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 sorting can be performed in $O(n \log n)$ time, $p$ being the number of processors, and executes only $O(n)$ operations in total. Consequently, adaptive bitonic sorting can be performed in $O(n \log n/p)$ time, which is optimal. So, one of its advantages is that it executes a factor of $O(n)$ less operations than bitonic sorting. Another advantage is that it can be implemented efficiently on modern GPUs.

Discussion
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 $O(n \log n/p)$ 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 $2^{19}$) [5].

© Springer Science+Business Media LLC 2011
Fundamental Properties

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

Definition 1 (Bitonic sequence) Let $a = (a_0, \ldots, a_{n-1})$ be a sequence of numbers. Then, $a$ is bitonic, if 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.

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

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

$$LR_a = \left( \min(a_j, a_{j+\frac{n}{2}}), \ldots, \min(a_{j-1}, a_{n-1}) \right),$$

$$= R_j L a.$$

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 $a,$

$$\max(La) \leq \min(Ua).$$

Moreover, $La$ and $Ua$ are bitonic too.

In other words, each element of $La$ is less than or equal to each element of $Ua$.

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:

$$BM_n(a) = \left( BM_{\frac{n}{2}}(La), BM_{\frac{n}{2}}(Ua) \right).$$

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 $a$ of length $n$, one can show that

$$BM_n(a) = \text{Sorted}(a).$$

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 \( O(n \log \frac{n}{2}) \) 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

\[
L a = (a_q, \ldots, a_{q+\frac{n}{2}-1}) \tag{4}
\]

\[
U a = (a_{q+\frac{n}{2}}, \ldots, a_{q-1}) \tag{5}
\]

(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 1.

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,

\[
\max\{L_a\} \leq m < \min\{U_a\}.
\]

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 into 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 \( (a^1, a^2, a^3, a^4) \) such that either

\[
(L_a, U_a) = (a^1, a^4, a^3, a^2) \quad (6)
\]

or

\[
(L_a, U_a) = (a^3, a^2, a^1, a^4). \quad (7)
\]

Furthermore,

\[
|a^1| + |a^2| = |a^3| + |a^4| = \frac{n}{2}, \quad (8)
\]

\[
|a^1| = |a^4|, \quad (9)
\]

and

\[
|a^3| = |a^2|, \quad (10)
\]

where \( |a| \) 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 \( \max\{a^1\} < m < \min\{a^3\} \), nothing will change between those two subsequences (see Fig. 10). However, in that case \( \min\{a^2\} > m > \max\{a^4\} \); therefore, by swapping \( a^2 \) and \( a^4 \) (which have equal length), the bounds \( \max\{(a^1, a^4)\} < m < \min\{(a^3, a^2)\} \) 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 \( a_i, a_{i+\frac{n}{2}} \). 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 \( a^1 \) and \( a^3 \) or \( a^2 \) and \( a^4 \), 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 naive 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 $L_a, U_a$ 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 $O(\frac{n}{p} \log \frac{n}{p})$ for $p \in O(\log n)$, that is, 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 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 $O(\frac{n \log n}{p})$ time, for $p \in O(\frac{n}{\log n})$.

**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” [11, 12]. For the sake of simplicity, the following always assumes

### Algorithm 1: Adaptive construction of $L_a$ and $U_a$

(one stage of adaptive bitonic merging)

**input**: Bitonic tree, with root node $r$ and extra node $e$, representing bitonic sequence $a$

**output**: $L_a$ in the left subtree of $r$ plus root $r$, and $U_a$ in the right subtree of $r$ plus extra node $e$

// phase 0: determine case

if $\text{value}(r) < \text{value}(e)$ then
  case = 1
else
  case = 2

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

else
  ($p, q$) = ($\text{right}(p), \text{right}(q)$)
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 \( \text{sort}(a) \) when traversed in-order)

- Construct \( L_a \) and \( U_a \) in the bitonic tree by 1 call merging recursively with \( \text{left}(r) \) as root and \( r \) as extra node
- Call merging recursively with \( \text{right}(r) \) as root and \( e \) as extra node

increasing sorting direction, and it is thus not explicitly specified. As noted above, the sorting direction must be reversed in the right branch of the recursion in the bitonic sort, 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 \( (L_a, U_a) \) 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 \( a_{q-1} \) and \( a_{n-1} \).

This leads to 1. Note that the root of the bitonic tree stores element \( a_{q-1} \) 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 \( R_{n/2} = (a_2, \ldots, a_{n-1}, a_0, \ldots, a_{2-1}) \) is bitonic; provided \( a \) is bitonic too. This operation basically amounts to swapping the two pointers \( \text{left}(\text{root}) \) and \( \text{right}(\text{root}) \). The simplified construction of \( L_a \) and \( U_a \) 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

Algorithm 3: Simplified adaptive construction of \( L_a \) and \( U_a \)

**input**: Bitonic tree, with root node \( r \) and extra node \( e \), representing bitonic sequence \( a \)

**output**: \( L_a \) in the left subtree of \( r \) plus root \( r \), and \( U_a \) in the right subtree of \( r \) plus extra node \( e \)

// phase 0
if \( \text{value}(r) > \text{value}(e) \) then
    swap \( \text{value}(r) \) and \( \text{value}(e) \)
    swap pointers \( \text{left}(r) \) and \( \text{right}(r) \)
    \((p, q) = (\text{left}(r), \text{right}(r))\)
for \( i = 1, \ldots, \log n - 1 \) do
   // phase i
   if \( \text{value}(p) > \text{value}(q) \) then
      swap \( \text{value}(p) \) and \( \text{value}(q) \)
      swap pointers \( \text{left}(p) \) and \( \text{left}(q) \)
   else
      \((p, q) = (\text{left}(p), \text{left}(q))\)

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 \( L_a \) and \( U_a \) is executed on \( 2^k \) subtrees. Therefore, \( 2^{\log n - j + k} \) instances of the \( L_a \) / \( U_a \) construction algorithm can be executed in parallel during that stage. On a stream architecture, this potential parallelism can be exploited by allocating a stream consisting of \( 2^{\log n - j + k} \) elements and executing a so-called kernel on each element.

The \( L_a \) / \( U_a \) 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
384 preserved from one phase of the algorithm to the next
385 one are just two node pointers (p and q) per kernel
386 instance. Thus, each of the $2^{\log n - j + k}$ elements of the allo-
387 cated stream consist of exactly these two node pointers.
388 When the kernel is invoked on that stream, each kernel
389 instance reads a pair of node pointers, (p,q), from the
390 stream, performs one phase of the La/UA construction
391 algorithm, and finally writes the updated pair of node
392 pointers (p,q) back to the stream.

393 Eliminating Random-Access Writes
394 Since GPUs do not support random-access writes (at
395 least, for almost all practical purposes, random-access
396 writes would kill any performance gained by the paral-
397 lelism) the kernel has to be implement so that it modifies
398 node pairs (p,q) of the bitonic tree without random-
399 access writes. This means that it can output node pairs
400 from the kernel only via linear stream write. But this
401 way it cannot write a modified node pair to its original
402 location from where it was read. In addition, it cannot
403 not simply take an input stream (containing a bitonic
404 tree) and produce another output stream (containing
405 the modified bitonic tree), because then it would have to
406 process the nodes in the same order as they are stored in
407 memory, but the adaptive bitonic merge processes them
408 in a random, data-dependent order.

409 Fortunately, the bitonic tree is a linked data structure
410 where all nodes are directly or indirectly linked to the
411 root (except for the extra node). This allows us to change
412 the location of nodes in memory during the merge algo-
413 rithm as long as the child pointers of their respective
414 parent nodes are updated (and the root and extra node
415 of the bitonic tree are kept at well-defined memory loca-
416 tions). This means that for each node that is modified its
417 parent node has to be modified also, in order to update
418 its child pointers.

419 Notice that 3 basically traverses the bitonic tree
down along a path, changing some of the nodes as nec-
420 essary. The strategy is simple: simply output every node
421 visited along this path to a stream. Since the data lay-
422 out is fixed and predetermined, the kernel can store the
423 index of the children with the node as it is being writ-
424 ten to the output stream. One child address remains
425 the same anyway, while the other is determined when
426 the kernel is still executing for the current node. Fig-
427 ure 11 demonstrates the operation of the stream pro-
428 gram 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 exe-
439 cution of a new stage can start at every other step of the
440 algorithm.

441 The only difference from the simple implementa-
442 tion is that kernels now must write to parts of the output
443 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 program-
447 ming: allocate two such streams and alternatingly use
448 one of them as input and the other one as output stream.
449

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.

455 The same technique can be applied here by imple-
456 menting such a local sort as a kernel program. However,
457 since there is no random write access to non-temporary
458 memory from a kernel, the number of values that can be
459 sorted locally by a kernel is restricted by the number of
460 temporary registers.

461 On recent GPUs, the maximum output data size of
462 a kernel is $16 \times 4$ bytes. Since usually the input consists
463 of key/pointer pairs, the method starts with a local sort
464 of 8-key/pointer pairs per kernel. For such small num-
465 bers of keys, an algorithm with asymptotic complexity
466 of $O(n)$ performs faster than asymptotically optimal
467 algorithms.

468 After the local sort, a further stream operation
469 converts the resulting sorted subsequences of length
470 8 pairwise to bitonic trees, each containing 16 nodes. 471 Thereafter, the GPU-ABiSort approach can be applied
472 as described above, starting with $j = 4$. 473
Adaptive Bitonic Sorting. Fig. 11 To execute several instances of the adaptive LA/UA construction algorithm in parallel, where each instance operates on a bitonic tree of $2^n$ 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.
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 \( n \geq 2^{27} \). Furthermore, up to the maximum tested sequence length \( n = 2^{27} = 1,048,576 \), 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.

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

Adaptive Bitonic Sorting

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 \( n \geq 2^{27} \). Furthermore, up to the maximum tested sequence length \( n = 2^{27} = 1,048,576 \), 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, 547 streaming architectures. This spurred the development of a number of practical implementations [9, 11–13, 16, 17, 19].

### Bibliography