A

Adaptive Bitonic Sorting

Gabriel Zachmann
Clausthal University, Clausthal-Zellerfeld, Germany

Definition

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.

Discussion

Introduction

This chapter describes a parallel sorting algorithm, *adaptive bitonic sorting* [5], 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, that is, 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 [2]). 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 [5], 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.)

The first to present a sorting network with optimal asymptotic complexity were Ajtai, Komlós, and Szemerédi [1]. Also, Cole [6] 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 [8, 15].

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 $p \le \frac{n}{\log n}$.

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}) [5].

David Padua (ed.), *Encyclopedia of Parallel Computing*, DOI 10.1007/978-0-387-09766-4,

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 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*, that is, index $i + k \equiv i + k \bmod n$, *unless* otherwise noted, so indices range from 0 through $n - 1$.

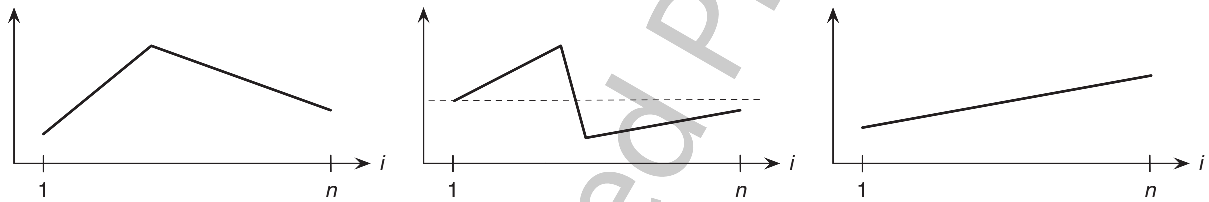
As mentioned above, adaptive bitonic sorting can be regarded as a variant of bitonic sorting, which is in order to capture the notion of “rotational invariance” (in some sense) of bitonic sequences; it is convenient to define the following *rotation operator*.

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 $\mathbf{a}$:

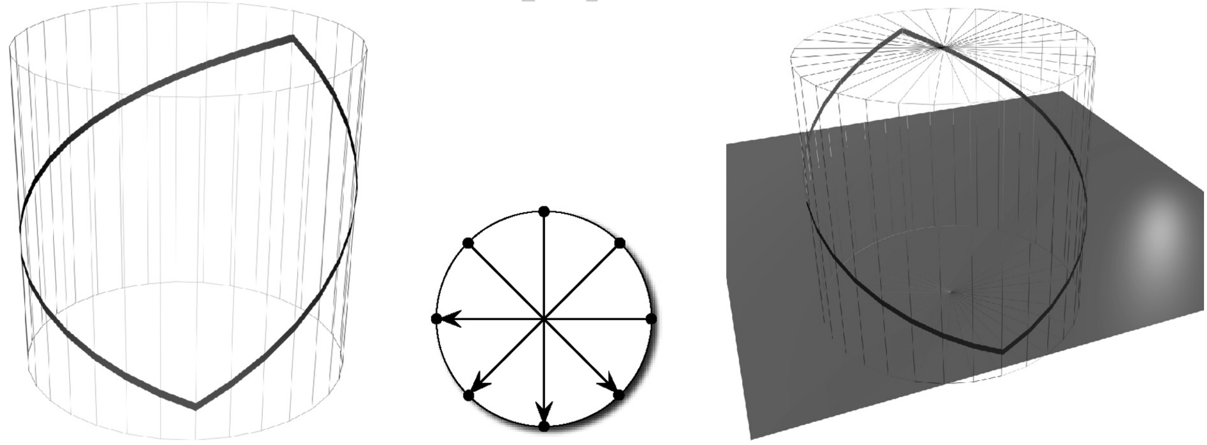
$$R_j\mathbf{a} = (a_j, a_{j+1}, \ldots, a_{j+n-1})$$

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

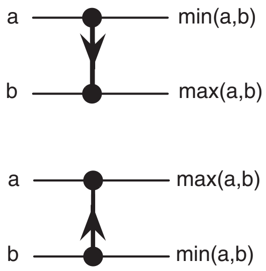
Two other operators are convenient to describe sorting.



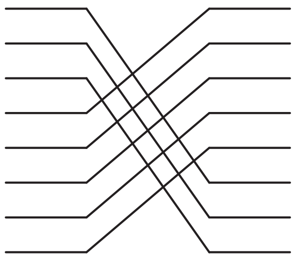
Adaptive Bitonic Sorting. Fig. 1 Three examples of sequences that are bitonic. Obviously, the mirrored sequences (either way) are bitonic, too



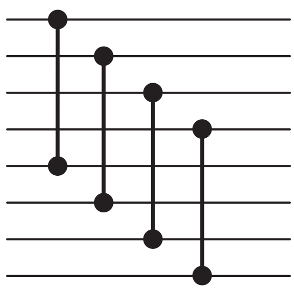
Adaptive Bitonic Sorting. Fig. 2 *Left*: according to their definition, bitonic sequences can be regarded as lying on a cylinder or as being arranged in a circle. As such, they consist of one monotonically increasing and one decreasing part. *Middle*: in this point of view, the network that performs the *L* and *U* operators (see Fig. 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



Adaptive Bitonic Sorting. Fig. 3 Comparator/exchange elements



Adaptive Bitonic Sorting. Fig. 4 A network that performs the rotation operator



Adaptive Bitonic Sorting. Fig. 5 A network that performs the *L* and *U* operators

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

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

In [7], a network that performs these operations together is called a *half-cleaner* (see Fig. 5).

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

$$L\mathbf{a} = R_{-j \bmod \frac{n}{2}} LR_j\mathbf{a}, \qquad (1)$$

and

$$U\mathbf{a} = R_{-j \bmod \frac{n}{2}} UR_j\mathbf{a}. \qquad (2)$$

This is the reason why the cylinder metaphor is valid.

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

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

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, respectively. Some examples are shown in Fig. 6.

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

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

$$\max\{L\mathbf{a}\} \le \min\{U\mathbf{a}\}.$$

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 [4]. 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:

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

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 Fig. 7.

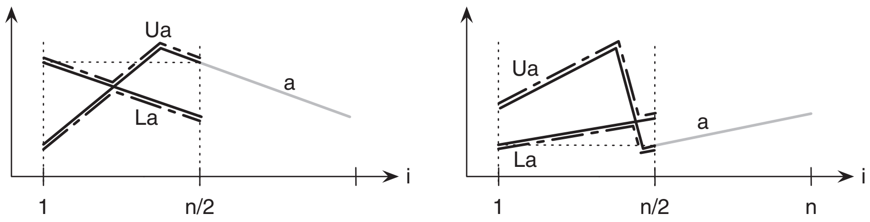
Given a bitonic sequence $\mathbf{a}$ of length n, one can show that

$$\text{BM}_n(\mathbf{a}) = \text{Sorted}(\mathbf{a}). \qquad (3)$$

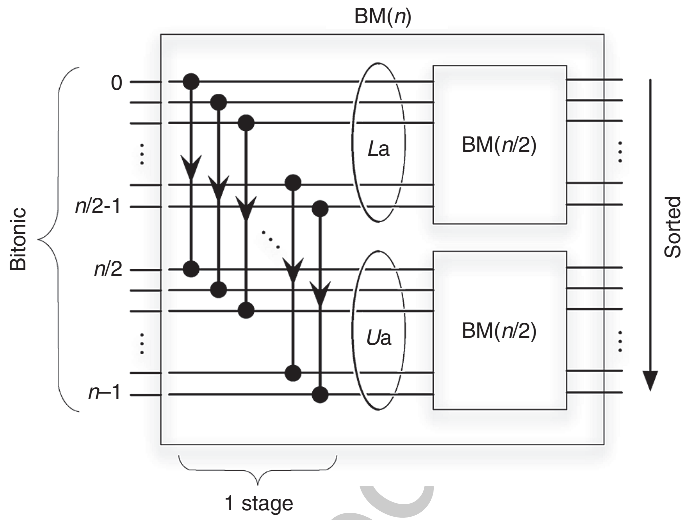
It should be obvious that the sorting direction can be changed simply by swapping the direction of the elementary comparators.

Coming back to the metaphor of the cylinder, the first stage of the bitonic merger in Fig. 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 straightforward to derive a bitonic sorter, BS_n, that takes an unsorted sequence,



Adaptive Bitonic Sorting. Fig. 6 Examples of the result of the *L* and *U* operators. Conceptually, these operators fold the bitonic sequence (black), such that the part from indices $\frac{n}{2}+1$ through *n* (light gray) is shifted into the range 1 through $\frac{n}{2}$ (black); then, *L* and *U* yield the upper (medium gray) and lower (dark gray) hull, respectively



Adaptive Bitonic Sorting. Fig. 7 Schematic, recursive diagram of a network that performs bitonic merging

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 Fig. 8). Again, the base case is the two-key sequence.

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.

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

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

which amounts to

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

As a consequence, the bitonic sorter consists of $O\left(n \log^2 n\right)$ 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*.

Derivation of Adaptive Bitonic Merging

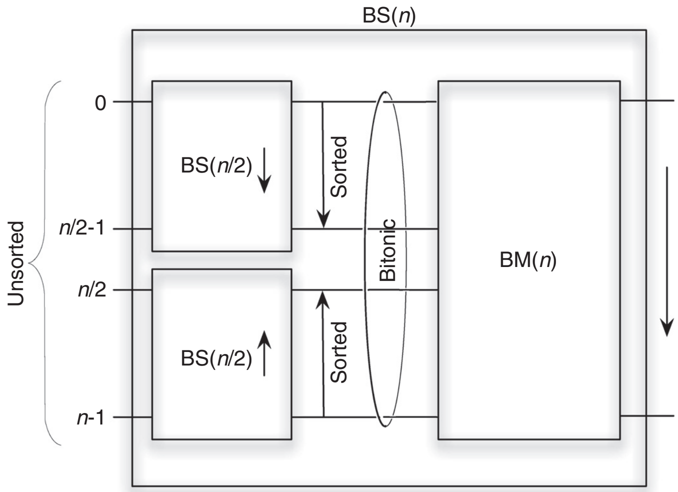
The algorithm for adaptive bitonic sorting is based on the following theorem.

Theorem 2 Let **a** be a bitonic sequence. Then, there is an index q such that

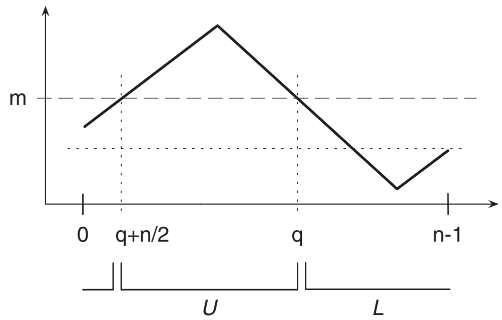
$$L\mathbf{a} = \left(a_q, \ldots, a_{q+\frac{n}{2}-1}\right) \quad (4)$$

$$U\mathbf{a} = \left(a_{q+\frac{n}{2}}, \ldots, a_{q-1}\right) \quad (5)$$

(Remember that index arithmetic is always modulo *n*.)



Adaptive Bitonic Sorting. Fig. 8 Schematic, recursive diagram of a bitonic sorting network



Adaptive Bitonic Sorting. Fig. 9 Visualization for the proof of Theorem 2

The following outline of the proof assumes, for the sake of simplicity, that all elements in **a** are distinct. Let m be the median of all a_i, that is, $\frac{n}{2}$ elements of **a** are less than or equal to m, and $\frac{n}{2}$ elements are larger. Because of Theorem 1,

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

Employing the cylinder metaphor again, the median m can be visualized as a horizontal plane $z = m$ that cuts the cylinder. Since **a** is bitonic, this plane cuts the sequence in exactly two places, that is, it partitions the sequence into two contiguous halves (actually, any horizontal plane, i.e., any percentile partitions a bitonic sequence in two contiguous halves), and since it is the median, each half must have length $\frac{n}{2}$. The indices where the cut happens are q and $q + \frac{n}{2}$. Figure 9 shows an example (in one dimension).

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

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

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

or

$$(L\mathbf{a}, U\mathbf{a}) = (\mathbf{a}^3, \mathbf{a}^2, \mathbf{a}^1, \mathbf{a}^4). \tag{7}$$

Furthermore,

$$|\mathbf{a}^1| + |\mathbf{a}^2| = |\mathbf{a}^3| + |\mathbf{a}^4| = \frac{n}{2}, \tag{8}$$

$$|\mathbf{a}^1| = |\mathbf{a}^3|, \tag{9}$$

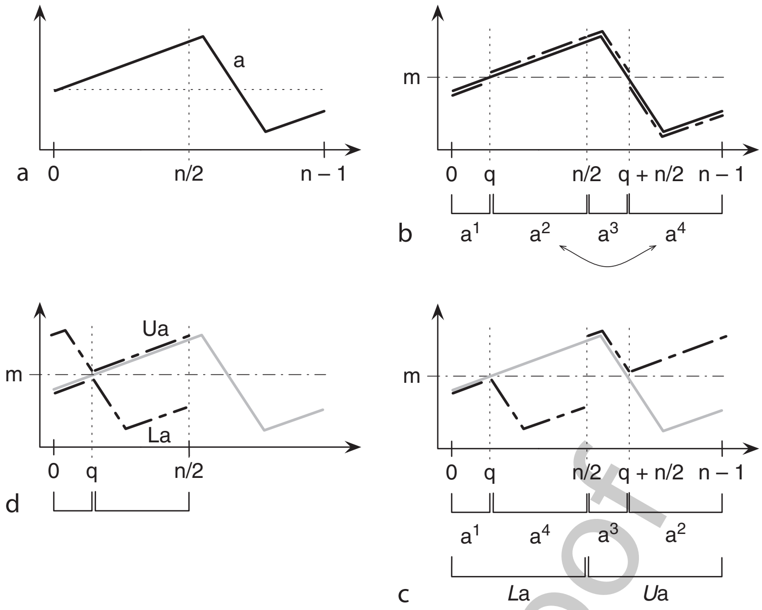
and

$$|\mathbf{a}^2| = |\mathbf{a}^4|, \tag{10}$$

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

Figure 10 illustrates this theorem by an example.

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 subsequences (see Fig. 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 $\max\{(\mathbf{a}^1, \mathbf{a}^4)\} < m < \min\{\mathbf{a}^3, \mathbf{a}^3)\}$ are obtained. The other case can be handled analogously.



Adaptive Bitonic Sorting. Fig. 10 Example illustrating Theorem 3

Remember that 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 subsequences, without ever looking at the data.

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

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

Overall, instead of performing $\frac{n}{2}$ comparisons in the first stage of the bitonic merger (see Fig. 7), the adaptive bitonic merger performs $\log\left(\frac{n}{2}\right)$ 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

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

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

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

The only question that remains is how to achieve the data rearrangement, that is, the swapping of the subsequences $\mathbf{a}^1$ and $\mathbf{a}^3$ or $\mathbf{a}^2$ and $\mathbf{a}^4$, respectively, 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 subtrees. With this technique, the overall number of operations of adaptive bitonic merging is $O(n)$. Details can be found in [5].

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 Fig. 8).

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

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 pairwise comparisons within one stage can be performed in parallel. But this is not straightforward 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 [5]. As $L\mathbf{a}$, $U\mathbf{a}$ are being constructed in one stage 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 $O\left(\frac{n}{p}\right)$ for $p \in O\left(\frac{n}{\log n}\right)$, that is, it can be executed in $(\log n)$ 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 straightforward to apply the classical sorting-by-merging approach here (see Fig. 8), which yields the *adaptive bitonic sorting* algorithm. This can be implemented on an EREW machine with p processors in $O\left(\frac{n \log n}{p}\right)$ time, for $p \in O\left(\frac{n}{\log n}\right)$.

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 $O\left(\log^2(n)\right)$ 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" [11, 12]. For the sake of simplicity, the following always assumes

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 a
output : La in the left subtree of r plus root r, and Ua
         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, ..., 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 **a**

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

construct $L\mathbf{a}$ and $U\mathbf{a}$ in the bitonic tree by 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 **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))
```

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, that is, compare for < instead of > in 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 **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 and Eqs. 6 and 7 can be distinguished simply by comparing elements $a_{\frac{n}{2}-1}$ and a_{n-1}.

This leads to 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 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 $R_{n/2}\mathbf{a} = \left(a_{\frac{n}{2}}, \ldots, a_{n-1}, a_0, \ldots, a_{\frac{n}{2}-1}\right)$ is bitonic, provided **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 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.)

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.

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 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 cannot 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 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 predetermined, 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 when the kernel is still executing for the current node. Figure 11 demonstrates the operation of the stream program using the described stream output technique.

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 [5], that is, 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 at 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.

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.

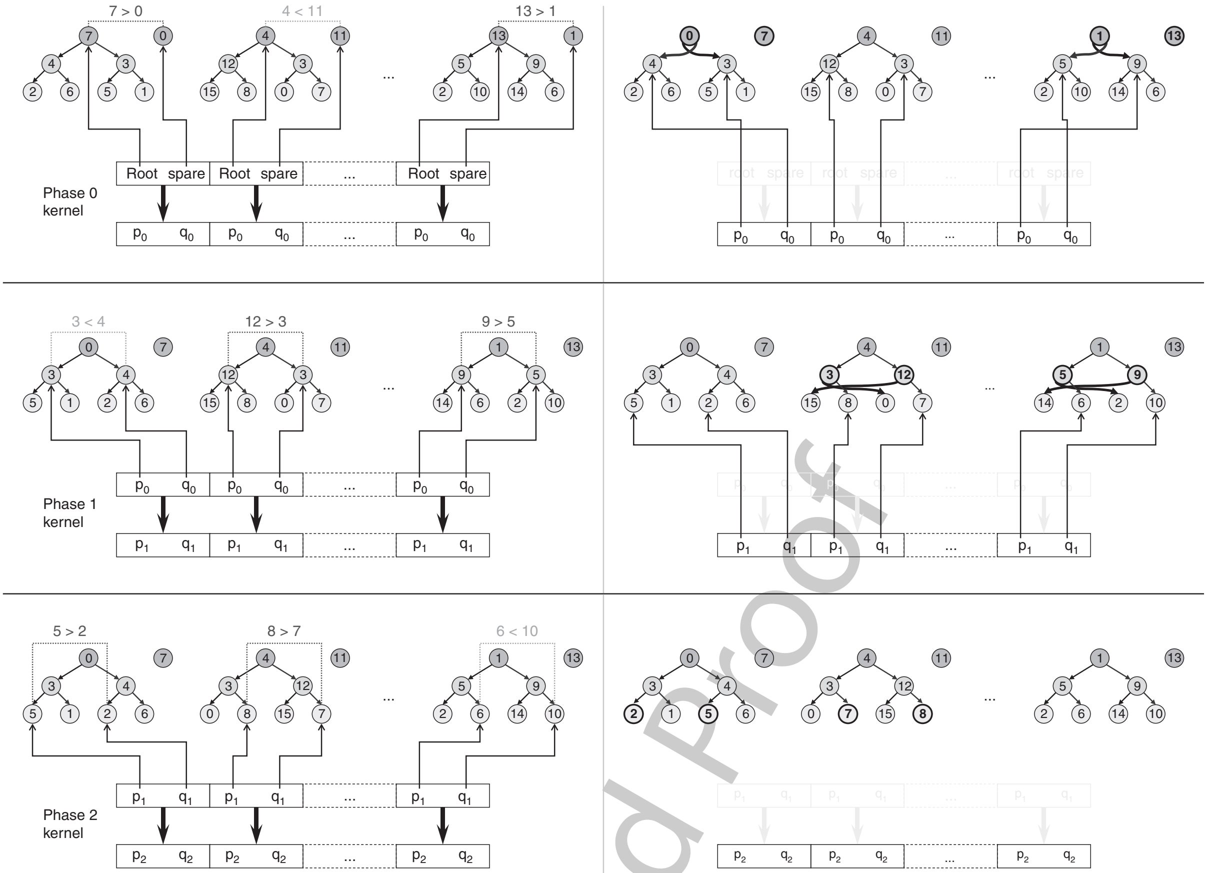
Preconditioning the Input

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, that is, 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 $j = 4$.



Adaptive Bitonic Sorting. Fig. 11 To execute several instances of the adaptive *L***a**/*U***a** construction algorithm in parallel, where each instance operates on a bitonic tree of 2^3 nodes, three phases are required. This figure illustrates the operation of these three 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 3

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 (nonadaptive) bitonic merge implementation that can be executed in a single pass of the whole stream.

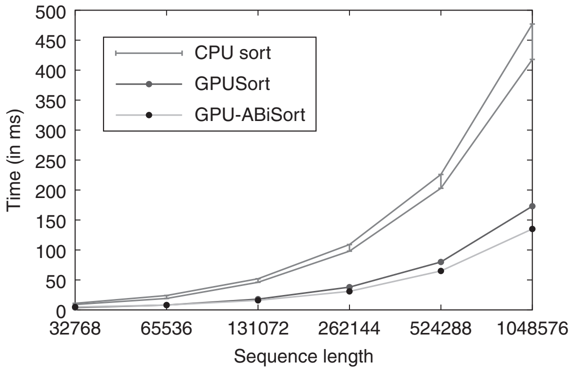
Timings

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.

The experiments described in the following compare the implementation of GPU-ABiSort of [11, 12] with sorting on the CPU using the C++ STL sort function (an optimized quicksort implementation) as well as with the (nonadaptive) bitonic sorting network implementation on the GPU by Govindaraju et al., called GPUSort [10].

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.

n	CPU sort	GPUSort	GPU-ABiSort
32,768	9–11 ms	4 ms	5 ms
65,536	19–24 ms	8 ms	8 ms
131,072	46–52 ms	18 ms	16 ms
262,144	98–109 ms	38 ms	31 ms
524,288	203–226 ms	80 ms	65 ms
1,048,576	418–477 ms	173 ms	135 ms



Adaptive Bitonic Sorting. Fig. 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)

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 speedup 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 speedup 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 speedup: according to measurements by [11], the transfer of one million key/pointer pairs from CPU to GPU and back takes in total roughly 20 ms on a PCI Express bus PC.

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).

Related Entries

- ►AKS Network
- ►Bitonic Sort
- ►Lock-Free Algorithms
- ►Scalability
- ►Speedup

Bibliographic Notes and Further Reading

As mentioned in the introduction, this line of research began with the seminal work of Batcher [4] in the late 1960s, who described parallel sorting as a network. Research of parallel sorting algorithms was reinvigorated in the 1980s, where a number of theoretical questions have been settled [1, 3, 5, 6, 14, 18].

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 [9, 11–13, 16, 17, 19].

Bibliography

1. Ajtai M, Komlós J, Szemerédi J (1983) An O(n log n) sorting network. In: Proceedings of the fifteenth annual ACM symposium on theory of computing (STOC '83), New York, NY, pp 1–9
2. Akl SG (1990) Parallel sorting algorithms. Academic, Orlando, FL
3. Azar Y, Vishkin U (1987) Tight comparison bounds on the complexity of parallel sorting. SIAM J Comput 16(3):458–464

4. Batcher KE (1968) Sorting networks and their applications. In: Proceedings of the 1968 Spring joint computer conference (SJCC), Atlanta City, NJ, volume 32, pp 307–314
5. Bilardi G, Nicolau A (1989) Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. SIAM J Comput 18(2):216–228
6. Cole R (1988) Parallel merge sort. SIAM J Comput 17(4):770–785. see Correction in SIAM J. Comput. 22, 1349
7. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. MIT Press, Cambridge, MA
8. Gibbons A, Rytter W (1988) Efficient parallel algorithms. Cambridge University Press, Cambridge, England
9. Govindaraju NK, Gray J, Kumar R, Manocha D (2006) GPUTeraSort: high performance graphics coprocessor sorting for large database management. Technical Report MSR-TR-2005-183, Microsoft Research (MSR), December 2005. In: Proceedings of ACM SIGMOD conference, Chicago, IL
10. Govindaraju NK, Raghuvanshi N, Henson M, Manocha D (2005) A cachee efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina, Chapel Hill
11. Greß A, Zachmann G (2006) GPU-ABiSort: optimal parallel sorting on stream architectures. In: Proceedings of the 20th IEEE international parallel and distributed processing symposium (IPDPS), Rhodes Island, Greece, p 45
12. Greß A, Zachmann G (2006) Gpu-abisort: Optimal parallel sorting on stream architectures. Technical Report IfI-06-11, TU Clausthal, Computer Science Department, Clausthal-Zellerfeld, Germany
13. Kipfer P, Westermann R (2005) Improved GPU sorting. In: Pharr M (ed) GPU Gems 2: programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley, Reading, MA, pp 733–746
14. Leighton T (1984) Tight bounds on the complexity of parallel sorting. In: STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing, ACM, New York, NY, USA, pp 71–80
15. Natvig L (1990) Logarithmic time cost optimal parallel sorting is not yet fast in practice! In: Proceedings supercomputing '90, New York, NY, pp 486–494
16. Purcell TJ, Donner C, Cammarano M, Jensen HW, Hanrahan P (2003) Photon mapping on programmable graphics hardware. In: Proceedings of the 2003 annual ACM SIGGRAPH/eurographics conference on graphics hardware (EGGH '03), ACM, New York, pp 41–50
17. Satish N, Harris M, Garland M (2009) Designing efficient sorting algorithms for manycore gpus. In: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS), IEEE Computer Society, Washington, DC, USA, pp 1–102
18. Schnorr CP, Shamir A (1986) An optimal sorting algorithm for mesh connected computers. In: Proceedings of the eighteenth annual ACM symposium on theory of computing (STOC), ACM, New York, NY, USA, pp 255–263
19. Sintorn E, Assarsson U (2008) Fast parallel gpu-sorting using a hybrid algorithm. J Parallel Distrib Comput 68(10):1381–1388