that gets equally shared by the nodes. This means that nodes on less populated SPEs, for instance the root merger that has a single SPE on its own, can get larger buffers (yet multiples of the packet size). Also, a SPE with high locality (few edges to tasks on other SPEs) needs fewer output buffers and thus may use larger buffers than another SPE with equally many nodes but where more output buffers are needed. A larger buffering capacity for certain tasks (compared to applying the worst-case size for all) reduces the likelihood of an SPE sitting idle as none of its merger tasks is data-ready.

*Communication* Data is pushed upwards the tree (i.e., producers/child nodes control cross-SPE data transfer and consumers/parent nodes acknowledge receipt) except for the leaf nodes which pull their input data from main memory.

The communication looks different depending on whether the parent (consumer) node being pushed to is located on the same SPE or not. If the parent is local, the memory flow controller cannot be used because it demands that the receiving address is outside the sender's local store. Instead, the child's output buffer and its parent's input buffer can simply be the same. This eliminates the need for an extra output buffer and makes more efficient use of the limited amount of memory in the local store.

The (system-global) addresses of buffers in the local store on the opposite side of cross-SPE DMA communications are exchanged between the SPEs in the beginning.

*Synchronization* Each buffer is organized as cyclic buffer with a head and a tail pointer.

A task only reads from its input buffers and thus only updates the tail pointers and never writes to the head pointers. A child node only writes to its parent's input buffers, which means it only writes to the head pointer and only reads from the tail position.

The parent task updates the tail pointer of the input buffer for the corresponding child task; the child knows how large the parent's buffer is and how much it has written itself to the parent's input buffer so far, and thus knows how much space is left for writing data into the buffer. In particular, when a child reads the tail position of its parent's input buffer, the value is pessimistic so it is safe to use even if the parent is currently using its buffer and is updating the tail position simultaneously. The reverse is true for the head position, the child writes to the parent's head position of the corresponding input buffer and the parent only reads. This means that no locks are needed for the synchronization between nodes.

*DMA tag management* A SPE can have up to 32 DMA transfers in flight simultaneously and uses tags in $\{0, ..., 31\}$ to distinguish between these when polling the DMA status. The Cell SDK offers an automatic tag manager for dynamic tag allocation and release. However, if an SPE has many buffers used for remote communication, it may run out of tags. If that happens, the tag-requesting task gives up, steps back into the task queue and tries to initiate that DMA transfer again when it gets scheduled next.

## 5 Experimental results

We used a Sony PlayStation-3 (PS3) with IBM Cell SDK 3.0 and an IBM blade server QS20 with SDK 3.1 for the measurements. We evaluated for as large data sets as could

fit into RAM on each system, which means up to 32Mi integers on PS3 (6 SPEs, 256 MiB RAM) and up to 128Mi integers on QS20 (16 SPEs, 1GiB RAM). The code was compiled using gcc version 4.1.1 and run on Linux kernel version 2.6.18-128.e15.

A number of blocks equal to the number of leaf nodes in the tree to be tested were filled with random data and sorted. This corresponds to the state of the data after the local sorting phase (phase 1) of CellSort [2]. Ideally, each such block would be of the size of the aggregated local storage available for buffering on the processor. CellSort sorts 32Ki (32,768) integers per SPE, blocks would thus be 4 × 128KiB = 512KiB on the PS3 and 16 × 128KiB = 2MiB on the QS20. For example, a 6-level tree has 64 leaf nodes, hence the optimal data size on the QS20 would be 64 × 512KiB = 32MiB. However, block sizes of other sizes were used when testing in order to magnify the differences between mappings.

Different mappings (usually for $\epsilon = 0.1$, 0.5 and 0.9) were tested.

### 5.1 On-chip-pipelined merging times

The resulting times with on-chip pipelining for 5-level and 6-level trees on PS3 are shown in Fig. 3. For QS20, mappings generated with $\epsilon = 0.1$, 0.5 and 0.9 were tested on different data sizes and merger tree sizes from $k = 5$ to $k = 8$, see Fig 4. We see that the choice of the mapping can have a major impact on merging time, as even mappings that are optimal for different optimization goals exhibit timing differences of up to 25%.

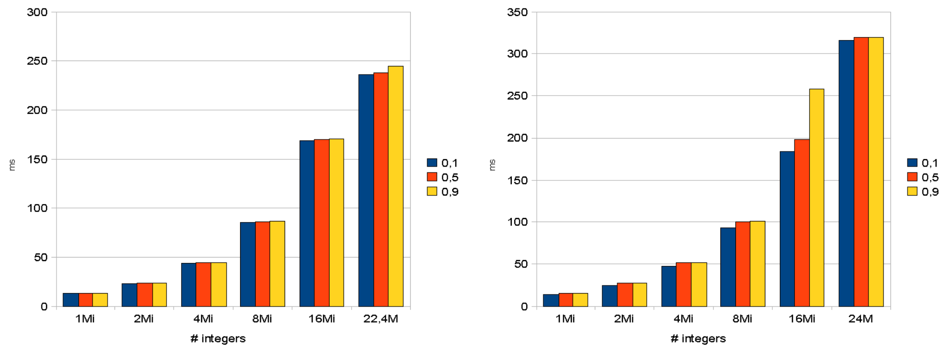


**Fig. 3.** Merge times for $k = 5$ (left) and $k = 6$ (right) for different mappings ($\epsilon$) on PS3.

### 5.2 Results of DC-map

Using the DC-map algorithm [4], mappings for trees for $k = 8$, 7 and 6 were constructed by recursive composition using optimal mappings (computed with the ILP algorithm with $\epsilon = 0.5$) as base cases for smaller trees. Fig. 5 shows the result for merging 64Mi integers on QS20.

### 5.3 Comparison to CellSort

Table 1 shows the direct comparison between the global merging phase of CellSort (which is dominating overall sorting time for large data sets like these) and on-chip-

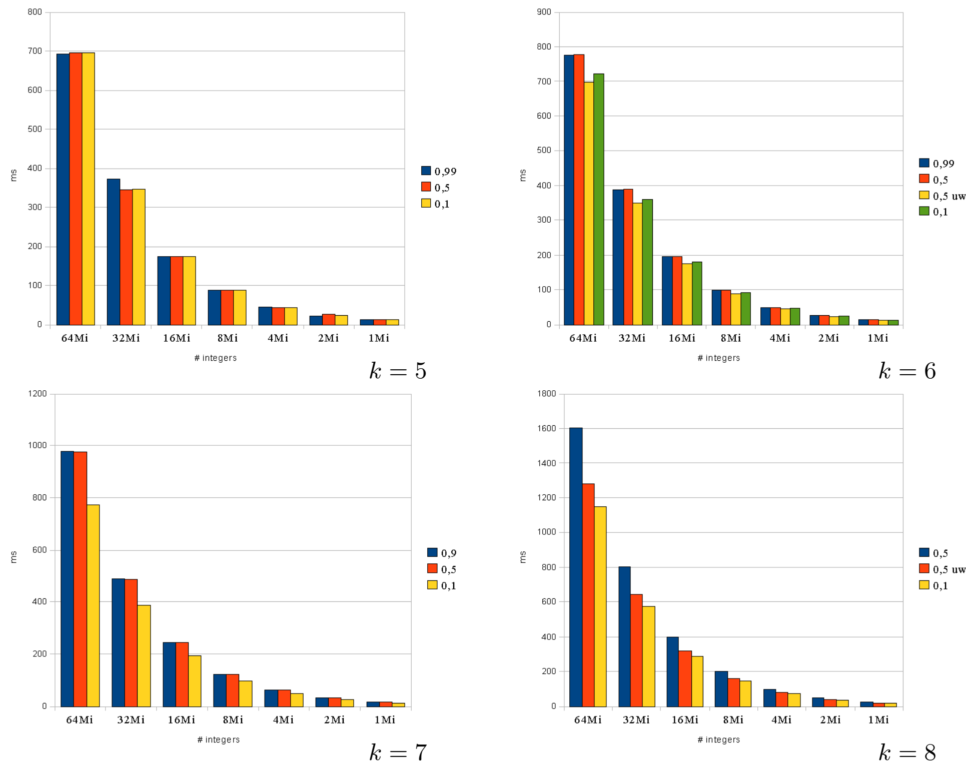


**Fig. 4.** Merge times for $k = 5, 6, 7, 8$ and different input sizes and mappings ($\epsilon$) on QS20.

**Fig. 5.** Merge times (64 Mi integers) for trees ($k = 8, 7, 6$) constructed from smaller trees using DC-map.

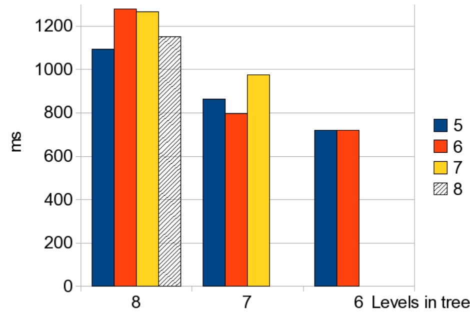


**Table 1.** Timings for the CellSort global merging phase vs. Optimized on-chip-pipelined merging for global merging of integers on QS20

| $k$ | #ints | CellSort Global Merging | On-Chip-Pipelined Merging | Speedup |
|---|---|---|---|---|
| 5 | 16Mi | 219 ms | 174 ms | 1.26 |
| 6 | 32Mi | 565 ms | 350 ms | 1.61 |
| 7 | 64Mi | 1316 ms | 772 ms | 1.70 |

pipelined merging with the best mapping chosen. We achieve significant speedups for

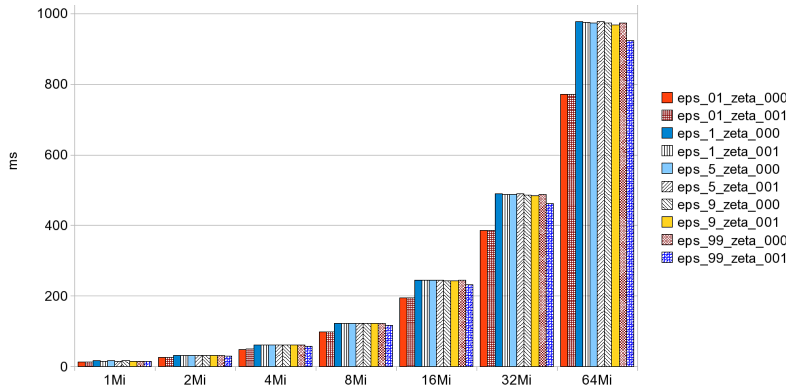


**Fig. 6.** Merge times on QS20 for $k = 7$, further mappings.

on-chip-pipelining in all cases; the best speedup of 70% can be obtained with 7 SPEs (64Mi elements) on QS20, using the mapping with $\epsilon = 0.01$ in Fig. 6; the corresponding speedup figure for the PS3 is 143% at $k = 5$, 16Mi elements. This is due to less communication with off-chip memory.

### 5.4 Discussion

Different mappings gives some variation in execution times, it seems like the cost model used in the mapping optimizer is more important than the priority parameters in it.

Also with on-chip pipelining, using deeper tree pipelines (to fully utilize more SPEs) is not always beneficial beyond a certain depth $k$, here for $k = 6$ for PS3 and $k = 8$ for QS20, as a too large number of tasks increases the overhead of on-chip pipelining (smaller buffers, scheduling overhead, tag administration, synchronization, communication overhead). The overall pipeline fill/drain overhead is more significant for lower workloads but negligible for the larger ones.

From Fig. 1 it is clear that, with optimized mappings, buffer size may be lowered without losing much performance, which frees more space in the local store of the SPEs, e.g. for accommodating the code for all phases of CellSort, saving the time overhead for loading in a different SPE program segment for the merge phase.

## 6 Conclusion and Future Work

With an implementation of the global merging phase of parallel mergesort as a case study of a memory-intensive computation, we have demonstrated how to lower memory bandwidth requirements in code for the Cell BE by optimized on-chip pipelining. We obtained speedups of up to 70% on QS20 and 143% on PS3 over the global merging phase of CellSort, which dominates the sorting time for larger input sizes.

On-chip pipelining is made possible by several architectural features of Cell that may not be available in other multicore processors. For instance, the possibility to forward data by DMA between individual on-chip memory units is not available on current GPUs where communication is only to and from off-chip global memory. The possibility to lay out buffers in on-chip memory and move data explicitly is not available on

cache-based multicore architectures. Nevertheless, on-chip pipelining will be applicable in upcoming heterogeneous architectures for the DSP and multimedia domain with a design similar to Cell, such as ePUMA [9]. Intels forthcoming 48-core *single-chip cloud computer* [8] will support on-chip forwarding between tiles of two cores, with 16KB buffer space per tile, to save off-chip memory accesses.

On-chip pipelining is also applicable to other streaming computations such as general data-parallel computations or FFT. In [10] we have described optimal and heuristic methods for optimizing mappings for general pipelined task graphs.

The downside of on-chip pipelining is complex code that is hard to debug. We are currently working on an approach to *generic on-chip pipelining* where, given an arbitrary acyclic pipeline task graph, an (optimized) on-chip-pipelined implementation will be generated for Cell. This feature is intended to extend our BlockLib skeleton programming library for Cell [11].

*Acknowledgements* C. Kessler acknowledges partial funding from EU FP7 (project *PEPPHER*, #248481), VR (*Integr. Softw. Pipelining*), SSF (*ePUMA*), Vinnova, and CUGS. We thank Niklas Dahl and his colleagues from IBM Sweden for giving us access to their QS20 blade server.

## References

1. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell Broadband Engine Architecture and its first implementation—a performance view. IBM J. Res. Devel. **51**(5) (Sept. 2007) 559–572
2. Gedik, B., Bordawekar, R., Yu, P.S.: Cellsort: High performance sorting on the Cell processor. In: Proc. 33rd Int.l Conf. on Very Large Data Bases. (2007) 1286–1207
3. Inoue, H., Moriyama, T., Komatsu, H., Nakatani, T.: 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), IEEE Computer Society (2007) 189–198
4. Keller, J., Kessler, C.W.: Optimized pipelined parallel merge sort on the Cell BE. In: Proc. 2nd Workshop on Highly Parallel Processing on a Chip (HPPC-2008) at Euro-Par 2008, Gran Canaria, Spain. (2008)
5. Kessler, C.W., Keller, J.: Optimized on-chip pipelining of memory-intensive computations on the Cell BE. In: Proc. 1st Swedish Workshop on Multicore Computing (MCC-2008), Ronneby, Sweden. (2008)
6. Hultén, R.: On-chip pipelining on Cell BE. Forthcoming master thesis, Dept. of Computer and Information Science, Linköping University, Sweden (2010)
7. ILOG Inc.: Cplex version 10.2. www.ilog.com (2007)
8. Howard, J., *et al.*: A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. Proc. IEEE International Solid-State Circuits Conference, pp. 19–21 (February 2010)
9. Liu, D., *et al.*: ePUMA parallel computing architecture with unique memory access. www.da.isy.liu.se/research/scratchpad/ (2009)
10. Kessler, C.W., Keller, J.: Optimized mapping of pipelined task graphs on the Cell BE. In: Proc. 14th Int. Worksh. on Compilers for Par. Computing Zürich, Switzerland. (Jan. 2009)
11. Ålind, M., Eriksson, M., Kessler, C.: Blocklib: A skeleton library for Cell Broadband Engine. In: Proc. ACM Int. Workshop on Multicore Software Engineering (IWMSE-2008) at ICSE-2008, Leipzig, Germany. (May 2008)