Adaptive Bitonic Sorting*

Gabriel Zachmann
Department of Computer Science, University of Bremen, Germany,
Email: zach ät uni-bremen.de

April 6, 2013

Abstract

Adaptive bitonic sorting is a sorting algorithm suitable for implementation on EREW parallel architectures. Similar to bitonic sorting, it is based on merging, which is recursively applied to obtain a sorted sequence. In contrast to bitonic sorting, it is data dependent. Adaptive bitonic merging can be performed in $O\left(\frac{n}{p}\right)$ parallel time, p being the number of processors, and executes only $O(n)$ operations in total. Consequently, adaptive bitonic sorting can be performed in $O\left(\frac{n \log n}{p}\right)$ time, which is optimal. So, one of its advantages is that it executes a factor of $O(\log n)$ less operations than bitonic sorting. Another advantage is that it can be implemented efficiently on modern GPUs.

1 Introduction

This chapter describes a parallel sorting algorithm, *adaptive bitonic sorting* [2], that offers the following benefits:

- it needs only the optimal total number of comparison/exchange operations, $O(n \log n)$;
- the hidden constant in the asymptotic number of operations is less than in other optimal parallel sorting methods;
- it can be implemented in a highly parallel manner on modern architectures, such as a streaming architecture (GPUs), even without any scatter operations, i.e., without random access writes.

One of the main differences between "regular" bitonic sorting and adaptive bitonic sorting is that regular bitonic sorting is data-independent, while adaptive bitonic sorting is data-dependent (hence the name).

As a consequence, adaptive bitonic sorting cannot be implemented as a sorting network, but only on architectures that offer some kind of flow control. Nonetheless, it is convenient to derive the method of adaptive bitonic sorting from bitonic sorting.

Sorting networks have a long history in computer science research (see the comprehensive survey [3]). One reason is that sorting networks are a convenient way to describe parallel sorting algorithms on CREW-PRAMs or even EREW-PRAMs (which is also called PRAC for "parallel random access computer").

In the following, let n denote the number of keys to be sorted, and p the number of processors. For the sake of clarity, n will always be assumed to be a power of 2. (In their original paper [2], Bilardi and Nicolau have described how to modify the algorithms such that they can handle arbitrary numbers of keys, but these technical details will be omitted in this article.)

*This is a slightly updated version of [1].

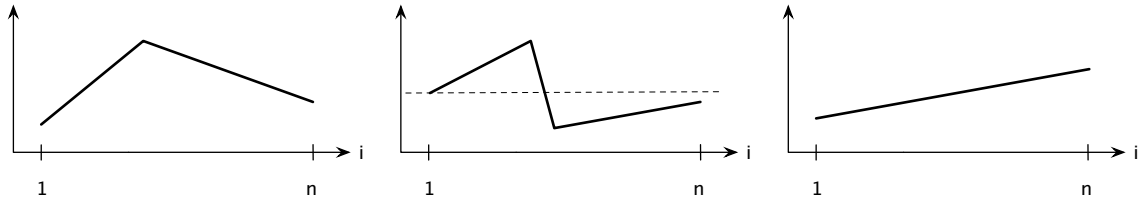

Figure 1: Three examples of sequences that are bitonic. Obviously, the mirrored sequences (either way) are bitonic, too.

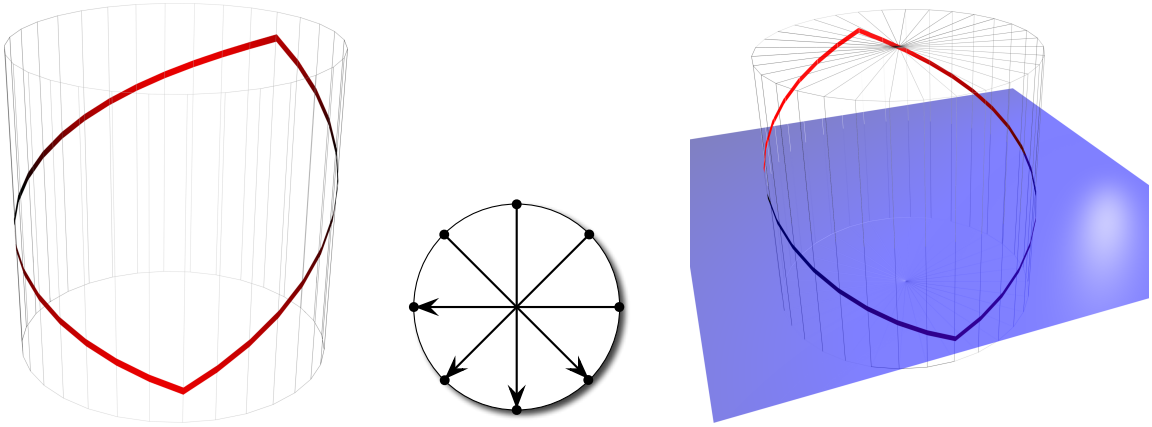

Figure 2: Left: according to their definition, bitonic sequences can be regarded as lying on a cylinder or as being arranged in a circle. As such, they consist of one monotonically increasing and one decreasing part. Middle: in this point of view, the network that performs the L and U operators (see Figure 5) can be visualized as a wheel of "spokes". Right: visualization of the effect of the L and U operators; the blue plane represents the median.

The first to present a sorting network with optimal asymptotic complexity were Ajtai, Komlós, and Szemerédi [4]. Also, Cole [5] presented an optimal parallel merge sort approach for the CREW-PRAM as well as for the EREW-PRAM. However, it has been shown that neither is fast in practice for reasonable numbers of keys [6, 7].

In contrast, adaptive bitonic sorting requires less than $2n \log n$ comparisons in total, independent of the number of processors. On p processors, it can be implemented in $O\left(\frac{n \log n}{p}\right)$ time, for $p \leq \frac{n}{\log n}$.

Even with a small number of processors it is efficient in practice: in its original implementation, the sequential version of the algorithm was at most by a factor 2.5 slower than quicksort (for sequence lengths up to 2^{19}) [2].

2 Fundamental Properties

One of the fundamental concepts in this context is the notion of a *bitonic sequence*.

Definition 1 (Bitonic sequence)
Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$ be a sequence of numbers. Then, $\mathbf{a}$ is *bitonic*, iff it monotonically increases and then monotonically decreases, *or* if it can be cyclically shifted (i.e., rotated) to become monotonically increasing and then monotonically decreasing.

Figure 1 shows some examples of bitonic sequences.

In the following, it will be easier to understand any reasoning about bitonic sequences, if one considers them as being arranged in a circle or on a cylinder: then, there are only two inflection

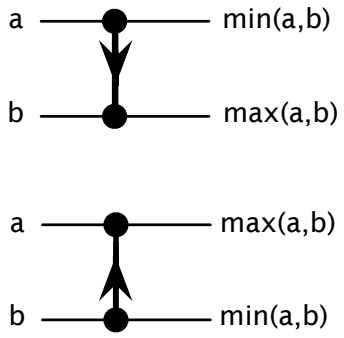

Figure 3: Comparator/exchange elements.

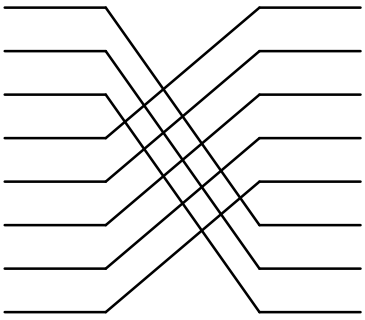

Figure 4: A network that performs the rotation operator.

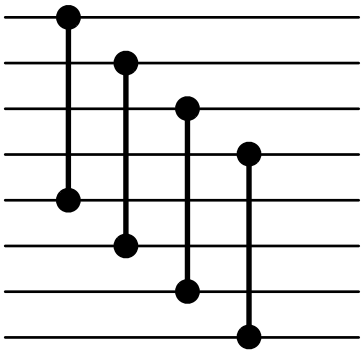

Figure 5: A half-cleaner that performs the L and U operators.

points around the circle. This is justified by Definition 1. Figure 2 depicts an example in this manner.

As a consequence, all index arithmetic is understood *modulo* n, i.e., index $i + k \equiv i + k \bmod n$, *unless* otherwise noted, so indices range from 0 through $n - 1$.

As mentioned above, adaptive bitonic sorting can be regarded as a variant of bitonic sorting, which is

In order to capture the notion of "rotational invariance" (in some sense) of bitonic sequences, it is convenient to define the following *rotation operator*.

Definition 2 (Rotation)
Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$ and $j \in \mathbb{N}$. We define a rotation as an operator R_j on the sequence $\mathbf{a}$:

$$R_j\mathbf{a} = (a_j, a_{j+1}, \ldots, a_{j+n-1})$$

This operation is performed by the network shown in Figure 4. Such networks are comprised of elementary *comparators* (see Figure 3).

Two other operators are convenient to describe sorting.

Definition 3 (Half-cleaner)
Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$.

$$L\mathbf{a} = (\min(a_0, a_{\frac{n}{2}}), \ldots, \min(a_{\frac{n}{2}-1}, a_{n-1})),$$
$$U\mathbf{a} = (\max(a_0, a_{\frac{n}{2}}), \ldots, \max(a_{\frac{n}{2}-1}, a_{n-1})).$$

In [8], a network that performs these operations together is called a *half-cleaner* (see Figure 5).

It is easy to see that, for any j and $\mathbf{a}$,

$$L\mathbf{a} = R_{-j \bmod \frac{n}{2}} L R_j \mathbf{a}, \tag{1}$$

and

$$U\mathbf{a} = R_{-j \bmod \frac{n}{2}} U R_j \mathbf{a}. \tag{2}$$

This is the reason why the cylinder metaphor is valid.

The proof needs to consider only two cases: $j = \frac{n}{2}$ and $1 \leq j < \frac{n}{2}$. In the former case, equation 1 becomes $L\mathbf{a} = LR_{\frac{n}{2}}\mathbf{a}$, which can be verified trivially. In the latter case, equation 1 becomes

$$\begin{aligned} LR_j\mathbf{a} &= \big(\min(a_j, a_{j+\frac{n}{2}}), \ldots, \min(a_{\frac{n}{2}-1}, a_{n-1}), \ldots, \min(a_{j-1}, a_{j-1+\frac{n}{2}})\big) \\ &= R_j L\mathbf{a}. \end{aligned}$$

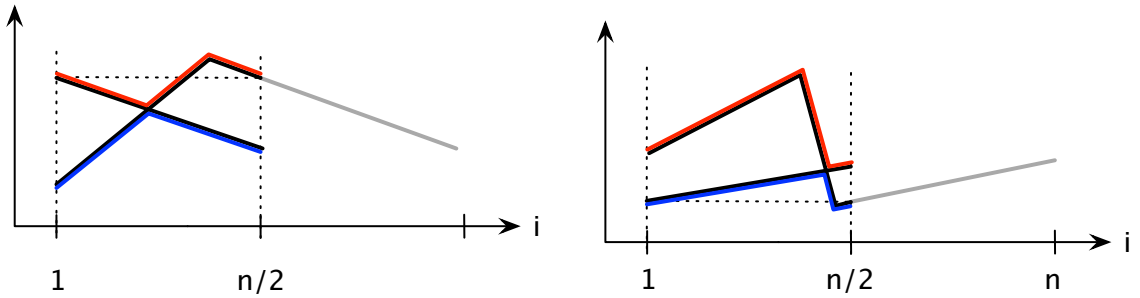

Figure 6: Examples of the result of the L and U operators. Conceptually, these operators fold the bitonic sequence (black), such that the range $\frac{n}{2}$ through $n-1$ (light gray) is shifted into the range 0 through $\frac{n}{2}-1$ (black); then, operators L and U yield the upper (red) and lower (blue) hull, resp.

Thus, with the cylinder metaphor, the L and U operators basically do the following: cut the cylinder with circumference n at any point, roll it around a cylinder with circumference $\frac{n}{2}$, and perform position-wise the max and min operator, resp. Some examples are shown in Figure 6.

The following theorem states some important properties of the L and U operators.

Theorem 1
Given a bitonic sequence $\mathbf{a}$,

$$\max\{L\mathbf{a}\} \leq \min\{U\mathbf{a}\}\,.$$

Moreover, $L\mathbf{a}$ and $U\mathbf{a}$ are bitonic, too.

In other words, each element of $L\mathbf{a}$ is less than or equal to each element of $U\mathbf{a}$.

This theorem is the basis for the construction of the bitonic sorter [9]. The first step is to devise a *bitonic merger* (BM). We denote a BM that takes as input bitonic sequences of length n with BM_n. A BM is recursively defined as follows

$$\mathrm{BM}_n(\mathbf{a}) = \left(\mathrm{BM}_{\frac{n}{2}}(L\mathbf{a}),\, \mathrm{BM}_{\frac{n}{2}}(U\mathbf{a})\right).$$

The base case is, of course, a two-key sequence, which is handled by a single comparator. A BM can be easily represented in a network as shown in Figure 7.

Given a bitonic sequence $\mathbf{a}$ of length n, one can show that

$$\mathrm{BM}_n(\mathbf{a}) = \mathrm{Sorted}(\mathbf{a}). \tag{3}$$

It should be obvious that the sorting direction can be changed simply by swapping the direction of the elementary comparators.

Coming back to the metaphor of the cylinder, the first stage of the bitonic merger in Figure 7 can be visualized as $\frac{n}{2}$ comparators, each one connecting an element of the cylinder with the opposite one, somewhat like spokes in a wheel. Note that here, while the cylinder can rotate freely, the "spokes" must remain fixed.

From a bitonic merger, it is straight-forward to derive a bitonic sorter, BS_n, that takes an unsorted sequence, and produces a sorted sequence, either up or down. Like the BM, it is defined recursively, consisting of two smaller bitonic sorters and a bitonic merger (see Figure 8). Again, the base case is the two-key sequence.

3 Analysis of the Number of Operations of Bitonic Sorting

Since a bitonic sorter basically consists of a number of bitonic mergers, it suffices to look at the total number of comparisons of the latter.

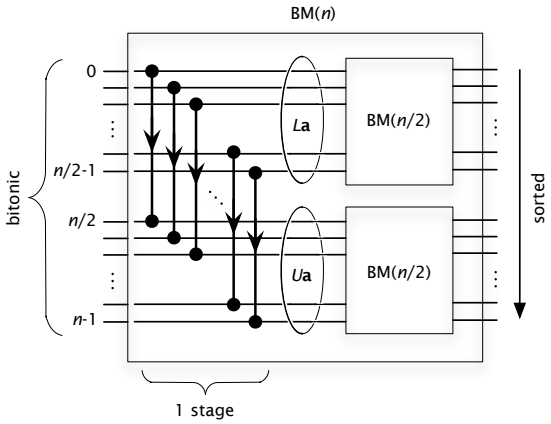

Figure 7: Schematic, recursive diagram of a network that performs bitonic merging.

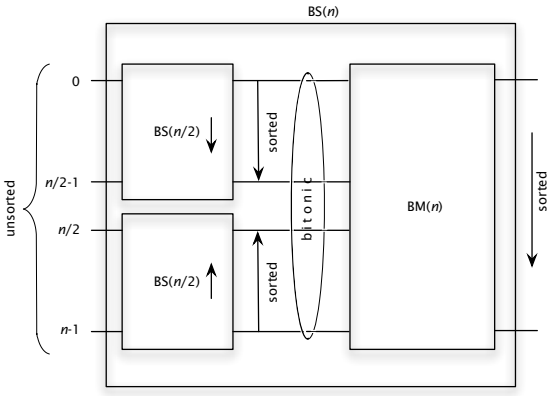

Figure 8: Schematic, recursive diagram of a bitonic sorting network.

The total number of comparators, $C(n)$, in the bitonic merger BM_n is given by

$$C(n) = 2C(\frac{n}{2}) + \frac{n}{2}\,, \quad \text{with } C(2) = 1$$

which amounts to

$$C(n) = \frac{1}{2} n \log n.$$

As a consequence, the bitonic sorter consists of $O\big(n \log^2 n\big)$ comparators.

Clearly, there is some redundancy in such a network, since n comparisons are sufficient to merge two sorted sequences. The reason is that the comparisons performed by the bitonic merger are *data-independent*.

4 Derivation of Adaptive Bitonic Merging

The algorithm for adaptive bitonic sorting is based on the following theorem.

Theorem 2
Let $\mathbf{a}$ be a bitonic sequence. Then, there is an index q such that

$$\max\big(a_q, \ldots, a_{q+\frac{n}{2}-1}\big) \leq \min\big(a_{q+\frac{n}{2}}, \ldots, a_{q-1}\big) \tag{4}$$

The following outline of the proof assumes, for the sake of simplicity, that all elements in $\mathbf{a}$ are distinct. Employing the cylinder metaphor again, we can clearly see that any horizontal plane cuts the sequence at exactly two places (i.e., indices) (see Figure 2, right hand side). Therefore, any such "cut plane" partitions a bitonic sequence into two contiguous sub-sequences, and all elements of the "lower" sequence are smaller then any element of the "upper" sequence. (Remember that index arithmetic is always meant modulo n, therefore, "contiguous" can involve a wrap-around, too.) Now, use the median as the cut plane, so each sub-sequence has exactly length $\frac{n}{2}$. The indices where this cut happens are just q and $q + \frac{n}{2}$. Figure 9 shows an example (in one dimension).

The following theorem is the final key stone for the adaptive bitonic sorting algorithm.

Theorem 3
Any bitonic sequence $\mathbf{a}$ can be partitioned into four sub-sequences $(\mathbf{a}^1, \mathbf{a}^2, \mathbf{a}^3, \mathbf{a}^4)$ such that either

$$(L\mathbf{a}, U\mathbf{a}) = (\mathbf{a}^1, \mathbf{a}^4, \mathbf{a}^3, \mathbf{a}^2) \tag{5}$$

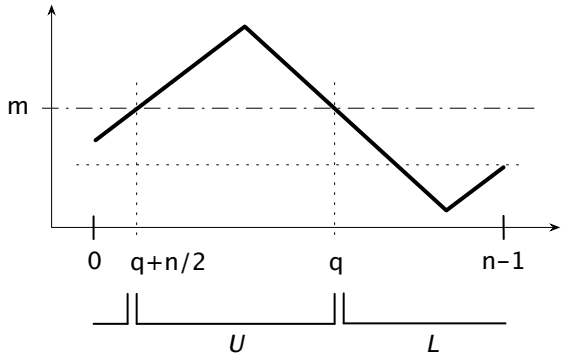

Figure 9: Visualization for the proof of Theorem 2.

or

$$(L\mathbf{a}, U\mathbf{a}) = (\mathbf{a}^3, \mathbf{a}^2, \mathbf{a}^1, \mathbf{a}^4). \tag{6}$$

Furthermore,

$$|\mathbf{a}^1| + |\mathbf{a}^2| = |\mathbf{a}^3| + |\mathbf{a}^4| = \frac{n}{2}\,, \tag{7}$$

$$|\mathbf{a}^1| = |\mathbf{a}^3|\,, \quad \text{and} \tag{8}$$

$$|\mathbf{a}^2| = |\mathbf{a}^4|\,, \tag{9}$$

where $|\mathbf{a}|$ denotes the length of sequence $\mathbf{a}$.

Figure 10 illustrates this theorem by an example.

In other words, in order to establish $(L\mathbf{a}, U\mathbf{a})$, we either need to swap (just) $\mathbf{a}^2$ and $\mathbf{a}^4$, or we need to swap $\mathbf{a}^1$ and $\mathbf{a}^3$.

This theorem can be proven fairly easily, too: the length of the subsequences is just q and $\frac{n}{2} - q$, where q is the same as in Theorem 2. Assuming that $\max\{\mathbf{a}^1\} < m < \min\{\mathbf{a}^3\}$, nothing will change between those two sub-sequences (see Figure 10). However, in that case $\min\{\mathbf{a}^2\} > m > \max\{\mathbf{a}^4\}$; therefore, by swapping $\mathbf{a}^2$ and $\mathbf{a}^4$ (which have equal length), the bounds $\max\{(\mathbf{a}^1, \mathbf{a}^4)\} < m < \min\{\mathbf{a}^2, \mathbf{a}^3)\}$ are obtained. The other case can be handled analogously.

Remember that, in a half-cleaner, there are $\frac{n}{2}$ comparator-and-exchange elements, each of which compares a_i and $a_{i+\frac{n}{2}}$. They will perform exactly this exchange of sub-sequences, without ever looking at the data.

Now, the idea of adaptive bitonic sorting is to find the sub-sequences, i.e., to find the index q that marks the border between the sub-sequences. Once q is found, one can (conceptually) swap the sub-sequences, instead of performing $\frac{n}{2}$ comparisons unconditionally.

Finding q can be done simply by binary search driven by comparisons of the form $(a_i, a_{i+\frac{n}{2}})$.

Overall, instead of performing $\frac{n}{2}$ comparisons in the first stage of the bitonic merger (see Figure 7), the adaptive bitonic merger performs $\log(\frac{n}{2})$ comparisons in its first stage (although this stage is no longer representable by a network).

Let $C(n)$ be the total number of comparisons performed by adaptive bitonic merging, in the worst case. Then

$$C(n) = 2C(\frac{n}{2}) + \log(n) = \sum_{i=0}^{k-1} 2^i \log(\frac{n}{2^i})$$

with $C(2) = 1, C(1) = 0$ and $n = 2^k$. This amounts to

$$C(n) = 2n - \log n - 2$$

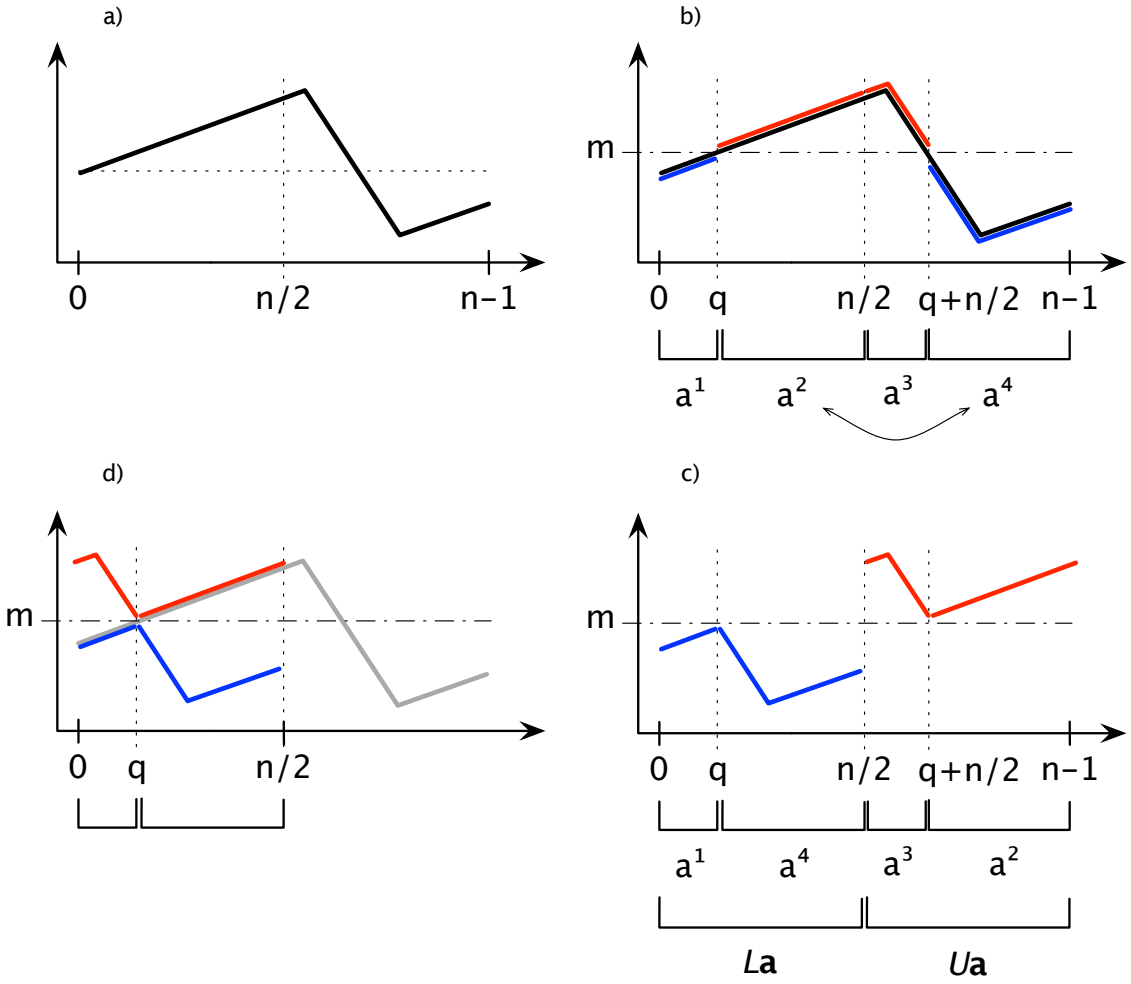

Figure 10: Example illustrating Theorem 3.

The only question that remains is how to achieve the data rearrangement, i.e., the swapping of the subsequences $\mathbf{a}^1$ and $\mathbf{a}^3$ or $\mathbf{a}^2$ and $\mathbf{a}^4$, resp., without sacrificing the worst-case performance of $O(n)$. This can be done by storing the keys in a perfectly balanced tree (assuming $n = 2^k$), the so-called *bitonic tree*. (The tree can, of course, store only $2^k - 1$ keys, so the n-th key is simply stored separately.) This tree is very similar to a search tree, which stores a monotonically increasing sequence: when traversed in-order, the bitonic tree produces a sequence that lists the keys such that there are exactly two inflection points (when regarded as a circular list).

Instead of actually copying elements of the sequence in order to achieve the exchange of subsequences, the adaptive bitonic merging algorithm swaps $O(\log n)$ pointers in the bitonic tree. The recursion then works on the two sub-trees. With this technique, the overall number of operations of adaptive bitonic merging is $O(n)$. Details can be found in [2].

Clearly, the adaptive bitonic sorting algorithm needs $O(n \log n)$ operations in total, because it consists of $\log(n)$ many complete merge stages (see Figure 8).

It should also be fairly obvious that the adaptive bitonic sorter performs an (adaptive) subset of the comparisons that are executed by the (non-adaptive) bitonic sorter.

5 The Parallel Algorithm

So far, the discussion assumed a sequential implementation. Obviously, the algorithm for adaptive bitonic merging can be implemented on a parallel architecture, just like the bitonic merger, by executing recursive calls on the same level in parallel.

Unfortunately, a naïve implementation would require $O(\log^2 n)$ steps in the worst-case, since there are $\log(n)$ levels. The bitonic merger achieves $O(\log n)$ parallel time, because all pair-wise comparisons within one stage can be performed in parallel. But this is not straight-forward to achieve for the $\log(n)$ comparisons of the binary-search method in adaptive bitonic merging, which are inherently sequential.

However, a careful analysis of the data dependencies between comparisons of successive stages reveals that the execution of different stages can be partially overlapped [2]. As $L\mathbf{a}, U\mathbf{a}$ are being constructed in one stange by moving down the tree in parallel layer by layer (occasionally swapping pointers), this process can be started for the next stage, which begins one layer beneath the one where the previous stage began, before the first stage has finished, provided the first stage has progressed "far enough" in the tree. Here, "far enough" means exactly two layers ahead.

This leads to a parallel version of the adaptive bitonic merge algorithm that executes in time $O(\frac{n}{p})$ for $p \in O(\frac{n}{\log n})$, i.e., it can be executed in $O(\log n)$ parallel time.

Furthermore, the data that needs to be communicated between processors (either via memory, or via communication channels) is in $O(p)$.

It is straight-forward to apply the classical sorting-by-merging approach here (see Figure 8), which yields the *adaptive bitonic sorting* algorithm. This can be implemented on an EREW machine with p processors in $O(\frac{n \log n}{p})$ time, for $p \in O(\frac{n}{\log n})$.

6 A GPU Implementation

Because adaptive bitonic sorting has excellent scalability (the number of processors, p, can go up to $n/\log(n)$) and the amount of inter-process communication is fairly low (only $O(p)$), it is perfectly suitable for implementation on stream processing architectures. In addition, although it was designed for a random access architecture, adaptive bitonic sorting can be adapted to a stream processor, which (in general) does not have the ability of random-access writes. Finally, it can be implemented on a GPU such that there are only $O(\log^2(n))$ passes (by utilizing $O(n/\log(n))$ (conceptual) processors), which is very important, since the number of passes is one of the main limiting factors on GPUs.

This section provides more details on the implementation on a GPU, called "GPU-ABiSort" [10, 11]. For the sake of simplicity, the following always assumes increasing sorting direction, and it is thus not explicitely specified. As noted above, the sorting direction must be reversed in the right branch of the recursion in the bitonic sorter, which basically amounts to reversing the comparison direction of the values of the keys, i.e., compare for $<$ instead of $>$ in Algorithm 3.

As noted above, the bitonic tree stores the sequence $(a_0, \ldots, a_{n-2})$ in in-order, and the key a_{n-1} is stored in the *extra node*. As mentioned above, an algorithm that constructs $(L\mathbf{a}, U\mathbf{a})$ from $\mathbf{a}$ can traverse this bitonic tree and swap pointers as necessary. The index q, which is mentioned in the proof for Theorem 3, is only determined implicitly. The two different cases that are mentioned in Theorem 3, equations (5) and (6), can be distinguished simply by comparing elements $a_{\frac{n}{2}-1}$ and a_{n-1}.

This leads to Algorithm 1. Note that the root of the bitonic tree stores element $a_{\frac{n}{2}-1}$ and the extra node stores a_{n-1}. Applying this recursively yields Algorithm 2. Note that the bitonic tree needs to be constructed only once at the beginning during setup time.

Because branches are very costly on GPUs, one should avoid as many conditionals in the inner loops as possible. Here, one can exploit the fact that $R_{n/2}\mathbf{a} = (a_{\frac{n}{2}}, \ldots, a_{n-1}, a_0, \ldots, a_{\frac{n}{2}-1})$ is

Algorithm 1: Adaptive construction of $L\mathbf{a}$ and $U\mathbf{a}$ (one stage of adaptive bitonic merging)

input : Bitonic tree, with root node `r` and extra node `e`, representing bitonic sequence $\mathbf{a}$

output: $L\mathbf{a}$ in the left subtree of `r` plus root `r`, and $U\mathbf{a}$ in the right subtree of `r` plus extra node `e`

// phase 0: determine case

if `value(r)` $<$ `value(e)` **then**
    case $= 1$

else
    case $= 2$
    swap `value(r)` and `value(e)`

(p, q) = (`left(r)` , `right(r)`)

for $i = 1, \ldots, \log n - 1$ **do**
    // phase i
    test = (`value(p)` $>$ `value(q)`)
    if test $==$ *true* **then**
        swap values of p and q
        if case $==$ *1* **then**
            swap the pointers `left(p)` and `left(q)`
        else
            swap the pointers `right(p)` and `right(q)`
    if (case $==$ *1 and* test $==$ *false*) *or* (case $==$ *2 and* test $==$ *true*) **then**
        (p, q) = (`left(p)` , `left(q)`)
    else
        (p, q) = (`right(p)` , `right(q)`)

Algorithm 2: Merging a bitonic sequence to obtain a sorted sequence

input : Bitonic tree, with root node `r` and extra node `e`, representing bitonic sequence $\mathbf{a}$

output: Sorted tree (produces `sort(a)` when traversed in-order)

construct $L\mathbf{a}$ and $U\mathbf{a}$ in the bitonic tree by Algorithm 1

call merging recursively with `left(r)` as root and `r` as extra node

call merging recursively with `right(r)` as root and `e` as extra node

Algorithm 3: Simplified adaptive construction of $L\mathbf{a}$ and $U\mathbf{a}$

input : Bitonic tree, with root node `r` and extra node `e`, representing bitonic sequence $\mathbf{a}$

output: $L\mathbf{a}$ in the left subtree of `r` plus root `r`, and $U\mathbf{a}$ in the right subtree of `r` plus extra node `e`

```
// phase 0
if value(r) > value(e) then
    swap value(r) and value(e)
    swap pointers left(r) and right(r)
( p, q ) = ( left(r) , right(r) )
for i = 1, ..., log n − 1 do
    // phase i
    if value(p) > value(q) then
        swap value(p) and value(q)
        swap pointers left(p) and left(q)
        ( p, q ) = ( right(p) , right(q) )
    else
        ( p, q ) = ( left(p) , left(q) )
```

bitonic, provided $\mathbf{a}$ is bitonic, too. This operation basically amounts to swapping the two pointers left(root) and right(root). The simplified construction of $L\mathbf{a}$ and $U\mathbf{a}$ is presented in Algorithm 3. (Obviously, the simplified algorithm now really needs trees with pointers, whereas Bilardi's original bitonic tree could be implemented pointer-less (since it is a complete tree). However, in a real-world implementation, the keys to be sorted must carry pointers to some "payload" data anyway, so the additional memory overhead incurred by the child pointers is at most a factor 1.5 .)

6.1 Outline of the implementation

As explained above, on each recursion level $j = 1, \ldots, \log(n)$ of the adaptive bitonic sorting algorithm, $2^{\log n - j + 1}$ bitonic trees, each consisting of 2^{j-1} nodes, have to be merged into $2^{\log n - j}$ bitonic trees of 2^j nodes. The merge is performed in j stages. In each stage $k = 0, \ldots, j-1$, the construction of $L\mathbf{a}$ and $U\mathbf{a}$ is executed on 2^k subtrees. Therefore, $2^{\log n - j} \cdot 2^k$ instances of the $L\mathbf{a}$ / $U\mathbf{a}$ construction algorithm can be executed in parallel during 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 so-called kernel on each element.

The $L\mathbf{a}$ / $U\mathbf{a}$ construction algorithm consists of $j - k$ phases, where each phase reads and modifies a pair of nodes, (p, q), of a bitonic tree. Assume that a kernel implementation performs the operation of a single phase of this algorithm. (How such a kernel implementation is realized without random-access writes will be described below.) The temporary data that have to be preserved from one phase of the algorithm to the next one are just two node pointers (p and q) per kernel instance. Thus, each of the $2^{\log n - j + k}$ elements of the allocated stream consist of exactly these 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 one phase of the $L\mathbf{a}/U\mathbf{a}$ construction algorithm, and finally writes the updated pair of node pointers (p, q) back to the stream.

6.2 Eliminating random-access writes

Since GPUs do not support random-access writes (at least, for almost all practical purposes, random-access writes would kill any performance gained by the parallelism) the kernel has to be

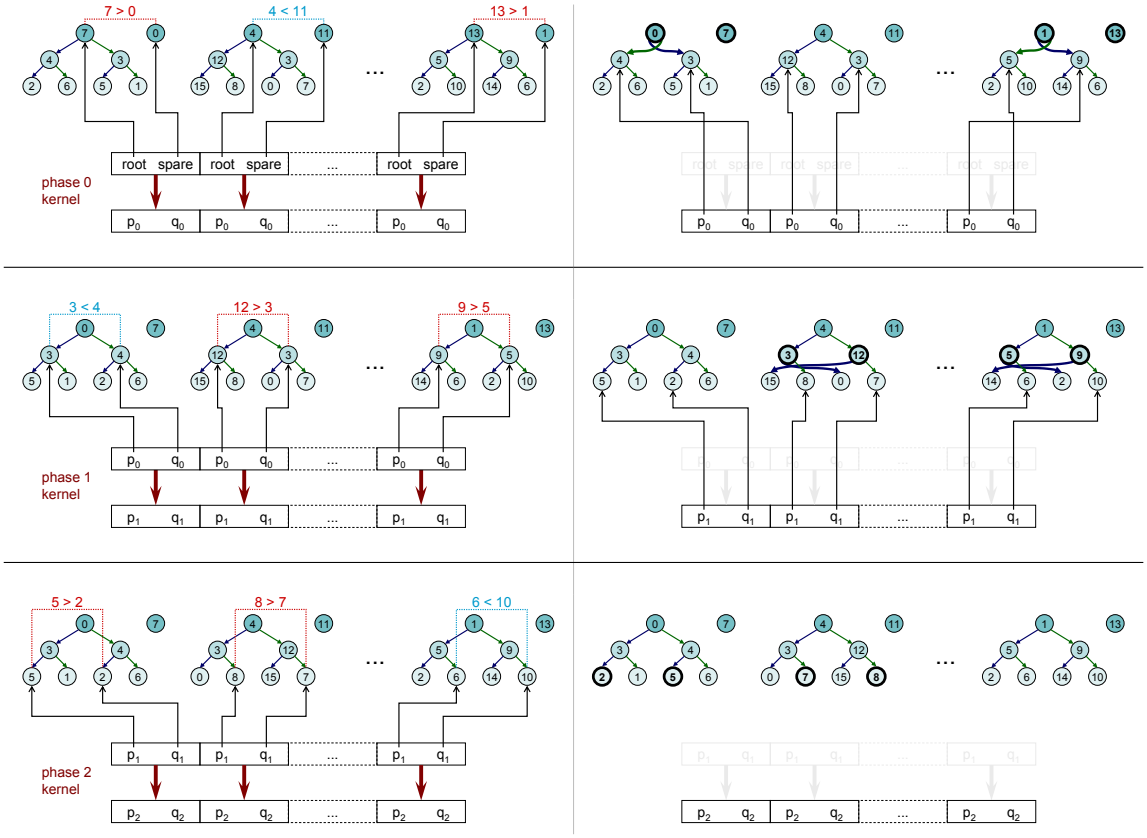

Figure 11: To execute several instances of the adaptive La/Ua construction algorithm in parallel, where each instance operates on a bitonic tree of 2^3 nodes, 3 phases are required. This figure illustrates the operation of these 3 phases. On the left, the node pointers contained in the input stream are shown as well as the comparisons performed by the kernel program. On the right, the node pointers written to the output stream are shown as well as the modifications of the child pointers and node values performed by the kernel program according to Algorithm 3.

implement so that it modifies node pairs (p, q) of the bitonic tree without random-access writes. This means that it can output node pairs from the kernel only via linear stream write. But this way it cannot write a modified node pair to its original location from where it was read. In addition, it can not simply take an input stream (containing a bitonic tree) and produce another output stream (containing the modified bitonic tree), because then it 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.

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

Notice that Algorithm 3 basically traverses the bitonic tree down along a path, changing some of the nodes as necessary. The strategy is simple: simply output every node visited along this path to a stream. Since the data layout is fixed and pre-determined, the kernel can store the index of the children with the node as it is being written to the output stream. One child address remains the

same anyway, while the other is determined while the kernel is still executing for the current node. Figure 11 demonstrates the operation of the stream program using the described stream output technique.

6.3 Complexity

A simple implementation on the GPU would need $O(\log^2 n)$ phases (or "passes" in GPU parlance) in total for adaptive bitonic sorting, which amounts to $O(\log^3 n)$ operations in total.

This is already very fast in practice. However, the optimal complexity of $O(\log n)$ passes can be achieved exactly as described in the original work [2], i.e., phase i of a stage k can be executed immediately after phase $i+1$ of stage $k-1$ has finished. Therefore, the execution of a new stage can start every other step of the algorithm.

The only difference from the simple implementation is that kernels now must write to parts of the output stream, because other parts are still in use.

7 GPU-specific Details

For the input and output streams, it is best to apply the *ping-pong* technique commonly used in GPU programming: allocate two such streams and alternatingly use one of them as input and the other one as output stream.

7.1 Preconditioning the input

For merge-based sorting on a PRAM architecture (and assuming $p < n$), it is a common technique to sort *locally*, in a first step, p blocks of n/p values, i.e. each processor sorts n/p values using a standard sequential algorithm.

The same technique can be applied here 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.

On recent GPUs, the maximum output data size of a kernel is 16×4 bytes. Since usually the input consists of key/pointer pairs, the method starts with a local sort of 8 key/pointer pairs per kernel. For such small numbers of keys, an algorithm with asymptotic complexity of $O(n)$ performs faster than asymptotically optimal algorithms.

After the local sort, a further stream operation converts the resulting sorted subsequences of length 8 pairwise to bitonic trees, each containing 16 nodes. Thereafter, the GPU-ABiSort approach can be applied as described above, starting with $j = 4$.

7.2 The last stage of each merge

Adaptive bitonic merging, being a recursive procedure, eventually merges small subsequences, for instance of length 16. For such small subsequences it is better to use a (non-adaptive) bitonic merge implementation that can be executed in a single pass of the whole stream.

8 Timings

The following experiments were done on arrays consisting of key/pointer pairs, where the key is a uniformly distributed random 32-bit floating point value and the pointer a 4-byte address. Since one can assume (without loss of generality) that all pointers in the given array are unique, these can be used as secondary sort keys for the adaptive bitonic merge.

n	CPU sort	GPUSort	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

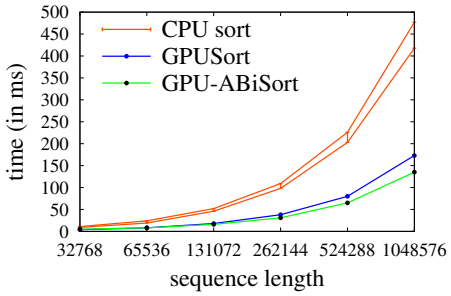

Figure 12: Timings on a GeForce 7800 system. (There are two curves for the CPU sort, so as to visualize that its running time is somewhat data-dependent.)

The experiments described in the following compare the implementation of GPU-ABiSort of [10, 11] 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 [12].

Contrary to the CPU STL sort, the timings of GPU-ABiSort do not depend very much on the data to be sorted, because the total number of comparisons performed by the adaptive bitonic sorting is not data-dependent.

Table 12 shows the results of timings performed 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. Obviously, the speed-up of GPU-ABiSort compared to CPU sorting is 3.1–3.5 for $n \geq 2^{17}$. Furthermore, up to the maximum tested sequence length $n = 2^{20}$ ($= 1\,048\,576$), GPU-ABiSort is up to 1.3 times faster than GPUSort, and this speed-up is increasing with the sequence length n, as expected.

The timings of the GPU approaches assume that the input data is already stored in GPU memory. When embedding the GPU-based sorting into an otherwise purely CPU-based application, the input data has to be transferred from CPU to GPU memory, and afterwards the output data has to be transferred back to CPU memory. However, the overhead of this transfer is usually negligible compared to the achieved sorting speed-up: according to measurements by [10], the transfer of 1 million key/pointer pairs from CPU to GPU and back takes in total roughly 20 ms on a PCI Express bus PC.

9 Conclusion

Adaptive bitonic sorting is not only appealing from a theoretical point of view, but also from a practical one. Unlike other parallel sorting algorithms that exhibit optimal asymptotic complexity, too, adaptive bitonic sorting offers low hidden constants in its asymptotic complexity and can be implemented on parallel architectures by a reasonably experienced programmer. The practical implementation of it on a GPU outperforms the implementation of simple bitonic sorting on the same GPU by a factor 1.3, and it is a factor 3 faster than a standard CPU sorting implementation (STL).

10 Bibliographic Notes and Further Reading

As mentioned in the introduction, this line of research began with the seminal work of Batcher [9] in the late 1960s, who described parallel sorting as a network. Research of parallel sorting algorithms was re-invigorated in the 1980s, where a number of theoretical questions have been settled [4, 5, 13, 14, 15, 2].

Another wave of research on parallel sorting ensued from the advent of affordable, massively parallel architectures, namely GPUs, which are, more precisely, streaming architectures. This spurred the development of a number of practical implementations [10, 11, 16, 17, 18, 19, 20].

References

[1] Gabriel Zachmann. Adaptive Bitonic Sorting. In *Encyclopedia of Parallel Computing*, David Padua, Ed., pages 146–157. Springer, 2011. ISBN 978-0-387-09765-7. 1

[2] Gianfranco Bilardi, and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.*, 18(2):216–228, April 1989. 1, 2, 7, 8, 12, 13

[3] Selim G. Akl. *Parallel Sorting Algorithms*. Academic Press, Inc., Orlando, FL, USA, 1990. ISBN 0120476800. 1

[4] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ Sorting Network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC '83)*, pages 1–9, 1983. 2, 13

[5] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, August 1988. see Correction in SIAM J. Comput. 22, 1349. 2, 13

[6] Alan Gibbons, and Wojciech Rytter. *Efficient parallel algorithms*. Cambridge University Press, Cambridge, England, 1988. 259 pp. 2

[7] Lasse Natvig. Logarithmic Time Cost Optimal Parallel Sorting is Not Yet Fast in Practice! In *Proceedings Supercomputing '90*, pages 486–494, 1990. 2

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3 ed., 2009. 3

[9] Kenneth E. Batcher. Sorting Networks and Their Applications. In *Proceedings of the 1968 Spring Joint Computer Conference (SJCC)*, vol. 32, pages 307–314, 1968. 4, 13

[10] A. Greß, and G. Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 45, April 2006. 8, 13, 14

[11] Alexander Greß, and Gabriel Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. Tech. Rep. IfI-06-11, TU Clausthal, Computer Science Department, Clausthal-Zellerfeld, Germany, October 2006. URL http://cg.in.tu-clausthal.de/publications.shtml. 8, 13, 14

[12] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Tech. rep., University of North Carolina, Chapel Hill, 2005. 13

[13] Tom Leighton. Tight bounds on the complexity of parallel sorting. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 71–80. ACM, New York, NY, USA, 1984. ISBN 0-89791-133-4. 13

[14] Yossi Azar, and Uzi Vishkin. Tight comparison bounds on the complexity of parallel sorting. *SIAM J. Comput.*, 16(3):458–464, 1987. ISSN 0097-5397. 13

[15] C P Schnorr, and A Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the eighteenth annual ACM Symposium on Theory of Computing (STOC)*, pages 255–263. ACM, New York, NY, USA, 1986. ISBN 0-89791-193-8. 13

[16] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. Tech. Rep. MSR-TR-2005-183, Microsoft Research (MSR), December 2005. URL ftp://ftp.research.microsoft.com/pub/tr/TR-2005-183.pdf. Proceedings of ACM SIGMOD Conference, Chicago, IL, June, 2006. 14

[17] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the 2003 Annual ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (EGGH '03)*, pages 41–50, 2003. 14

[18] Peter Kipfer, and Rüdiger Westermann. Improved GPU Sorting. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Matt Pharr, Ed., pages 733–746. Addison-Wesley, 2005. 14

[19] Erik Sintorn, and Ulf Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008. ISSN 0743-7315. 14

[20] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS)*, pages 1–10. IEEE Computer Society, Washington, DC, USA, 2009. ISBN 978-1-4244-3751-1. 14