IFI TECHNICAL REPORTS

Institute of Computer Science,
Clausthal University of Technology

IfI-06-10

Clausthal-Zellerfeld 2006

# GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures

(extended version)*

Alexander Greß[1] and Gabriel Zachmann[2]

[1]Institute of Computer Science II
Rhein. Friedr.-Wilh.-Universität Bonn
Bonn, Germany
gress@cs.uni-bonn.de

[2]Institute of Computer Science
Clausthal University of Technology
Clausthal, Germany
zach@in.tu-clausthal.de

**Abstract**

In this paper, we present a novel approach for parallel sorting on stream processing architectures. It is based on adaptive bitonic sorting. For sorting $n$ values utilizing $p$ stream processor units, this approach achieves the optimal time complexity $O((n \log n)/p)$.

While this makes our approach competitive with common sequential sorting algorithms not only from a theoretical viewpoint, it is also very fast from a practical viewpoint. This is achieved by using efficient linear stream memory accesses and by combining the optimal time approach with algorithms optimized for small input sequences.

We present an implementation on modern programmable graphics hardware (GPUs). On recent GPUs, our optimal parallel sorting approach has shown to be remarkably faster than sequential sorting on the CPU, and it is also faster than previous non-optimal sorting approaches on the GPU for sufficiently large input sequences. Because of the excellent scalability of our algorithm with the number of stream processor units $p$ (up to $n/\log^2 n$ or even $n/\log n$ units, depending on the stream architecture), our approach profits heavily from the trend of increasing number of fragment processor units on GPUs, so that we can expect further speed improvement with upcoming GPU generations.

## 1 Introduction

Sorting is one of the most well-studied problems in computer science since it is a fundamental problem in many applications, in particular as a preprocessing step to accelerate searching. Due to the current trend of parallel architectures finding their way into common consumer hardware, parallel algorithms such as parallel sorting are becoming more and more important for the practice of programming.

While the classical programming model used in languages like C/C++ had been very successful for the development of non-parallel applications as it provides an efficient mapping to the classical von Neumann architecture, this model does not map very well to next generation parallel architectures which demand further input from the programmer to exploit the parallelism of an algorithm more effectively. For developing efficient applications on such architectures with maximum programmer productivity, alternative programming paradigms seem to be required [Ama05]. The stream programming model has shown to be a promising approach going in this direction. Furthermore, the stream programming model provided the foundations for the architecture of modern programmable high-performance graphics hardware (GPUs) that can be found in today's consumer hardware.

However, sorting on stream architectures is not much explored until now. Recent work on sorting on stream architectures includes several approaches based on sorting networks with $O(n \log^2 n/p)$ average and worst-case time, but to our knowledge no sorting algorithms for stream processors with optimal time complexity $O(n \log n/p)$ have been proposed so far.

Our approach, which is based on Adaptive Bitonic Sorting [BN89], achieves this optimal time complexity on stream architectures with up to $p = n/\log n$ processor units. The approach can even be implemented on stream architectures with the restriction that a stream must consist of a single contiguous memory block, in

*A shortened version of this paper appeared in [GZ06]

which case the optimal time complexity is achieved up to $p = n/\log^2 n$ units. Altogether this means that our approach will scale well to practically any future stream architecture.

Although we specify our approach completely in a general stream programming model, it has been designed with special attention to the practicability on modern GPUs, hence the name *GPU-ABiSort*. The GPU implementation and timings we provide in this paper show that our approach is not only optimal from a theoretical viewpoint, but also efficient in practice. Because of the scalability of our approach, we conjecture that the performance benefit of our parallel algorithm compared to sequential sorting will be even higher on future GPUs, provided that their rapid performance increase continues.

The rest of this paper is organized as follows: In Section 2 we will describe the related work on GPU-based sorting and on parallel sorting in general. In Section 3 we will summarize the stream programming model that lays the foundations for the specification of our approach. In Section 4 we will recap and slightly improve the classic adaptive bitonic sorting in the sequential case. We will present our novel optimal parallel sorting approach on stream architectures in Section 5 and supplement the description with some GPU-specific details in Section 6. In Section 7 we will show how to combine this asymptotically optimal approach with algorithms specifically tuned to small input sequences to obtain an even better overall performance in practice. Finally, we will provide the timings of our GPU implementation in Section 8. In addition, Appendix A provides a documented pseudo code of our sorting approach on stream architectures.

# 2 Related work

## 2.1 Optimal parallel sorting

Many innovative parallel sorting algorithms have been proposed for several different parallel architectures. For a comprehensive review, we refer the reader to [Akl90].

Especially parallel sorting using sorting networks as well as algorithms for sorting on a CREW-PRAM or EREW-PRAM model have been extensively studied. Ajtai, Komlos, and Szemeredi [AKS83] showed how optimal asymptotic complexity can be achieved with a sorting network. Cole [Col88] presented a parallel merge sort approach for the CREW-PRAM as well as for the EREW-PRAM, which achieves optimal asymptotic complexity on that architecture. However, although asymptotically optimal, it has been shown, that neither the AKS sorting network nor Cole's parallel merge sort are fast in practice for reasonable numbers of values to sort [GR88, Nat90].

Adaptive bitonic sorting [BN89] is another optimal parallel sorting approach for a shared-memory EREW-PRAM architecture (also called PRAC for parallel random access computer). It requires a smaller number of comparisons than Cole's approach (less than $2n \log n$ in total for a sequence of length $n$) and has a smaller constant factor in the running time. Even with a small number of processors it is efficient: In its original implementation, the sequential version of the algorithm was maximally 2.5 times slower than quick sort (for sequence lengths up to $2^{19}$) [BN89].

Besides, the main motivations for choosing this algorithm as basis for our parallel sorting approach on stream architectures were the following observations: First, adaptive bitonic sorting can run in $O(\log^2 n)$ parallel time on a PRAC with $O(n/\log n)$ processors. This allows us to develop an algorithm for stream architectures with only $O(\log^2 n)$ stream operations, as we will show in this paper. Note that a low number of stream operations is a key requirement for an efficient stream architecture implementation (see Section 3.1). Second, although originally designed for a random-access architecture, adaptive bitonic sorting can be adapted to a stream processor, which does not have the ability of random-access writes, as we will show in this paper.

Adaptive bitonic sorting is based on Batcher's bitonic sorting network [Bat68], which is a conceptually simpler approach that achieves only the non-optimal parallel running time $O(\log^2 n)$ for a sorting network of $n$ nodes.

## 2.2 GPU-based sorting

Several sorting approaches on stream architectures have been published so far. Apparently all of them are based on the bitonic or similar sorting networks and thus achieve only the non-optimal time complexity $O((n \log^2 n)/p)$ on a stream architecture with $p$ processor units (in worst *and* average case since sorting networks are data-independent).

Purcell et al. [PDC*03] presented a bitonic sorting network implementation for the GPU which is based on an equivalent implementation for the Imagine stream processor by Kapasi et al. [KDR*00]. Kipfer et al. [KSW04, KW05] implemented a bitonic as well as an odd-even merge sort network on the GPU.

Govindaraju et al. presented an implementation based on the periodic balanced sorting network [GRM05] and, more recently, also an implementation based on the bitonic sorting network [GRHM05]. The latter has been highly optimized for cache efficiency and is the fastest of the approaches above. On an NVIDIA GeForce 7800 GTX GPU it performs more than twice as fast as the best quick sort implementation on a single-core Intel Pentium IV CPU (up to the maximum data size that can be handled on such a GPU). However, because of the non-optimal time complexity of the bitonic sorting network it is not clear to what extent their approach will be competitive to optimal sorting on the CPU in the future, especially with the advent of multi-core CPUs, on which optimal parallel sorting can be implemented. As in other bitonic sorting network based approaches, their GPU implementation is restricted to power-of-two sequence lengths.

In a recent paper [GGKM05], Govindaraju et al. embedded the GPU-based bitonic sorting algorithm into a hybrid CPU/GPU sorting approach which is capable of processing large out-of-core databases and wide sort keys. This is achieved by adding a key generator stage and a reorder stage, which are performed on the CPU, as well as separate reader and writer stages to transfer data between disks and main memory using direct memory access (DMA). The resulting hybrid bitonic-radix sort technique utilizing GPU and CPU demonstrates nicely how GPU-based sorting in general can be made applicable to large databases and wide sort keys independent of current GPU register and memory size restrictions. This technique should also be transferable to alternative GPU-based sorting approaches.

# 3 The stream programming model

## 3.1 The basics

In the stream programming model [KDR*00, Owe02, BFH*04, Owe05], the basic program structure is described by streams of data passing through computation kernels. A *stream* is an ordered set of data of an arbitrary (simple or complex) data type. *Kernels* perform computation on entire streams or substreams, usually by applying a function to each element of the stream or substream (in parallel or in sequence). Kernels operate on one or more streams as inputs and produce one or more streams as outputs.

Programs expressed in the stream programming model are specified at two levels: the stream level and the kernel level (possibly using different programming languages at both levels). Computations on stream elements, usually consisting of multiple arithmetic operations, are specified at the kernel level. At the stream level, the program is constructed by chaining these computations together.

Furthermore, at the stream level it is possible to derive a *substream* from a given stream. A substream can be defined as a contiguous range of elements from a given stream. This way we can declare any contiguous block of stream memory as a stream or substream on which stream operations can be performed. On some stream hardware (including the GPU), a substream can also be defined by multiple non-overlapping ranges of elements from a stream.

The execution of a certain kernel for all elements of a stream or substream is invoked by a single operation on the stream level (*stream operation*). Since in theory all kernel instances for a single stream operation may be executed in parallel, the number of stream operations of a given stream program also provides a theoretical bound for the parallel running time of an algorithm. Therefore, if an identical operation is to be performed on a number of data elements, it is more efficient if these data elements reside in a common stream, on which a single stream operation can be applied, than if they are contained in multiple small streams, which would require the execution of multiple stream operations.

In addition to improving the scalability of an approach, the reduction of the number of stream operations is also very relevant for the practical performance of an algorithm on a given stream hardware. This is because of the (constant) overhead associated with each stream operation. Current stream hardware, especially GPUs, have the best throughput for large streams (consisting of hundreds or more elements) [Owe05]. Furthermore, it can be assumed that an operation on a substream defined by a single large contiguous range of elements is more efficient than the same operation on a substream defined by numerous small ranges of elements.

### 3.2 The target architecture for our approach in more detail

While the stream programming model described in the previous section was originally developed for special stream processor hardware such as Imagine and Merrimac [KDR*00, KRD*03], also the programmable graphics hardware (GPUs) contained in recent PCs have very similar capabilities, and thus recently the stream programming model is also often used to describe general purpose applications and algorithms implemented on this kind of hardware. However, since GPUs were originally designed for graphics applications, there are some GPU-specific properties and limitations when implementing stream programs for the GPU.

On the GPU, streams can be organized as 1D, 2D, or 3D arrays. Unfortunately, streams currently have restrictions on their size in each dimension (usually 2048 or 4096 elements on recent GPUs). This restriction is especially unpleasant for 1D streams which can thus be used only for a very small amount of stream memory. However, larger 1D streams can be represented by packing the data into a 2D stream. Each time an element of such a stream is accessed from a kernel via an index, the 1D index must be converted to a 2D index [BFH*04]. In 2D, we define a substream as a rectangular block or a set of multiple (non-overlapping) rectangular blocks of successive elements from a 2D stream.

On the GPU, *gathering* from a stream, i.e. random reads from a computed address, is possible, although in general less efficient than streaming reads. *Scattering* to a stream, i.e. random writes to a computed address, is not possible directly. It can at best be emulated on recent GPUs (see [BFH*04]), but such an emulation has a large overhead and, depending on the used technique, either increases the asymptotic time per processor or endangers the scalability of the algorithm by performing random writes successively that could theoretically be executed in parallel. Therefore, such an emulation is not suitable for our approach.

Summarizing, our targeted processor model is a stream processor with the ability to gather but without the ability to scatter. To apply the technique of adaptive bitonic sorting, originally proposed for an EREW-PRAM architecture, to such a target model, random access writes have to be replaced by stream writes, preferably to contiguous stream blocks as large as possible.

## 4 The sequential case

In the following, we will give a quick recap of the classic adaptive bitonic sorting approach for the sequential case (Section 4.1). Afterwards, we will propose a small modification of the merge algorithm, which will lead to a slightly more efficient implementation on stream architectures (Section 4.2).

Note that for simplicity, we assume in this description that the length of the input sequence $n$ is a power of two. This can be achieved by padding the input sequence. (Alternatively, Bilardi and Nicolau show an extended variant of their algorithm that works for arbitrary $n$ [BN89].) Further, it is assumed that all elements of the input sequence are distinct. Distinctness can be enforced by using the original position of the elements in the input sequence as secondary sort key.

### 4.1 The classic adaptive bitonic sorting approach

As already mentioned, the adaptive bitonic sorting approach [BN89] is based on the bitonic sorting scheme originally proposed by Batcher [Bat68]. This is a merge-sort based scheme, where the merge step is performed by reordering a bitonic sequence.

A sequence is called *bitonic* if there is a value of $j$ such that after rotation by $j$ elements, the sequence consists of a monotonic increasing part followed by a monotonic decreasing part. In this context, *rotation* by $j$ elements, $j \in \{0, \ldots, n-1\}$, denotes the following operation on the sequence: $(a_0, \ldots, a_{n-1}) \mapsto (a_j, \ldots, a_{n-1}, a_0, \ldots, a_{j-1})$. For an arbitrary $j$, the rotation is defined as the rotation by $j \mod n$.

For bitonic sorting, an algorithm is needed to transform a bitonic sequence into its corresponding monotonic increasing (or monotonic decreasing) sequence. With such an algorithm, the merging of two sorted sequences can be performed as follows: Assuming that the two sequences are sorted in opposite sorting directions (otherwise one of them would have to be reversed), the concatenation of the two sequences yields a bitonic sequence. Thus the result of the transformation into a monotonic increasing (or decreasing) sequence corresponds to the result of merging the two input sequences according to the respective sorting direction.

A key idea of bitonic sorting is to perform this transformation, which is called *bitonic merge*, recursively. For simplicity, we assume that the length of the bitonic input sequence is a power of two. Furthermore,

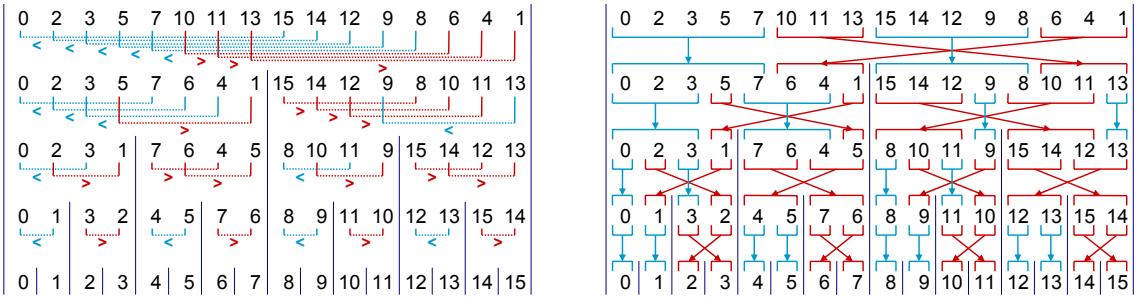



Figure 1: Bitonic merge of 16 values. *Left:* Each element of the first half is compared with its respective element in the second half. The minimum values are written to the first half, the maximum values to the second half. *Right:* This can be achieved by exchanging a block of $|j^*|$ values from the two halves (marked red).

we assume in the following that a monotonic increasing sequence has to be constructed. (A monotonic decreasing sequence could be constructed in an analogous manner.) Then the recursive scheme of the bitonic merge is as follows:

**Bitonic merge:**

- Let $p = (p_0, \ldots, p_{\frac{n}{2}-1})$ be the first half and $q = (q_0, \ldots, q_{\frac{n}{2}-1})$ the second half of input sequence $a = (a_0, \ldots, a_{n-1})$, i.e. $p_i = a_i$ and $q_i = a_{i+\frac{n}{2}}$.
- Let $p'$ and $q'$ be the component-wise minimum and maximum, respectively, of $p$ and $q$, i.e. $p'_i = \min(p_i, q_i)$ and $q'_i = \max(p_i, q_i)$.
- Then the following proposition holds (as we will show):
  (*) $p'$ and $q'$ are bitonic sequences, and the largest element of $p'$ is not greater than the smallest element of $q'$.
- Apply the **bitonic merge** recursively to the sequences $p'$ and $q'$. Afterwards, the concatenation of the two results yields the monotonic increasing sequence.

Figure 1 left demonstrates this algorithm on a bitonic sequence of 16 values.

We will shortly explain proposition (*) here (a more detailed proof can be found in [BN89]): It is easy to see that for each bitonic sequence $a$ consisting of $n$ elements, there is a $j^* \in \{-\frac{n}{2}, \ldots, \frac{n}{2}-1\}$ such that after rotation of $a$ by $j^*$ elements, all elements of the first half, which we call $p^*$, are not greater than any element of the second part, which we call $q^*$. (Note that it is sufficient to prove this for sequences consisting of a monotonic increasing part followed by a monotonic decreasing part.) Moreover it is obvious that $p^*$ and $q^*$ are bitonic sequences since they are parts of a bitonic sequence. If we rotate $p^*$ and $q^*$ by $-j^*$ elements, these sequences are equal to $p'$ and $q'$, respectively, which follows from the definition of $p', q'$ and the fact that, per definition of $p^*, q^*$, each $p^*_i$ cannot be greater than $q^*_i$. Therefore, proposition (*) follows from the definition of the sequences $p^*, q^*$ and the mentioned property that they are bitonic.

From these observations, we can derive an alternative method for determining the sequences $p'$ and $q'$. It is easy to see that if we have determined a value of $j^* \in \{-\frac{n}{2}, \ldots, \frac{n}{2}-1\}$ satisfying the above definition, $p'$ and $q'$ can be constructed from $p, q$ by exchanging the first $j^*$ elements of $p$ with the first $j^*$ elements of $q$ (in the case of $j^* \geq 0$) or by exchanging the last $-j^*$ elements of $p$ with the last $-j^*$ elements of $q$ (in the case of $j^* < 0$). This is demonstrated in Figure 1 right.

Consequently, $j^*$ is an index such that in case of

$$j^* \geq 0: \quad p_0 \geq q_0, \quad \ldots, \quad p_{j^*-1} \geq q_{j^*-1}, \qquad p_{j^*} \leq q_{j^*}, \quad \ldots, \quad p_{\frac{n}{2}-1} \leq q_{\frac{n}{2}-1}$$

$$j^* < 0: \quad p_0 \leq q_0, \quad \ldots, \quad p_{\frac{n}{2}+j^*-1} \leq q_{\frac{n}{2}+j^*-1}, \quad p_{\frac{n}{2}+j^*} \geq q_{\frac{n}{2}+j^*}, \quad \ldots, \quad p_{\frac{n}{2}-1} \geq q_{\frac{n}{2}-1}$$

If we assume that all elements of the input sequence $a$ are distinct (what we will do in the following), we can determine which of the two cases ($j^* \geq 0$ or $j^* < 0$) applies by a single comparison, for example according to the equivalence $j^* \geq 0 \Leftrightarrow p_{\frac{n}{2}-1} < q_{\frac{n}{2}-1}$. Thereafter, the exact value of $j^*$ (which is uniquely determined by the above definition in the case of distinct input elements) can be determined by a binary search. (In the case of $j^* \geq 0$ this means that, starting with $i = \frac{n}{4} - 1$, $i$ is decremented by a certain value if $p_i < q_i$, and incremented if $p_i > q_i$.)

Thus we have a method to determine $j^*$ in logarithmic time (using $\log n$ comparisons for a sequence consisting of $n$ elements). The key idea of the adaptive bitonic sorting approach [BN89] is to use this technique

to reduce the time complexity of the bitonic merge. For this purpose, also the number of exchanges (or data transfer operations in general) that is required to calculate $p'$ and $q'$ from a given $j^*$ has to be logarithmic.

To achieve this, the elements of a given bitonic sequence are stored as nodes of a binary search tree, which is called *bitonic tree*. The assumption that the sequence length $n$ is a power of two allows us to use only fully balanced binary trees. Each node of the tree contains an element of the subsequence $(a_0, \ldots, a_{n-2})$ in such a way that the in-order traversal of the tree yields this subsequence in correct order. $a_{n-1}$, the last element of the sequence, is stored separately (called *spare* node).

The benefit of using a binary tree is that a whole subtree (containing $2^k - 1$ sequence elements for a $k \in \{0, \ldots, \log n - 1\}$) can be replaced with another subtree by a single pointer exchange. This way, we can efficiently construct $p'$ and $q'$ during the binary search for determining the value of $j^*$. This leads to the following algorithm for the construction of $p'$ and $q'$, which operates on the bitonic tree that corresponds to the given bitonic sequence:

**Adaptive min/max determination:**

Phase 0: Determine, which of the two cases applies:
(a) **root** value $<$ **spare** value or
(b) **root** value $>$ **spare** value
Only in case (b):
Exchange the values of **root** and **spare**.
Let **p** be the left and **q** the right son of **root**.

For $i = 1, \ldots, \log n - 1$:

Phase $i$: Test if: value of **p** $>$ value of **q** (**)
If condition (**) is true:
Exchange the values of **p** and **q** as well as
in case (a) the left sons of **p** and **q**,
in case (b) the right sons of **p** and **q**.
Assign the left sons of **p**, **q** to **p**, **q** iff
case (a) applies and condition (**) is false or
case (b) applies and condition (**) is true;
otherwise assign the right sons of **p**, **q** to **p**, **q**.

Note that **root** contains the sequence element $p_{\frac{n}{2}-1}$ and **spare** the sequence element $q_{\frac{n}{2}-1}$ (where $p, q$ are the two halves of the given bitonic sequence). Therefore, case (a) corresponds to $j^* \geq 0$ and case (b) to $j^* < 0$ according to denotations above.

The described method requires $\log n$ comparisons and less than $2 \log n$ exchanges for the determination of $p'$ and $q'$. If this method is used within the bitonic merge scheme described before, we get a recursive merge algorithm in $O(n)$ which is called *adaptive bitonic merge*. This is because on each recursion level $k \in \{0, \ldots, \log n - 1\}$ (called *stage* in the following) there are $2^k$ sequences, each of them having the length $2^{\log n - k}$. So, on a stage $k$, we need $2^k(\log n - k)$ comparisons, which makes a total of $2n - \log n - 2$ and thus a linear time complexity for the whole merge algorithm.

Note that the bitonic tree does not need to be rebuild on each stage. Instead, we can formulate the *adaptive bitonic merge* algorithm completely on the basis of the bitonic tree:

**Adaptive bitonic merge:**

- Assume that a bitonic tree (for a sequence consisting of $n$ elements) is given by the nodes **root** and **spare**.
- Execute phases $0, \ldots, \log n - 1$ of the **adaptive min/max determination** algorithm as described above.
- Apply the **adaptive bitonic merge** recursively
  1. with **root's left son** as new root and **root** as new spare node,
  2. with **root's right son** as new root and **spare** as new spare node.

  (Finally, the in-order traversal of the whole bitonic tree results in the monotonic ascending sequence that was to be determined.)

Using the adaptive bitonic merge as merge algorithm in a classic recursive merge sort scheme the way it was described at the beginning of this section finally gives us the sequential version of adaptive bitonic sorting. It has a total running time of $O(n \log n)$ for input sequences of length $n$. Before extending this approach to a parallel algorithm for stream architectures, we will at first propose a slight modification of the

classic adaptive bitonic merge algorithm presented in this section, which eliminates the distinction of cases and thus will make an implementation on stream architectures easier and also more efficient.

### 4.2 Adaptive bitonic merge simplified

As described in the previous section, at the heart of the adaptive bitonic merge is an *adaptive min/max determination* algorithm that determines the component-wise minimum as well as the component-wise maximum of the bitonic sequences $p$ and $q$ in $O(\log n)$ time. As minimum and maximum are commutative, the result does not change if $p$ and $q$ are exchanged before applying this algorithm. Therefore, it is easy to assure that for any input sequences $p, q$ the inequality $p_{\frac{n}{2}-1} < q_{\frac{n}{2}-1}$ holds by simply exchanging $p$ and $q$ if applicable. This way, case (b) in the algorithm will be reduced to case (a). If this potential exchange of $p$ and $q$ is incorporated in phase 0 of the algorithm, this results in the following simplified implementation of the algorithm:

**Adaptive min/max determination:**

Phase 0: If **root** value $>$ **spare** value:
Exchange the values of **root** and **spare**
as well as the two sons of **root** with each other.
Let **p** be the left and **q** the right son of **root**.

For $i = 1, \ldots, \log n - 1$:

Phase $i$: If value of **p** $>$ value of **q**:
Exchange the values of **p** and **q**
as well as the left sons of **p** and **q**.
Assign the right sons of **p**, **q** to **p**, **q**.
Otherwise:
Assign the left sons of **p**, **q** to **p**, **q**.

In comparison to the implementation described in Section 4.1 only a single pointer exchange was added. Instead, it was possible to remove the distinction of cases.

## 5 Adaptive bitonic sorting on stream architectures

Based on the sequential sorting approach described in the previous section, we will now develop our optimal parallel sorting approach for stream architectures. For simplicity, we will initially ignore the fact that random-access writes are not possible on our targeted architecture, and start the description with an overview of the general outline of our approach.

### 5.1 GPU-ABiSort basic outline

On each recursion level $j = 1, \ldots, \log n$ of the adaptive bitonic sort, the adaptive bitonic merge algorithm has to be applied to $2^{\log n - j}$ bitonic trees, each consisting of $2^j$ nodes. As explained in Section 4.1, the merge is performed in $j$ stages. In each stage $k = 0, \ldots, j-1$, the adaptive min/max determination algorithm is executed on $2^k$ subtrees for each pair of bitonic trees that is to be merged. Therefore $2^{\log n - j} \cdot 2^k$ instances of the adaptive min/max determination algorithm can be executed in parallel in that stage. On a stream architecture this potential parallelism can be exposed by allocating a stream consisting of $2^{\log n - j + k}$ elements and executing a kernel on each element.

The adaptive min/max determination algorithm consists of $j - k$ phases, where each phase reads and modifies a pair of nodes from a bitonic tree. Let us assume that a kernel implementation is given that performs the operation of a single phase of the adaptive min/max determination algorithm. (How such a kernel implementation is realized without random-access writes will be described in Section 5.2.) The temporary data that has to be preserved from one phase of the algorithm to the other are just two node pointers (**p** and **q**) per kernel instance in case of the simplified version of the algorithm, which was described in Section 4.2. Thus each of the $2^{\log n - j + k}$ elements of the allocated stream should consist of exactly two node pointers. When the kernel is invoked on that stream, each kernel instance reads a pair of node pointers **p**, **q** from the stream, performs a phase of the adaptive min/max determination algorithm using **p**, **q** (as

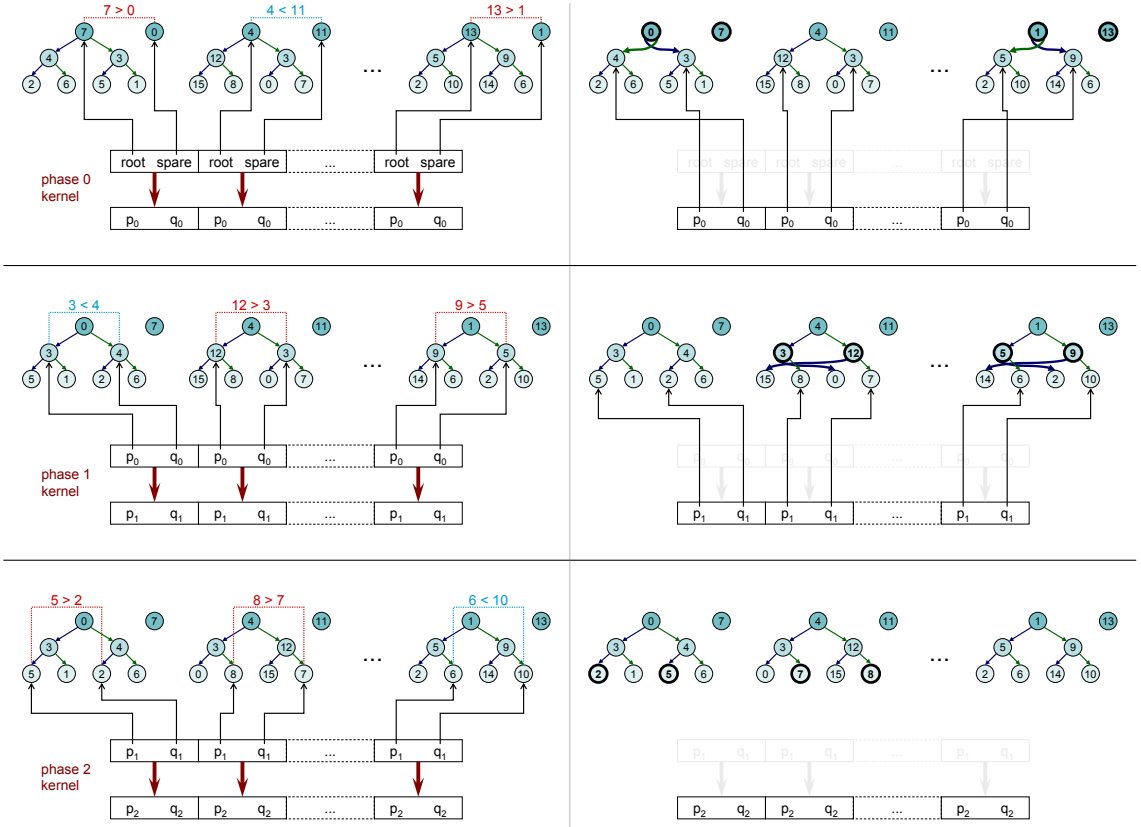



Figure 2: To execute several instances of the adaptive min/max determination algorithm in parallel, where each instance operates on a bitonic tree of $2^3$ nodes, 3 kernel invocations are required. This figure illustrates the operation of these 3 kernels. On the left, the node pointers contained in the respective input stream are shown as well as the comparisons performed by the kernel program. On the right, the node pointers written into the respective output stream are shown as well as the modifications of the child pointers and node values performed by the kernel program according to the algorithm described in Section 4.2.

described in Section 4.2), and finally writes the updated pair of node pointers **p**, **q** back to the stream. This is illustrated in Figure 2.

## 5.2 Eliminating random-access writes

Since the targeted stream architecture does not support random-access writes, we have to find a way to implement a kernel that modifies node pairs of the bitonic tree without random-access writes. This means that we can output modified node pairs from the kernel only via linear stream write. But this way we cannot write back a modified node pair to its original location where it was read. (Otherwise we would have to process the nodes in the same order as they are stored in memory, but the adaptive bitonic merge processes them in a random, data dependent order.) Of course we have to assure that subsequent stages of the adaptive bitonic merge use the modified nodes instead of the original ones, if we output the modified nodes to different locations in memory.

Fortunately the bitonic tree is a linked data structure where all nodes are directly or indirectly linked to the root (except for the spare node). This allows us to change the location of nodes in memory during the merge algorithm as long as we update the child pointers of their respective parent nodes (and keep the root and spare node of the bitonic tree at well-defined memory locations). This means that for each node that is modified during the algorithm, also its parent node has to be modified to update its child pointers.

Recall that the adaptive bitonic merge traverses the bitonic trees downwards along certain paths. If any node on that path is to be modified, also all previously visited nodes on that path have to be modified to update their child pointers. Therefore we use the following strategy to assure the correct update of child

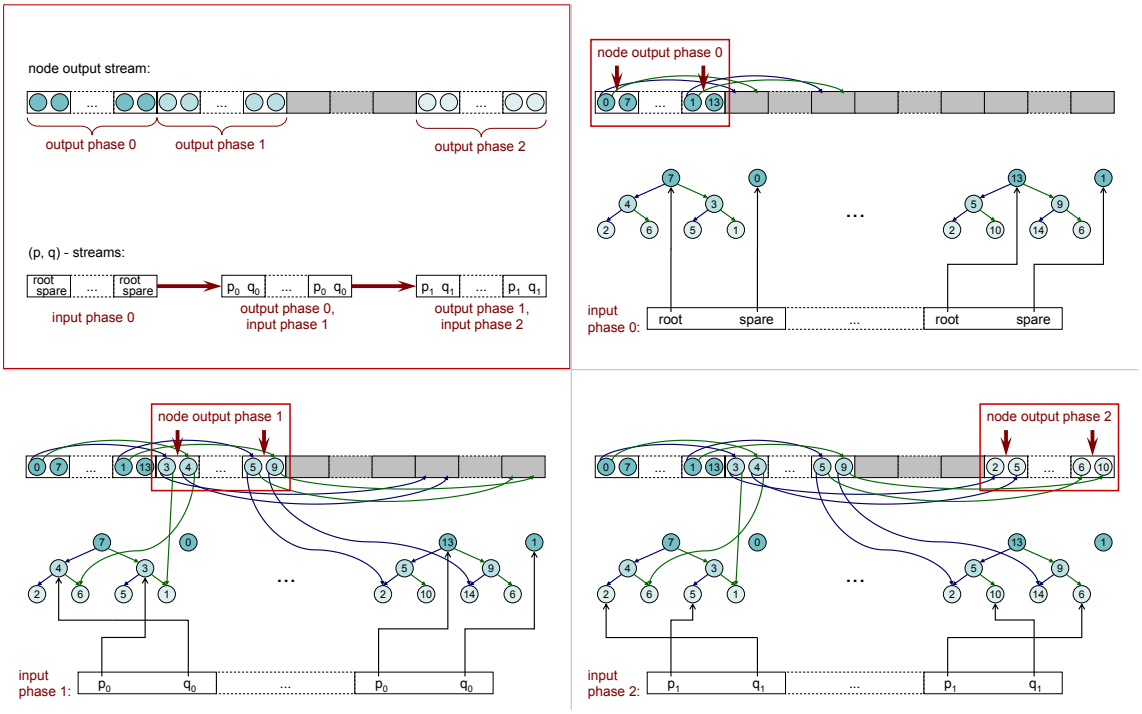



Figure 3: This figure illustrates a detail of the stream program implementation of the adaptive min/max determination algorithm that was not shown in Figure 2: how the nodes that are modified by the kernel program are written back using only linear stream writes. This is achieved by allocating a node output stream (in addition to the streams holding the node pointers **p** and **q**) and by defining in advance to which part of the stream the modified nodes will be written in each phase of the algorithm. If for example these stream parts (*substreams*) are chosen as shown in the upper left, the operation of the three phases of the algorithm for bitonic trees of $2^3$ nodes is as depicted. Note that these three phases together correspond to a single stage of the adaptive bitonic merge. The subsequent stages will write to different substreams of the same output stream. (How to choose these substreams without having to increase the total size of the stream will be explained in Section 5.3.)

pointers: We simply output every node visited during this traversal to a stream. At the same time we update the child pointers of these nodes to point to those locations where the modified child nodes will be stored in the next step of the traversal. This implies that we use a common output stream for all steps of the traversal and define in advance to which locations the modified nodes will be stored in each step. Since in a single merge stage no node is visited more than once, obviously a stream providing space for $n$ nodes (or $\frac{n}{2}$ node pairs) is sufficient for the output of all phases of that stage. Figure 3 demonstrates the operation of the stream program using the described stream output technique. More details about the actual implementation of the kernel programs will be given in Appendix A.2.

While the stream memory used for the temporary node pointers **p** and **q** may be freed and reused in each stage of the adaptive bitonic merge (see Section 5.1), it is not possible to reuse the memory of the node output stream in the sense that modified node pairs are written to the same memory locations in each stage of the merge, as this might result in overwriting nodes that are still in use. The reason for this is that a single stage of the adaptive bitonic merge does not visit all nodes of the bitonic tree (except for the last two stages); thus the output of a certain merge stage may contain nodes with children that have not yet been modified (as in Figure 3 lower right) or were modified and written to the node output stream in a previous stage.

A simple solution would be to append the output of every stage to a large stream without overwriting nodes written in previous stages. However, this would increase the memory requirement further, which might be an issue when sorting large sequences on a stream architecture with limited amount of stream memory. Therefore, we present a more light-weight memory layout for the node output stream in the following section that only requires a stream providing space for $n$ nodes (or $\frac{n}{2}$ node pairs) in total; i.e. we specify to which parts of a stream of that size the output of each merge stage should be directed such that

only those locations are overwritten that do not contain valid nodes anymore.

## 5.3 Reducing the memory overhead

As outlined in Section 5.1, on every stage $k$ of a recursion level $j$ of the adaptive bitonic sort exactly $2^{\log n-j} \cdot 2^k$ kernel instances are executed simultaneously; and since each kernel instance modifies and writes a single node pair to the stream, the output of every phase of stage $k$ consists of exactly that amount of node pairs. Therefore, for each phase we have to specify a contiguous block of stream memory (which we call *substream*) providing space for $2^{\log n-j} \cdot 2^k$ node pairs. We do this as follows: For the whole recursion level $j$, we allocate a single stream with a total size of $\frac{n}{2}$ node pairs and use certain parts of that stream as output in every phase of the algorithm as specified in Table 1.

| phase | start of substream | end of substream |
|---|---|---|
| 0 | 0 | $2^k \cdot 2^{\log n-j}$ |
| 1 | $2^k \cdot 2^{\log n-j}$ | $2^{k+1} \cdot 2^{\log n-j}$ |
| $i > 1$ | $(2^{k+i-1} + 2^k)\ 2^{\log n-j}$ | $(2^{k+i-1} + 2^{k+1})\ 2^{\log n-j}$ |

Table 1: Specification of the stream memory blocks (*substreams*) to which modified node pairs are written for each phase of stage $k$. Here the memory locations are given in the unit of *node pairs*.

This scheme is based on the following observations: In phase 0 of a stage $k$, *all* tree nodes of the levels $0, \ldots, k$ are modified and written. Thus any tree node of a level $0, \ldots, k$ that has been written previous to that phase will not be further required in subsequent phases and can be overwritten safely. (Note that according to the substream specification above, tree nodes of the levels $0, \ldots, k$ are always contained in the first $2^k \cdot 2^{\log n-j}$ node pairs of the stream.) Furthermore, in phase 1 of stage $k$, *all* tree nodes of level $k+1$ are modified and written. Thus in this phase, all previously written nodes of level $k+1$ can be overwritten.

Using this scheme, the output of the last step of the merge (which was directed to the full stream of $\frac{n}{2}$ node pairs) contains all $2^{\log n-j}$ completely modified bitonic trees of recursion level $j$ (each of which represents now a fully sorted sequence of length $2^j$) in a non-interleaved manner. This stream is then used as input for the subsequent recursion level $j+1$ of the adaptive bitonic sort. Since at the end of each recursion level all input tree nodes have been replaced by modified nodes in the output stream, it is sufficient to allocate two streams of $\frac{n}{2}$ node pairs for the whole sort algorithm and alternately use one them as output stream in each recursion level.

Figures 4 – 5 demonstrate this stream layout when the merge is performed on one bitonic tree of $2^4$ nodes (Figure 4) or simultaneously on two bitonic trees containing $2^4$ nodes each (Figure 5) assuming a sequential execution of all stages. The numbers in these tables specify the tree level of each node in the output stream (where 0 corresponds to the root); s is the spare node of the bitonic tree. While the node pairs shown in deep black are those written in the respective phase (indicated on the left), the node pairs shown in gray are the ones still accessible from previous phases. Note that the order of the nodes written in phase 0 of each stage $k$ (shown in bold font) corresponds to an in-order traversal of the $k$ upper levels of the bitonic tree.

Appendix A summarizes the whole algorithm up to this point.

| | | *output stream layout:* tree levels of node pair at stream memory location | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| stage | phase | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | **0** | **0s** | | | | | | | |
| 0 | 1 | 0s | 11 | | | | | | |
| 0 | 2 | 0s | 11 | | 22 | | | | |
| 0 | 3 | 0s | 11 | | 22 | | 33 | | |
| 1 | **0** | **10** | **1s** | | 22 | | 33 | | |
| 1 | 1 | 10 | 1s | 22 | 22 | | 33 | | |
| 1 | 2 | 10 | 1s | 22 | 22 | | 33 | 33 | 33 |
| 2 | **0** | **21** | **20** | **21** | **2s** | | 33 | 33 | 33 |
| 2 | 1 | 21 | 20 | 21 | 2s | 33 | 33 | 33 | 33 |
| 3 | **0** | **32** | **31** | **32** | **30** | **32** | **31** | **32** | **3s** |

Figure 4: Output stream layout for the last recursion level ($j = 4$) of sorting $n = 2^4$ values.

| stage | phase | *output stream layout:* tree levels of node pair at stream memory location | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 | **0** | **0s** | **0s** | | | | | | | | | | | | | | |
| 0 | 1 | 0s | 0s | 11 | 11 | | | | | | | | | | | | |
| 0 | 2 | 0s | 0s | 11 | 11 | | | 22 | 22 | | | | | | | | |
| 0 | 3 | 0s | 0s | 11 | 11 | | | 22 | 22 | | | 33 | 33 | | | | |
| 1 | **0** | **10** | **1s** | **10** | **1s** | | | 22 | 22 | | | 33 | 33 | | | | |
| 1 | 1 | 10 | 1s | 10 | 1s | 22 | 22 | 22 | 22 | | | 33 | 33 | | | | |
| 1 | 2 | 10 | 1s | 10 | 1s | 22 | 22 | 22 | 22 | | | 33 | 33 | 33 | 33 | 33 | 33 |
| 2 | **0** | **21** | **20** | **21** | **2s** | **21** | **20** | **21** | **2s** | | | 33 | 33 | 33 | 33 | 33 | 33 |
| 2 | 1 | 21 | 20 | 21 | 2s | 21 | 20 | 21 | 2s | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 |
| 3 | **0** | **32** | **31** | **32** | **30** | **32** | **31** | **32** | **3s** | **32** | **31** | **32** | **30** | **32** | **31** | **32** | **3s** |

Figure 5: Output stream layout for recursion level $j = 4$ of sorting $n = 2^5$ values. (In this case, the merge algorithm is applied to two bitonic trees. Nodes belonging to the second bitonic tree are shown in red.)

## 5.4 GPU-ABiSort in $O(\log^2 n)$ stream operations

Since each stage $k$ of a recursion level $j$ of the adaptive bitonic sort consists of $j - k$ phases, $O(\log n)$ stream operations are required for each stage. Together, all $j$ stages of recursion level $j$ consist of $\frac{1}{2}j^2 + \frac{1}{2}j$ phases in total. Therefore, the sequential execution of these phases requires $O(\log^2 n)$ stream operations per recursion level and in total $O(\log^3 n)$ stream operations for the whole sort algorithm.

While this already allows to achieve the optimal time complexity $O((n \log n)/p)$ for up to $p = n/\log^2 n$ stream processor units, we will present in the following an improved GPU-ABiSort implementation for stream architectures that allow the specification of substreams consisting of multiple separate memory blocks, which requires only $O(\log^2 n)$ stream operations for the whole sorting (and is thus theoretically capable of achieving the optimal time complexity for up to $n/\log n$ stream processor units). The reduction of the number of stream operations by the factor $O(\log n)$ is accomplished by adapting a technique from the parallel PRAC implementation of the adaptive bitonic sorting [BN89]: Instead of a completely sequential execution of all stages, we execute them partially overlapped.

By observing which tree levels are visited in each of the phases and in which phases they have been visited the last time before, we notice that phase $i$ of a stage $k$ can be executed immediately after phase $i+1$ of stage $k-1$. Therefore, we can start the execution of a new stage every other step of the algorithm (cf. [BN89]), which leads to an adaptive bitonic merge implementation in a total of $2 \log n - 1$ steps and thus in $O(\log n)$ stream operations for each of the $\log n$ recursion levels of the adaptive bitonic sort.

For such an improved implementation, the previously used specification to which memory locations modified nodes are written in each phase of the algorithm (see Table 1) is still applicable. However, instead of defining a single contiguous memory block as substream in each step of the algorithm, now multiple memory blocks together form a substream that is to be used as output stream for the corresponding stream operation. In this context, the memory blocks that form a common substream correspond to those phases that can be executed in the same step of the algorithm (and thus potentially in parallel) according to the above observation. Figure 6 shows the respective stream layout. As it can be seen there, the memory blocks belonging to a single step of the algorithm do not overlap.

| step | stages | *output stream layout:* tree levels of node pair at stream memory location | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 | 0 | **0s** | **0s** | | | | | | | | | | | | | | |
| 1 | 0 | 0s | 0s | 11 | 11 | | | | | | | | | | | | |
| 2 | 0,1 | **10** | **1s** | **10** | **1s** | | | 22 | 22 | | | | | | | | |
| 3 | 0,1 | 10 | 1s | 10 | 1s | 22 | 22 | 22 | 22 | | | 33 | 33 | | | | |
| 4 | 1,2 | **21** | **20** | **21** | **2s** | **21** | **20** | **21** | **2s** | | | 33 | 33 | 33 | 33 | 33 | 33 |
| 5 | 2 | 21 | 20 | 21 | 2s | 21 | 20 | 21 | 2s | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 |
| 6 | 3 | **32** | **31** | **32** | **30** | **32** | **31** | **32** | **3s** | **32** | **31** | **32** | **30** | **32** | **31** | **32** | **3s** |

Figure 6: Output stream layout for recursion level $j = 4$ of sorting $n = 2^5$ values (cf. Figure 5) when executing the merge stages partially overlapped.

Finally, we can summarize the complete sorting algorithm at the stream level as follows:

**GPU-ABiSort:**
for each recursion level $j$ of the adaptive bitonic sort, i.e. for $j = 1, \ldots, \log n$:
{
  $k_0 := 0$  (first active stage of a step of the merge)
  $k_1 := 0$  (last active stage of a step of the merge)
  for each step $i$ of the merge, i.e. for $i = 0, \ldots, 2j - 2$:
  {
    if $i$ is even (and $i > 0$): increment $k_0$ by 1
    if $i >= \log n$: decrement $k_1$ by 1
    the substream to be used as output in this step is defined by the memory blocks
      that are, according to Table 1, associated with the following phases:
      stage $k_0$ phase $i - 2k_0$, stage $k_0 + 1$ phase $i - 2(k_0 + 1)$, ..., stage $k_1$ phase $i - 2k_1$
    invoke a kernel on all elements of that substream (which performs a step of the
      *adaptive min/max determination* and updates child pointers, see Section 5.2)
  }
}

# 6 GPU-specific details

## 6.1 Distinctness of input and output streams

In the preceding section, we assumed that it is possible to use the same stream as input and output of a stream operation. However, on current GPUs input and output streams must always be distinct (and it is currently not sufficient to use just distinct substreams from the same stream for input and output).

For the stream holding the temporary node pointers **p** and **q** (see Section 5.1) we apply the *ping-pong* technique commonly used in GPU programming: We allocate two such streams and alternately use one of them as input and the other one as output stream.

For the stream holding the modified node pairs (see Section 5.2) this technique cannot be applied since not all stream elements are modified in each step of the algorithm. Therefore, in our current implementation, we allocate two such streams and permanently use one of them as input and the other one as output stream. After each step of the algorithm, all nodes that have just been written to the output stream are simply copied back to the input stream.

## 6.2 GPU-ABiSort using a 2D stream layout

For most applications, we can expect that the input sequence will be longer than the maximum allowed size of a 1D stream on current GPUs (see Section 3.2). Therefore, in our GPU implementation we have to pack our stream contents (i.e. the node pointers and the modified node pairs) into 2D streams. We tested two different possibilities to map the 1D stream contents to 2D streams, a row-wise mapping as well as a Z-order mapping, which will be described in the following.

### 6.2.1 Row-wise 1D-2D mapping

The simplest solution is a row-wise mapping, i.e. if $a$ is an index corresponding to the location of a node in the 1D stream, this node will be mapped to the location $(a \mod w, \lfloor a/w \rfloor)$ in the 2D stream (where $w$ specifies the width of the 2D stream). We thereby assume that the width $w$ is a power of two.

To be able to use the stream program described in Section 5.4 with such a 2D stream without further modifications, it is necessary that the contiguous memory blocks which define a substream correspond to rectangular blocks of the 2D stream after the mapping. Our specification of these memory blocks according to Table 1 meets these demands: The length $l$ of each block is a power of two. Furthermore, the start location $s$ of each block is a multiple of $l$. Hence, if $l \leq w$, then $w$ is obviously a multiple of $l$, just like the start location $s$ and the end location $s + l$ of the block, and thus the block is located completely within

a single line of the 2D stream after the mapping. And if $l \geq w$, then $l$ as well as $s$ are multiples of $w$, and thus the block spans the complete lines $\frac{s}{w}, \ldots, \frac{s}{w} + \frac{l}{w} - 1$ of the 2D stream after the mapping.

#### 6.2.2 Z-order 1D-2D mapping

A disadvantage of the row-wise mapping described above is that it is not very GPU-cache-friendly. This has the following background: In the previous description of our approach (and also in the pseudo code in Appendix A) we clearly differentiated between streaming reads and random-access reads. In the original stream programming model this distinction was motivated by the fact that streaming reads can be implemented more efficiently in hardware since their memory access patterns are fully known in advance and thus for these read accesses no conventional cache logic is needed that tries to predict which data will be required in future memory accesses based on heuristics [KRD*03]. However, current GPUs (or more precisely their fragment processor units) do not differentiate between streaming reads and random-access reads and thus use the same cache logic for both types of read accesses [BFH*04]; and this cache logic is obviously optimized for the use case of accessing 2D texture data during rasterization, for which in general a cache architecture where each cache block holds a square or near-square region of the texture data is favorable [HG97]. As a consequence, for streaming reads from a rectangular memory block (substream) of a 2D stream the maximum read bandwidth is only achieved if this substream has a square or near-square shape (as it was demonstrated by performance tests in recent work [GRHM05, GGKM05]).

Since this is generally not the case for the substreams obtained by using the row-wise mapping described above, we propose the usage of an alternative, GPU-cache-optimized mapping between 1D and 2D streams where the 2D space is mapped to 1D along a space-filling curve known as Z-order or Morton order [Mor66]: Assuming that a 1D integer index $a$ is given, which has the bit representation $(a_{31}, \ldots, a_1, a_0)$. Then this index is mapped to the 2D index $(a_x, a_y)$ where $a_x$ has the bit representation $(a_{30}, \ldots, a_2, a_0)$ and $a_y$ has the bit representation $(a_{31}, \ldots, a_3, a_1)$. This mapping has some nice properties, which are easy to see:

- For any $a$, the 1D index $2 \cdot a$ is mapped to the 2D index $(2 \cdot a_y, a_x)$.
- For any $s$ that is a power of two and any $a < s$, $s + a$ is mapped to $(s_x + a_x, s_y + a_y)$.
- For any $l$ that is a power of two, $l' = l - 1$ is mapped to $(l'_x, l'_y)$, where $(l'_x + 1) \cdot (l'_y + 1) = l$ and either $l'_x + 1 = l'_y + 1$ or $l'_x + 1 = 2 \cdot (l'_y + 1)$.

Since according to the memory layout specified in Table 1 the length $l$ of each memory block is a power of two and its start location $s$ is a multiple of $l$, it follows from the last two propositions that the interval of 1D indexes $\{s, \ldots, s + l'\}$ with $l' = l - 1$ is mapped to the contiguous 2D block $\{s_x, \ldots, s_x + l'_x\} \times \{s_y, \ldots, s_y + l'_y\}$, which has square ($l'_x + 1 = l'_y + 1$) or near-square ($l'_x + 1 = 2 \cdot (l'_y + 1)$) shape.

Since it would be too expensive to calculate the mapping between 1D and 2D indexes in the GPU kernel programs (especially on current GPUs that do not support integer bit operations), we process and store all addresses in the kernel programs directly in form of 2D indexes (where we represent a 2D index by two 16 bit integer values packed into a 32 bit field). Using these 2D indexes, the address calculations required in the kernel programs are less trivial, but can easily be resolved by utilizing the propositions above.

## 7 GPU-ABiSort optimizations

In the following, we will describe how the practical running time of our asymptotically optimal algorithm presented in Sections 5 – 6 can be further improved. Therefor we will combine the adaptive bitonic sort and the adaptive bitonic merge with respective algorithms specifically tuned to small sequence lengths. This follows the common practice of combining sorting algorithms with optimal asymptotic complexity (like merge or heap sort) with sorting algorithms that are fast for small sequence lengths (like insertion sort).

### 7.1 Optimized replacement for the first recursion levels of the sort

For parallel merge-based sorting on a PRAM architecture with less processors than values to sort ($p < n$), it is a common technique that in a first step, $p$ blocks of $n/p$ values are sorted locally (i.e. each processor sorts $n/p$ values using a standard sequential algorithm). This way, the actual parallel merging algorithm can start by merging pairs of sorted blocks, each containing $n/p$ elements, instead of pairs of single elements.

The same technique can also be applied to a stream architecture by implementing such a *local sort* as a kernel program. However, since there is no random write access to non-temporary memory from a kernel,

the number of values that can be sorted locally by a kernel is restricted by the number of temporary registers as well as by the maximum data size that can be written to the output stream by a single kernel instance.

On recent GPUs, the maximum output data size of a kernel is $16 \times 32$ bit. As we sort value/pointer pairs (consisting of $2 \times 32$ bit each) in our implementation, we start with with a local sort of 8 value/pointer pairs per kernel. As known from standard sequential sorting approaches, $O(n^2)$ algorithms are in practice often faster than asymptotically optimal algorithms for sorting such a small number of values. In our GPU implementation, the local sort is performed in a single stream operation by an efficient odd-even transition sort implementation. (The comparison order of odd-even transition sort, that makes it also applicable as sorting network, allows for better SIMD optimizations than those of several other $O(n^2)$ sorting algorithms.)

After the local sort, a further stream operation converts the resulting sorted sequences of length 8 pairwise to bitonic trees, each containing 16 nodes. Thereafter, the GPU-ABiSort approach can be applied as described in Section 5, starting with $j = 4$. Hence we have replaced the recursion levels $j = 1, \ldots, 3$ of the adaptive bitonic sort by an optimized sort for small sequence lengths.

## 7.2 Optimized replacement for the last stages of each merge

In addition, the remaining recursion levels of the sort can also be accelerated. For this purpose, we can use a bitonic merge implementation which is faster than the adaptive bitonic merge implementation from Section 5 for a certain (small) sequence length $n'$ but not necessarily asymptotically optimal. Such a specialized merge implementation for a certain sequence length $n'$ can be used to speed up the merge executions for all sequence lengths $n \geq n'$. This is based on the recursive definition of bitonic merging: Bitonic merging of $n'$ values is a subroutine of the bitonic merge of $n > n'$ values. Consequently, the last $\log n'$ stages of the adaptive bitonic merge can be replaced by an alternative bitonic merge implementation for $n'$ values.

While it would be possible to use one of the previous GPU implementations of the bitonic sorting network for this purpose (see Section 2.2), we currently use an own implementation of the (non-adaptive) bitonic merge for the fixed sequence length $n' = 16$ instead, which is performed in a single stream operation. (To meet the afore-mentioned per-kernel output size restriction of current GPUs, each bitonic sequence of length 16 is processed by two kernel instances: one of them outputs the merged lower half $p'$ and the other one the merged upper half $q'$.)

The optimized merge approach for arbitrary sequence lengths $n$ starts with adaptive bitonic merging exactly as described in Section 5 with the only difference that the last 4 stages are left out. In consequence, if the stages are executed overlapped as described in Section 5.4, the total number of steps reduces to $2 \log n - 5$, and in the last 3 remaining steps only a reduced number of node pairs has to be processed (only those node pairs belonging to the remaining $\log n - 4$ stages). As an example, Figure 7 shows for the last recursion level of sorting $n = 2^6$ values, which node pairs are processed by the adaptive bitonic merge algorithm when taking the reduced number of stages into account. After this incomplete adaptive bitonic merge, the contents of the output stream do not yet correspond to an in-order node traversal of the bitonic tree, as it was the case with the full adaptive bitonic merge. Thus, before the optimized bitonic merge for $n' = 16$ can be applied, its input sequences of length 16 have to be determined by an in-order node traversal starting simultaneously from all output nodes of phase 0 of the last executed stage of the adaptive bitonic merge. Implemented as a kernel program, this traversal can also be performed in a single stream operation. Then, the optimized bitonic merge for $n' = 16$ is executed, and finally, the merged 16-value sequences are converted back to bitonic trees, as it was done after the local sort described in Section 7.1.

| | | *output stream layout:* tree levels of node pair at stream memory location | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| step | stages | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ... | 17 | 18 | 19 | ... |
| 0 | 0 | **0s** | | | | | | | | | | | | | | | | |
| 1 | 0 | 0s | 11 | | | | | | | | | | | | | | | |
| 2 | 0,1 | **10** | **1s** | | 22 | | | | | | | | | | | | | |
| 3 | 0,1 | 10 | 1s | 22 | 22 | | 33 | | | | | | | | | | | |
| 4 | 0,1 | 10 | 1s | 22 | 22 | | 33 | 33 | 33 | | 44 | | | | | | | |
| 5 | 0,1 | 10 | 1s | 22 | 22 | | 33 | 33 | 33 | | 44 | 44 | 44 | | 55 | | | |
| 6 | 1 | 10 | 1s | 22 | 22 | | 33 | 33 | 33 | | 44 | 44 | 44 | | 55 | 55 | 55 | |

Figure 7: Output stream layout for adaptive bitonic merging of $2^6$ values if an optimized bitonic merge of $2^4$ values is applied afterwards.

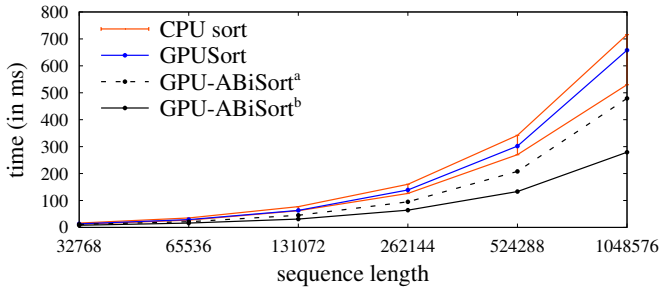
# 8 Results

The usual application scenario of a sorting algorithm is the sorting of arbitrary data (e.g. records of a database) based on a sort key, in our case a 32-bit floating point value. On the CPU this is implemented efficiently by sorting an array consisting of value/pointer pairs, where the value is used as sort key and the pointer refers to the associated data. (An alternative would be to sort a pure pointer array whose entries point to records containing the sort key, but according to own tests this alternative has shown to be clearly less efficient in general.) We also implemented GPU-ABiSort in such a way that the input and the final output of the sorting is given as an array of value/pointer pairs. Since we can assume (without loss of generality) that all pointers in the given array are unique, we can use these pointers at the same time as secondary sort keys for the adaptive bitonic merge.

We compared the performance of GPU-ABiSort with sorting on the CPU using the C++ STL sort function (an optimized quick sort implementation) as well as with the (non-adaptive) bitonic sorting network implementation on the GPU by Govindaraju et al., called GPUSort [GRHM05]. Our GPU-ABiSort implementation includes the optimizations covered in Section 7. Contrary to the CPU STL sort, the timings of GPU-ABiSort do not vary significantly dependent on the data to sort (because the total number of comparisons performed by the adaptive bitonic sorting is not data dependent). For the following timings, we use value/pointer pairs with uniformly distributed random floating point sort keys.

Table 2 shows the timings on an AGP bus PC system with an AMD Athlon-XP 3000+ CPU and an NVIDIA GeForce 6800 Ultra GPU with 256 MB memory. On this system, our approach achieves 1.9 – 2.6 times speed-up compared to CPU sort for $n \geq 2^{17}$ (using the Z-order 1D-2D mapping presented in Section 6.2.2) and up to 2.4 times speed-up compared to GPUSort (in its original implementation by [GRHM05]).

Note that also GPUSort was optimized with respect to cache efficiency [GRHM05].[1] Nevertheless, on the GeForce 6800 GPU our approach beats GPUSort even if we use the non-cache-optimized, row-wise 1D-2D mapping described in Section 6.2.1.




| $n$ | CPU sort | GPUSort[GRHM05] | GPU-ABiSort[a] | GPU-ABiSort[b] |
|---|---|---|---|---|
| 32768 | 12 – 16 ms | 13 ms | 11 ms | 8 ms |
| 65536 | 27 – 35 ms | 29 ms | 21 ms | 16 ms |
| 131072 | 62 – 77 ms | 63 ms | 45 ms | 31 ms |
| 262144 | 126 – 160 ms | 139 ms | 95 ms | 64 ms |
| 524288 | 270 – 342 ms | 302 ms | 208 ms | 133 ms |
| 1048576 | 530 – 716 ms | 658 ms | 479 ms | 279 ms |

Table 2: Timings on a GeForce 6800 system: a) using row-wise 1D-2D mapping (see Section 6.2.1), b) using Z-order 1D-2D mapping (see Section 6.2.2).

[1] However, the authors use a different strategy to achieve the cache efficiency. They employ a row-wise 1D-2D address mapping similar to the one presented in Section 6.2.1, but combined that with a decomposition of streams into square tiles of size $B \times B$ as follows: Based on the observation that the cache usage is non-optimal for streaming reads from non-square substreams (cf. Section 6.2.2), they split non-square substreams into multiple smaller substreams such that no substream covers more than one $B \times B$ sized tile (ideally $B \times B$ should correspond to the size of a GPU cache block). Then all substreams lying within a single $B \times B$ tile are processed consecutively. On a GeForce 7800 this technique achieves in fact near optimal read bandwidth with the parameter $B = 64$, as it was shown by experiments in [GGKM05]. However, in general it is hard to guess the optimal value for this parameter since no information about cache characteristics is disclosed by GPU vendors. (In contrast our Z-order based 1D-2D mapping is a cache-oblivious strategy, i.e. it is independent of hardware configuration parameters such as the size of the GPU cache or of a cache block.) The available GPUSort implementation simply uses the parameter $B = 64$ for any GPU architecture independent of its actual cache characteristics, which might be one of the reasons why GPUSort performs so much worse on the GeForce 6800 than on the GeForce 7800, showing a notably larger performance difference between these GPUs than our and several other approaches.

| $n$ | CPU sort | GPUSort[GRHM05] | GPU-ABiSort |
|---|---|---|---|
| 32768 | 9 – 11 ms | 4 ms | 5 ms |
| 65536 | 19 – 24 ms | 8 ms | 8 ms |
| 131072 | 46 – 52 ms | 18 ms | 16 ms |
| 262144 | 98 – 109 ms | 38 ms | 31 ms |
| 524288 | 203 – 226 ms | 80 ms | 65 ms |
| 1048576 | 418 – 477 ms | 173 ms | 135 ms |

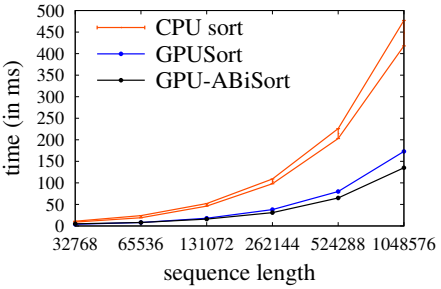



Table 3: Timings on a GeForce 7800 system (using Z-order 1D-2D mapping).

We also tested our implementation on a PCI Express bus PC system with an AMD Athlon-64 4200+ CPU and an NVIDIA GeForce 7800 GTX GPU with 256 MB memory, see Table 3 for the results. Even though this system has not only a more recent GPU but also a CPU of a newer generation, the speed-up of our approach compared to CPU sorting has significantly increased to a 3.1 – 3.5 times speed-up for $n \geq 2^{17}$. Furthermore, up to the maximum tested sequence length $n = 2^{20}$, our approach is up to 1.3 times faster than GPUSort, and as expected this speed-up is increasing with the sequence length $n$.

The timings of the GPU approaches assume that the input data is given in GPU memory as it is for example the case when the sorting is needed for a GPU-based application such as [PDC*03] or [KSW04]. When embedding the GPU-based sorting into an otherwise purely CPU-based application, the input data has to be transfered from CPU to GPU memory before the application of our approach, and afterwards the output data has to be transfered back from GPU to CPU memory. However, the overhead of this transfer is usually negligible compared to the achieved sorting speed-up: According to our measurements, the transfer of $2^{20}$ (1.048.576) value/pointer pairs from CPU to GPU and back takes in total roughly 100 ms on our AGP bus PC and roughly 20 ms on our PCI Express bus PC.

# 9 Conclusions and future work

We presented a novel approach for parallel sorting on stream architectures. As opposed to any previous sorting approach on stream processors, it achieves the optimal time complexity $O((n \log n)/p)$. Furthermore, our approach performs also very well in practice, which is caused by a well-chosen stream memory layout and by several optimizations we incorporated into our approach.

We implemented our approach on modern programmable graphics hardware (GPUs). The timings we obtained with this implementation are very promising, especially with regard to the performance improvements that can be expected with upcoming GPU generations. The implementation of our approach has shown that optimal parallel sorting on stream processors is indeed very efficient in practice.

As it was said, in our implementation we assumed that the length of the input sequence is a power of two (as it was also done in GPU sorting network implementations). However, it should certainly be possible to incorporate the extension of the adaptive bitonic sorting approach to non-power-of-two sequence lengths by the use of pruned bitonic trees [BN89] into our approach. The efficient implementation of such an extension remains a task of future work.

Finally, it would be interesting to explore to what extent the tree traversal and node modification techniques developed in this approach can be helpful for the adaption of other adaptive or hierarchical algorithms to the stream programming domain.

## Acknowledgments

This work was partially supported by the EU under the Sixth Framework Programme (INCO-CT2005-013408).

# A Implementation in detail

In this appendix, we present the complete pseudo code of the unoptimized version of GPU-ABiSort running in $O(log^3 n)$ stream operations (as of Section 5.3). It contains neither the reduction of the number of stream operations by the factor $O(\log n)$ described in Section 5.4 nor the runtime optimizations presented in Section 7. Since the pseudo code represents a general stream program, also the GPU-specific implementation details described in Section 6 are not covered here.

Some stream program compilers such as Brook for GPUs [BFH*04] assure that all read accesses initiated by a certain kernel program are carried out before any write access by this kernel to the same stream. We assume the same semantic in our pseudo code, thus hiding the implementation details necessary to avoid possible conflicts when using the same stream as input and output of a certain kernel (as described in Section 6.1 in a GPU-specific context).

Since the reader might not be familiar with the syntax of stream programming languages like Brook, we use a syntax much closer to the C++ standard for the given pseudo code, augmented by just a few stream programming specific constructs and keywords, which are shortly described in the following text.

## A.1 GPU-ABiSort main routine

**Basic data types (Listing 1):** Let us assume that the sequence to sort is given as a stream of data values of an arbitrary base type value_t. We further assume that a comparison operator > is given for this base type that defines a total order on the data. In our implementation, we use a base type value_t consisting of two fields: a floating point primary sort key and a unique id, which is used as secondary sort key to assure the total order required by the adaptive bitonic sort algorithm. The latter field can at the same time be used as a pointer to an arbitrary data record associated with the sort key (see Section 8).

A bitonic tree node (type node_t) consists of a data value (of type value_t) and pointers to the left and right child node. Instead of real pointers we use indexes (of an integral type index_t) in this implementation. For leaf and spare nodes, these indexes are not used and can be set to arbitrary values.

```
// let value_t be the base type of the data to sort, e.g.
struct value_t {
   float key;  // primary sort key
   void* id;   // unique id (or data pointer) used as secondary sort key
};

// including usual comparison operators such as
bool operator > (value_t p, value_t q) { return p.key > q.key ||
                                              (p.key == q.key && p.id > q.id); }

typedef int index_t;

// bitonic tree node:
struct node_t {
   value_t value;
   index_t left;   // index of left son
   index_t right;  // index of right son
};
```

Listing 1: Pseudo code of basic type definitions for GPU-ABiSort

**GPU-ABiSort main routine (Listing 2):** Let us assume that a sub-routine GPUABiMerge takes $2^{\log n - j}$ bitonic trees, each consisting of $2^j$ nodes, as input and simultaneously applies the adaptive bitonic merge algorithm to these trees. For sorting a sequence consisting of $n$ values, the main routine GPUABiSort has to call this sub-routine for each recursion level $j = 1, \ldots, \log n$ of the adaptive bitonic sort (see Section 5.1).

The input bitonic trees are passed to GPUABiMerge as a stream of bitonic tree nodes of the aforementioned base type node_t (parameter bitonicTrees). More precisely, to simplify the implementation of GPUABiMerge (that will be described in Appendix A.2) we assume that this parameter is a stream of $2n$ nodes, where the first $n$ nodes are reserved as a working space for GPUABiMerge, and the input bitonic trees are contained in the second $n$ nodes.

Since after applying our adaptive bitonic merge implementation, the order of the nodes of the $2^{\log n - j}$ modified bitonic trees in stream memory corresponds to an in-order traversal of these trees (see Section 5.3),

the data values in that stream can be interpreted as $2^{\log n - j}$ fully sorted sequences. Let us assume that the sub-routine GPUABiMerge returns these sorted sequences simply as a stream of $n$ data values of base type value_t. We further assume that the sequences have been sorted with alternating sorting directions. To use the resulting sequences as input for the next recursion level of the adaptive bitonic sort, we have to reinterpret them as bitonic trees. This can be accomplished by copying (or directing) the output of GPUABiMerge to the data values contained in the second half of the bitonicTrees stream while the left and right child indexes in this stream area are left unmodified. Only in the beginning, these left and right child indexes have to be initialized such as if the stream represents a single large bitonic tree with all nodes stored in order. (Since the child indexes of spare nodes are irrelevant, this stream can also be interpreted as representing multiple bitonic trees.)

**Notation:** The data type **stream**<t> denotes a stream with elements of base type t. In analogy to C arrays, s[i] refers to the element from stream s with index i. In addition, s[a .. b] denotes a substream of s that is defined by the elements of s ranging from index a up to index b, inclusively. Furthermore, for a stream s of type **stream**<node_t>, s.value denotes the substream of type **stream**<value_t> that consists of just the value components in s. (Accessing this kind of substream can be implemented on a stream architecture in several ways, e.g. by *striding* or using read/write masks.)

Using these notations the line bitonicTrees[n .. 2 * n − 1].value = sourceData in the pseudo code is equivalent to writing bitonicTrees[i].value = sourceData[i − n] inside the for-loop over $j = 1, \ldots, \log n$. Note also that – unlike the standard C++ semantic – passing streams as parameters or return values might be interpreted as call-by-reference throughout the given pseudo code, thus the assignment of streams (=) does not necessarily need to be implemented as copy operations.

```
// (GPUABiMerge implementation: see Appendix A.2)
stream<value_t> GPUABiMerge (int n, int j, stream<node_t> bitonicTrees);


// GPU-ABiSort - unoptimized version in O(log^3 n) stream operations:
stream<value_t> GPUABiSort (int n, stream<value_t> sourceData)
{
  // create a stream providing space for 2 * n tree nodes...
  stream<node_t> bitonicTrees (2 * n);

  // ...and initialize its second half with the source data and indexes of left and right sons
  //   set as if the stream represents a large balanced tree with all nodes stored in order
  for (int i = n; i < 2 * n; i++) {
    bitonicTrees[i].left  = i - ((i + 1) & ~i) / 2;
    bitonicTrees[i].right = i + ((i + 1) & ~i) / 2;
  }
  bitonicTrees[n .. 2 * n - 1].value = sourceData;

  // for each recursion level of the adaptive bitonic sort:
  for (int j = 1; j <= log(n); j++) {
    // merge 2^(log(n) - j) bitonic trees and finally write the merged
    // data back to the substream bitonicTrees[n .. 2 * n - 1].value
    bitonicTrees[n .. 2 * n - 1].value = GPUABiMerge(n, j, bitonicTrees);
  }

  // the result of the last GPUABiMerge contains the fully sorted data
  return bitonicTrees[n .. 2 * n - 1].value;
}
```

Listing 2: Pseudo code of the GPU-ABiSort main routine

## A.2 GPU-ABiMerge sub-routine

Before describing the sub-routine GPUABiMerge at the stream level, we will describe the implementation of the required kernel functions, which perform a phase of the *adaptive min/max determination* algorithm and update child pointers of the bitonic tree nodes.

**Phase 0 kernel (Listing 3):** According to the *adaptive min/max determination* algorithm (see Section 4.2), the input of a kernel instance in phase 0 consists of a root and a spare node from a subtree of the given bitonic tree. According to the recursion scheme of the adaptive bitonic merge (see Section 4.1), the root nodes of these subtrees are defined as the children of the root nodes from the previous stage of

the algorithm, which is exactly the output of phase 1 of that stage. Since we assume that the output of that phase was written linearly to a stream (of type **stream**<node_t>), the input root nodes of phase 0 can be read linearly (i.e. via stream read) from that stream. Likewise, also the input spare nodes can be read linearly by the kernel since they correspond to the root and spare nodes of the previous stage according to the recursion scheme and thus to the output of phase 0 of the previous stage. Since no child pointers are required for the spare nodes, we assume that the input spare nodes as well as the output nodes of phase 0 are provided as streams of type **stream**<value_t>, i.e. in contrast to the subsequent phases we do not need to output (or update) child pointers in phase 0. Note that for the nodes written in phase 0, child pointers are required only after the last stage of the merge algorithm when these nodes are used as input for the next recursion level of the sort algorithm; but in this case our implementation of the GPUABiSort function (see Appendix A.1) already assures the use of correct child pointers.

Besides the modified tree nodes, each kernel instance further outputs a pair of node pointers to a stream, which is to be used by the kernel instances of the next phase as input stream as described in Section 5.1. In this implementation we use a stream of type **stream**<index_t>, where a pair of successive elements represents the two node pointers that were called **p** and **q** in the *adaptive min/max determination* algorithm presented in Section 4.2.

**Notation:** In the pseudo code, the body of the **kernel** function describes the operation performed by each kernel instance. The keyword **instance_index** denotes the index of the respective kernel instance. Recall that from inside the kernel body, the access to streams is restricted (especially no random access write is possible). Therefore, we distinguish the following access types for the stream parameters of a kernel: The keyword **out** marks output streams, to which the output of all kernel instances is written linearly by the use of the command push_onto_stream. The keyword **in** marks input streams, from which the input of all kernel instances is read linearly by the use of the command read_from_stream. The keyword **gather** (only used in Listing 4) marks another kind of input stream, a so-called *gather stream*, which can be accesses randomly – but read-only – from the kernel using the C array syntax.

```
// phase 0 kernel:
kernel void phase0 (out stream<index_t> pqidxOutStream, out stream<value_t> nodeValueOutStream,
                    in stream<node_t> rootInStream, in stream<value_t> spareValueInStream,
                    const int numInstancesPerTree)
{
   // alternating sorting direction  (where isOdd(x) = x & 1)
   bool reverseSortDir = isOdd(instance_index / numInstancesPerTree);

   node_t  root       = read_from_stream(rootInStream);
   value_t spareValue = read_from_stream(spareValueInStream);

   if ((root.value > spareValue) != reverseSortDir) {
      swap(root.value, spareValue);
      swap(root.left,  root.right);
   }

   push_onto_stream(pqidxOutStream, root.left);    // new p index
   push_onto_stream(pqidxOutStream, root.right);   // new q index

   push_onto_stream(nodeValueOutStream, root.value);
   push_onto_stream(nodeValueOutStream, spareValue);
}
```

Listing 3: Pseudo code of the phase 0 kernel called by GPU-ABiMerge

**Phase $i > 0$ kernel (Listing 4):** The kernel phasei implements any phase $i > 0$ of the *adaptive min/max determination* algorithm. At first each kernel instance recovers the **p** and **q** indexes from the previous phase of the algorithm by reading them back from the pqidx-stream where they have been stored. Then these indexes are used to read the actual tree nodes p and q via a *gather* access from the stream containing the bitonic tree nodes. After these nodes have been compared and possibly modified, the **p** and **q** indexes to be used in the next phase of the algorithm are written to the pqidx-stream. According to the algorithm described in Section 4.2, the new **p** and **q** indexes are set either to the left or to the right child node indexes. Afterwards, these child node indexes have to be updated by the kernel, since they point to those nodes that will be replaced by modified nodes in the next phase of the algorithm. Note that in the next phase, the modified nodes will be written linearly to a stream (via the push_onto_stream command) and thus their

destination will be determined automatically by the stream hardware by iterating through a (sub)stream. To determine these destination indexes in advance in the current phase and to update the child node indexes accordingly, the same iterator mechanism can be used: We use a so-called *iterator stream*, which is a read-only stream containing a linear ascending sequence of indexes. For such an iterator stream, the hardware can realize the read_from_stream command using the iterator unit only, i.e. without memory lookups.

**Notation:** The type **iter_stream**<t> denotes an iterator stream containing indexes of base type t. Note that despite of the **if** statement, the same number of read and write accesses (from **in**/**out** streams) is performed on every control path of the kernel program, as it is required for a kernel implementation.

```
// phase i > 0 kernel:
kernel void phase1 (out stream<index_t> pqidxOutStream, out stream<node_t> nodeOutStream,
                    in stream<index_t> pqidxInStream, gather stream<node_t> bitonicTrees,
                    in iter_stream<index_t> indexGenerator, const int numInstancesPerTree)
{
   // alternating sorting direction
   bool reverseSortDir = isOdd(instance_index / numInstancesPerTree);

   index_t pidx = read_from_stream(pqidxInStream);
   index_t qidx = read_from_stream(pqidxInStream);
   node_t p = bitonicTrees[pidx];
   node_t q = bitonicTrees[qidx];

   if ((p.value > q.value) != reverseSortDir) {
      swap(p.value, q.value);
      swap(p.left, q.left);

      push_onto_stream(pqidxOutStream, p.right);  // new p index
      push_onto_stream(pqidxOutStream, q.right);  // new q index

      // update child pointers of p, q
      p.right = read_from_stream(indexGenerator);
      q.right = read_from_stream(indexGenerator);
   } else {
      push_onto_stream(pqidxOutStream, p.left);  // new p index
      push_onto_stream(pqidxOutStream, q.left);  // new q index

      // update child pointers of p, q
      p.left = read_from_stream(indexGenerator);
      q.left = read_from_stream(indexGenerator);
   }

   push_onto_stream(nodeOutStream, p);
   push_onto_stream(nodeOutStream, q);
}
```

Listing 4: Pseudo code of the phase $i > 0$ kernel called by GPU-ABiMerge

**GPU-ABiMerge sub-routine (Listing 5):** The merge algorithm for $2^{\log n - j}$ given bitonic trees is performed in $j$ stages and each stage in $j - k$ phases (see Section 5.1). The output of the last phase (i.e. phase 0 of stage $j$) finally contains the merged sequences that will be returned by this function. Table 1 in Section 5.3 specifies the substreams to be used as output streams for the kernels of each phase. Recall that we reserved the first $n$ nodes of stream bitonicTrees as a temporary workspace for this function (instead of using an additional stream of $n$ nodes as suggested in Section 5.3). As already mentioned, the input streams of phase 0 are exactly those substreams that form the output streams of phase 0 and 1 of the previous stage. The only exception is phase 0 of the first stage, where the input should consist of the root and spare nodes of the bitonic trees given in the second half of the stream bitonicTrees. Since according to the GPUABiSort implementation presented in Appendix A.1 the nodes of the bitonic tree are stored in-order in the given input stream, the root and spare nodes are contained at well-defined locations in that stream: Since each bitonic tree consists of $2^j$ nodes, the $2^{j-1}$th node of each tree is its root node, and the $2^j$th node of each tree is its spare node.

**Notation:** For simplicity, we list the root and spare nodes using a set notation at the beginning of the pseudo code and assign them to those substreams that are used in stage 0 phase 0 as input streams. (As an alternative, the corresponding kernel program could be implemented in such a way that it reads these root and spare nodes directly from the given locations by means of *striding*: each kernel instance would have

to skip $2^{j-1}-1$ stream nodes, read the root node from the stream, skip again $2^{j-1}-1$ stream nodes, and finally read the spare node from the stream.)

```
// GPU-ABiMerge - unoptimized version in O(log^2 n) stream operations:
stream<value_t> GPUABiMerge (int n, int j, stream<node_t> bitonicTrees)
{
   int numTrees = 1 << (log(n) - j);     // number of given bitonic trees
   int numPairsPerTree = 1 << (j - 1);   // number of node pairs per input tree

   // initialize root and spare input streams of stage 0 phase 0:
   int len = numTrees;
   bitonicTrees[len .. 2 * len - 1] = { bitonicTrees[n + 1 * numPairsPerTree - 1],
                                        bitonicTrees[n + 3 * numPairsPerTree - 1],
                                        ... };
   bitonicTrees[0 .. len - 1].value = { bitonicTrees[n + 2 * numPairsPerTree - 1].value,
                                        bitonicTrees[n + 4 * numPairsPerTree - 1].value,
                                        ... };

   for (int k = 0; k < j; k++) {  // stage

      // length (i.e. number of node pairs) of all substreams in this stage, cf. Table 1
      int len       = (1 << k) * numTrees;
      // start (node pair index) of phase 1 substream, cf. Table 1
      int nextStart = (1 << k) * numTrees;

      // create a stream providing space for 2 * len indexes
      stream<index_t> pqidxStream (2 * len);

      phase0(pqidxStream, bitonicTrees[0 .. 2 * len - 1].value,
             bitonicTrees[len .. 2 * len - 1], bitonicTrees[0 .. len - 1].value,
             1 << k);

      int start = nextStart;
      for (int i = 1; i < j - k; i++) {  // phase

         // start (node pair index) of phase i+1 substream, cf. Table 1
         int nextStart = ((1 << (k + i)) + (1 << k)) * numTrees;

         phasel(pqidxStream, bitonicTrees[2 * start .. 2 * (start + len) - 1],
                pqidxStream, bitonicTrees,
                iter_stream<index_t>(2 * nextStart .. 2 * (nextStart + len) - 1),
                1 << k);

         start = nextStart;
      }
   }

   return bitonicTrees[0 .. n - 1].value;
}
```

Listing 5: Pseudo code of the GPU-ABiMerge sub-routine

# References


[Akl90] AKL S. G.: *Parallel Sorting Algorithms*. Academic Press, Inc., Orlando, FL, USA, 1990.

[AKS83] AJTAI M., KOMLÓS J., SZEMERÉDI E.: An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC '83)* (1983), pp. 1–9.

[Ama05] AMARASINGHE S.: Multicores from the compiler's perspective: A blessing or a curse? In *Proceedings of the international symposium on Code generation and optimization (CGO '05)* (2005), pp. 137–137.

[Bat68] BATCHER K. E.: Sorting networks and their applications. In *Proceedings of the 1968 Spring Joint Computer Conference (SJCC)* (1968), vol. 32, pp. 307–314.

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 777–786.

[BN89] BILARDI G., NICOLAU A.: Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM J. Comput. 18*, 2 (Apr. 1989), 216–228.

[Col88] COLE R.: Parallel merge sort. *SIAM J. Comput. 17*, 4 (Aug. 1988), 770–785. see Correction in SIAM J. Comput. 22, 1349.

[GGKM05] GOVINDARAJU N. K., GRAY J., KUMAR R., MANOCHA D.: *GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management*. Tech. Rep. MSR-TR-2005-183, Microsoft Research (MSR), Dec. 2005. Proceedings of ACM SIGMOD Conference, Chicago, IL, June, 2006.

[GR88] GIBBONS A., RYTTER W.: *Efficient parallel algorithms*. Cambridge University Press, Cambridge, England, 1988.

[GRHM05] GOVINDARAJU N. K., RAGHUVANSHI N., HENSON M., MANOCHA D.: *A cache-efficient sorting algorithm for database and data mining computations using graphics processors*. Tech. rep., University of North Carolina, Chapel Hill, 2005.

[GRM05] GOVINDARAJU N. K., RAGHUVANSHI N., MANOCHA D.: Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05)* (2005), pp. 611–622.

[GZ06] GRESS A., ZACHMANN G.: GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)* (Apr. 2006), p. 45.

[HG97] HAKURA Z. S., GUPTA A.: The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)* (June 1997), vol. 25, 2 of *Computer Architecture News*, pp. 108–120.

[KDR*00] KAPASI U. J., DALLY W. J., RIXNER S., MATTSON P. R., OWENS J. D., KHAILANY B.: Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-33)* (2000), pp. 159–170.

[KRD*03] KAPASI U. J., RIXNER S., DALLY W. J., KHAILANY B., AHN J. H., MATTSON P. R., OWENS J. D.: Programmable stream processors. *IEEE Computer 36*, 8 (Aug. 2003), 54–62.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: Uberflow: a GPU-based particle engine. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics conference on Graphics hardware (EGGH '04)* (2004), pp. 115–122.

[KW05] KIPFER P., WESTERMANN R.: Improved GPU sorting. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (2005), Pharr M., (Ed.), Addison-Wesley, pp. 733–746.

[Mor66] MORTON G. M.: *A Computer-oriented Geodetic Data Base and a New Technique in File Sequencing*. Tech. rep., IBM Ltd. Ottawa, Canada, 1966.

[Nat90] NATVIG L.: Logarithmic time cost optimal parallel sorting is not yet fast in practice! In *Proceedings Supercomputing '90* (1990), pp. 486–494.

[Owe02] OWENS J. D.: *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University, Nov. 2002.

[Owe05] OWENS J.: Streaming architectures and technology trends. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (2005), Pharr M., (Ed.), Addison-Wesley, pp. 457–470.

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of the 2003 Annual ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (EGGH '03)* (2003), pp. 41–50.

## Impressum

**Publisher:** Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
**Editor of the series:** Jürgen Dix
**Technical editor:** Wojciech Jamroga
**Contact:** wjamroga@in.tu-clausthal.de
**URL:** http://www.in.tu-clausthal.de/~wjamroga/techreports/
**ISSN:** 1860-8477

## The IfI Review Board