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

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].

70 **Fundamental Properties**

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

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

79 Figure 1 shows some examples of bitonic sequences.
80 In the following, it will be easier to understand
81 any reasoning about bitonic sequences, if one consid-
82 ers them as being arranged in a circle or on a cylinder:
83 then, there are only two inflection points around the cir-
84 cle. This is justified by Definition 1. Figure 2 depicts an
85 example in this manner.

As a consequence, all index arithmetic is understood 86
modulo n, that is, index $i + k \equiv i + k \bmod n$, *unless* 87
otherwise noted, so indices range from 0 through $n-1$. 88

As mentioned above, adaptive bitonic sorting can be 89
regarded as a variant of bitonic sorting, which is in order 90
to capture the notion of "rotational invariance" (in some 91
sense) of bitonic sequences; it is convenient to define the 92
following *rotation operator*. 93

Definition 2 (Rotation) Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$ and 94
$j \in \mathbb{N}$. We define a rotation as an operator R_j on the 95
sequence $\mathbf{a}$: 96

$$R_j\mathbf{a} = (a_j, a_{j+1}, \ldots, a_{j+n-1})$$ 97

This operation is performed by the network shown 98
in Fig. 4. Such networks are comprised of elementary 99
*comparators* (see Fig. 3). 100

Two other operators are convenient to describe 101
sorting. 102
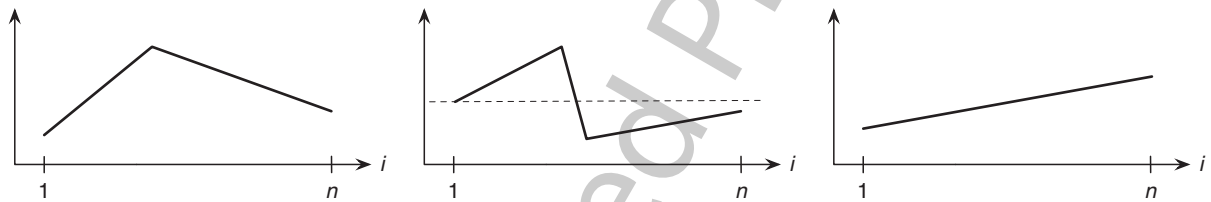
Adaptive Bitonic Sorting. Fig. 1 Three examples of sequences that are bitonic. Obviously, the mirrored sequences (either way) are bitonic, too
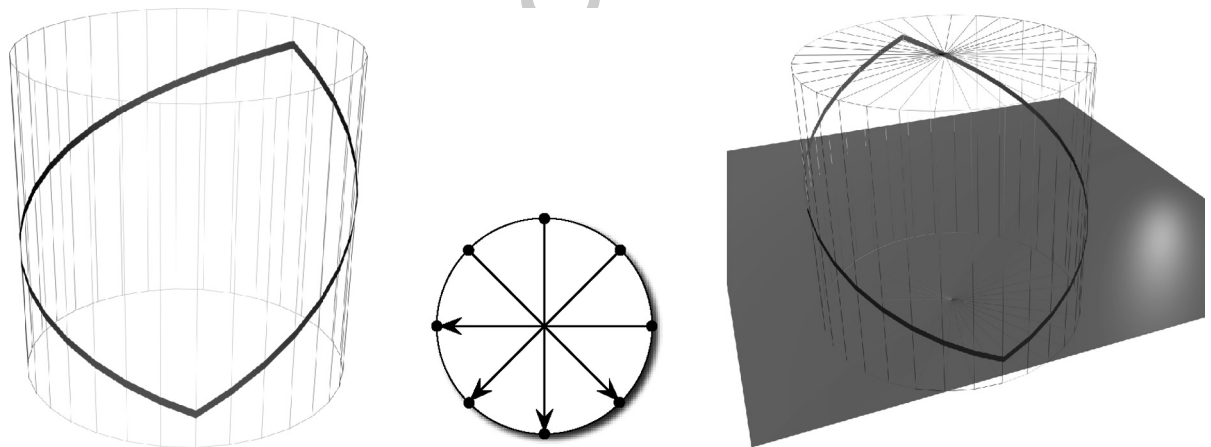
Adaptive Bitonic Sorting. Fig. 2 *Left*: according to their definition, bitonic sequences can be regarded as lying on a cylinder or as being arranged in a circle. As such, they consist of one monotonically increasing and one decreasing part. *Middle*: in this point of view, the network that performs 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
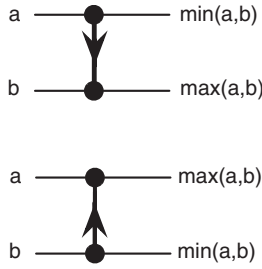
Adaptive Bitonic Sorting. Fig. 3 Comparator/exchange elements
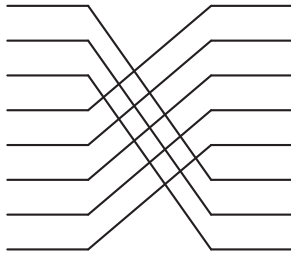
Adaptive Bitonic Sorting. Fig. 4 A network that performs the rotation operator
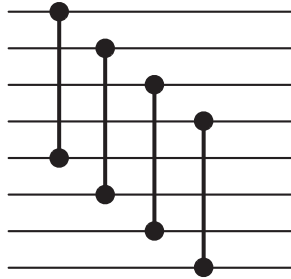
Adaptive Bitonic Sorting. Fig. 5 A network that performs the L and U operators

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 $\mathbf{a}$,

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

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 $1 \le j < \frac{n}{2}$. 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

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

$$\left.\min\left(a_{j-1}, a_{j-1+\frac{n}{2}}\right)\right)$$

$$= R_j L\mathbf{a}.$$

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 $\mathbf{a}$,

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

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:

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

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 Fig. 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 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,
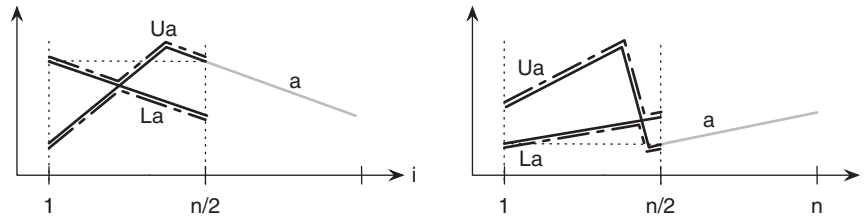
Adaptive Bitonic Sorting. Fig. 6 Examples of the result of the *L* and *U* operators. Conceptually, these operators fold the bitonic sequence (black), such that the part from indices $\frac{n}{2} + 1$ through n (light gray) is shifted into the range 1 through $\frac{n}{2}$ (black); then, *L* and *U* yield the upper (medium gray) and lower (dark gray) hull, respectively
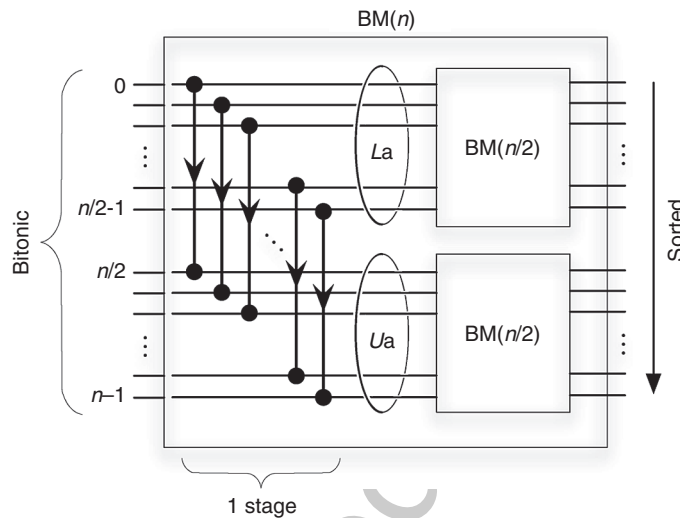
Adaptive Bitonic Sorting. Fig. 7 Schematic, recursive diagram of a network that performs bitonic merging

157 and produces a sorted sequence either up or down.
158 Like the BM, it is defined recursively, consisting of two
159 smaller bitonic sorters and a bitonic merger (see Fig. 8).
160 Again, the base case is the two-key sequence.

161 **Analysis of the Number of Operations of**
162 **Bitonic Sorting**
163 Since a bitonic sorter basically consists of a number of
164 bitonic mergers, it suffices to look at the total number of
165 comparisons of the latter.
166 The total number of comparators, $C(n)$, in the
167 bitonic merger BM_n is given by:

168 $$C(n) = 2C\left(\frac{n}{2}\right) + \frac{n}{2}, \quad \text{with } C(2) = 1,$$

169 which amounts to

170 $$C(n) = \frac{1}{2}n \log n.$$

171 As a consequence, the bitonic sorter consists of
172 $O(n \log^2 n)$ comparators.

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

177 **Derivation of Adaptive Bitonic Merging**
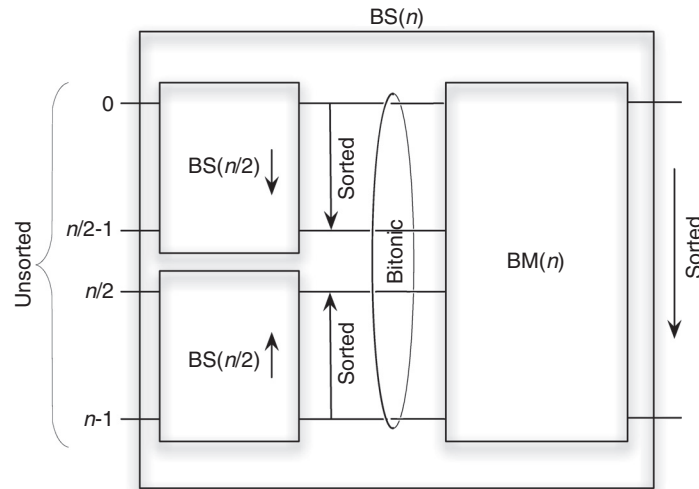178 The algorithm for adaptive bitonic sorting is based on
179 the following theorem.

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

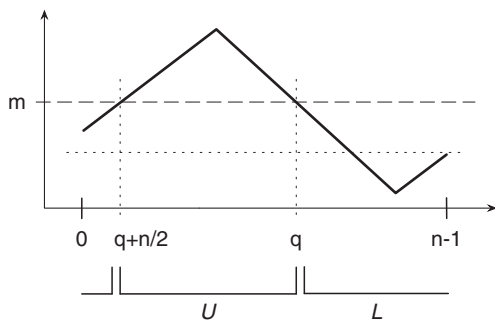182 $$L\mathbf{a} = \left(a_q, \ldots, a_{q+\frac{n}{2}-1}\right) \quad (4)$$

183 $$U\mathbf{a} = \left(a_{q+\frac{n}{2}}, \ldots, a_{q-1}\right) \quad (5)$$

184 (Remember that index arithmetic is always mod-
185 ulo 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

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

191 $$\max\{L\mathbf{a}\} \le m < \min\{U\mathbf{a}\}.$$

192 Employing the cylinder metaphor again, the median
193 m can be visualized as a horizontal plane $z = m$ that
194 cuts the cylinder. Since **a** is bitonic, this plane cuts the
195 sequence in exactly two places, that is, it partitions the
196 sequence into two contiguous halves (actually, any hor-
197 izontal plane, i.e., any percentile partitions a bitonic
198 sequence in two contiguous halves), and since it is
199 the median, each half must have length $\frac{n}{2}$. The indices

200 where the cut happens are q and $q + \frac{n}{2}$. Figure 9 shows
201 an example (in one dimension).

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

Theorem 3 Any bitonic sequence **a** can be partitioned 204
into four subsequences $(\mathbf{a}^1, \mathbf{a}^2, \mathbf{a}^3, \mathbf{a}^4)$ such that either 205

$$(L\mathbf{a}, U\mathbf{a}) = (\mathbf{a}^1, \mathbf{a}^4, \mathbf{a}^3, \mathbf{a}^2) \qquad (6)$$ 206

or 207

$$(L\mathbf{a}, U\mathbf{a}) = (\mathbf{a}^3, \mathbf{a}^2, \mathbf{a}^1, \mathbf{a}^4). \qquad (7)$$ 208

Furthermore, 209

$$|\mathbf{a}^1| + |\mathbf{a}^2| = |\mathbf{a}^3| + |\mathbf{a}^4| = \frac{n}{2}, \qquad (8)$$ 210

211

$$|\mathbf{a}^1| = |\mathbf{a}^3|, \qquad (9)$$ 212

and 213

$$|\mathbf{a}^2| = |\mathbf{a}^4|, \qquad (10)$$ 214

where $|\mathbf{a}|$ denotes the length of sequence **a**. 215

Figure 10 illustrates this theorem by an example. 216

This theorem can be proven fairly easily too: the 217
length of the subsequences is just q and $\frac{n}{2} - q$, where q is 218
the same as in Theorem 2. Assuming that $\max\{\mathbf{a}^1\} <$ 219
$m < \min\{\mathbf{a}^3\}$, nothing will change between those 220
two subsequences (see Fig. 10). However, in that case 221
$\min\{\mathbf{a}^2\} > m > \max\{\mathbf{a}^4\}$; therefore, by swap- 222
ping $\mathbf{a}^2$ and $\mathbf{a}^4$ (which have equal length), the bounds 223
$\max\{(\mathbf{a}^1, \mathbf{a}^4)\} < m < \min\{\mathbf{a}^3, \mathbf{a}^3\}$ are obtained. The 224
other case can be handled analogously. 225
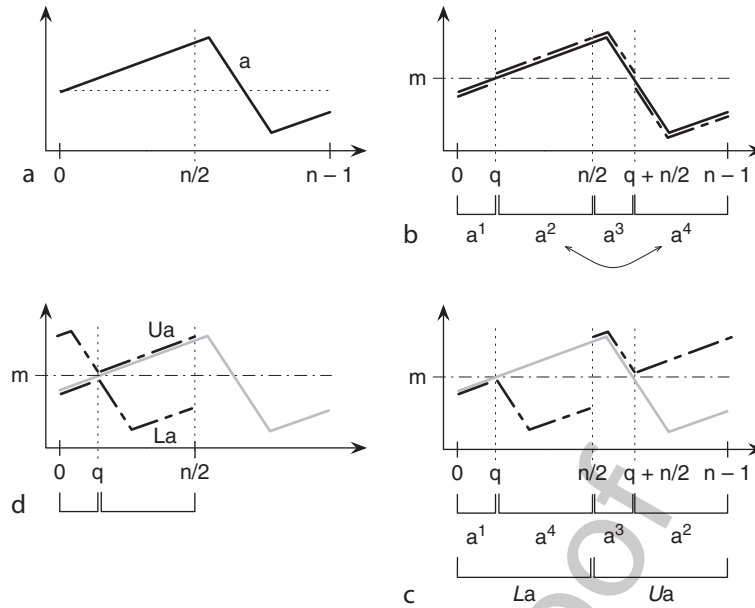
Adaptive Bitonic Sorting. Fig. 10 Example illustrating Theorem 3

226 Remember that there are $\frac{n}{2}$ comparator-and-
227 exchange elements, each of which compares a_i and
228 $a_{i+\frac{n}{2}}$. They will perform exactly this exchange of sub-
229 sequences, without ever looking at the data.

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

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

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

242 Let $C(n)$ be the total number of comparisons per-
243 formed by adaptive bitonic merging, in the worst case.
244 Then

245
$$C(n) = 2C\left(\frac{n}{2}\right) + \log(n) = \sum_{i=0}^{k-1} 2^i \log\left(\frac{n}{2^i}\right),$$

with $C(2) = 1, C(1) = 0$ and $n = 2^k$. This amounts to   246

$$C(n) = 2n - \log n - 2.$$ 247

The only question that remains is how to achieve the   248
data rearrangement, that is, the swapping of the subse-   249
quences $\mathbf{a}^1$ and $\mathbf{a}^3$ or $\mathbf{a}^2$ and $\mathbf{a}^4$, respectively, without   250
sacrificing the worst-case performance of $O(n)$. This   251
can be done by storing the keys in a perfectly balanced   252
tree (assuming $n = 2^k$), the so-called bitonic tree. (The   253
tree can, of course, store only $2^k - 1$ keys, so the n-th   254
key is simply stored separately.) This tree is very similar   255
to a search tree, which stores a monotonically increas-   256
ing sequence: when traversed in-order, the bitonic tree   257
produces a sequence that lists the keys such that there   258
are exactly two inflection points (when regarded as a   259
circular list).   260

Instead of actually copying elements of the sequence   261
in order to achieve the exchange of subsequences, the   262
adaptive bitonic merging algorithm swaps $O(\log n)$ 263
pointers in the bitonic tree. The recursion then works on   264
the two subtrees. With this technique, the overall num-   265
ber of operations of adaptive bitonic merging is $O(n)$.   266
Details can be found in [5].   267

Clearly, the adaptive bitonic sorting algorithm needs   268
$O(n \log n)$ operations in total, because it consists of   269
$\log(n)$ many complete merge stages (see Fig. 8).   270

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: $L\mathbf{a}$ 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, ..., 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 $i = 1, \ldots, \log n - 1$ **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)`)

---

329 increasing sorting direction, and it is thus not explicitly
330 specified. As noted above, the sorting direction must
331 be reversed in the right branch of the recursion in the
332 bitonic sorter, which basically amounts to reversing the
333 comparison direction of the values of the keys, that is,
334 compare for < instead of > in 3.

335   As noted above, the bitonic tree stores the sequence
336 $(a_0, \ldots, a_{n-2})$ in in-order, and the key a_{n-1} is stored in
337 the *extra node*. As mentioned above, an algorithm that
338 constructs $(L\mathbf{a}, U\mathbf{a})$ from **a** can traverse this bitonic tree
339 and swap pointers as necessary. The index q, which is
340 mentioned in the proof for Theorem 3, is only deter-
341 mined implicitly. The two different cases that are men-
342 tioned in Theorem 3 and Eqs. 6 and 7 can be distin-
343 guished simply by comparing elements $a_{\frac{n}{2}-1}$ and a_{n-1}.

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

349   Because branches are very costly on GPUs, one
350 should avoid as many conditionals in the inner loops
351 as possible. Here, one can exploit the fact that $R_{n/2}\mathbf{a} =$
352 $\left(a_{\frac{n}{2}}, \ldots, a_{n-1}, a_0, \ldots, a_{\frac{n}{2}-1}\right)$ is bitonic, provided **a** is
353 bitonic too. This operation basically amounts to swap-
354 ping the two pointers left(root) and right(root). The
355 simplified construction of $L\mathbf{a}$ and $U\mathbf{a}$ is presented in 3.
356 (Obviously, the simplified algorithm now really needs
357 trees with pointers, whereas Bilardi's original bitonic
358 tree could be implemented pointer-less (since it is a
359 complete tree). However, in a real-world implementa-
360 tion, the keys to be sorted must carry pointers to some

"payload" data anyway, so the additional memory over- 361
head incurred by the child pointers is at most a factor 362
1.5.) 363

Outline of the Implementation   364

As explained above, on each recursion level j = 365
$1, \ldots, \log(n)$ of the adaptive bitonic sorting algorithm, 366
$2^{\log n - j + 1}$ bitonic trees, each consisting of 2^{j-1} nodes, 367
have to be merged into $2^{\log n - j}$ bitonic trees of 2^j nodes. 368
The merge is performed in j stages. In each stage k = 369
$0, \ldots, j-1$, the construction of $L\mathbf{a}$ and $U\mathbf{a}$ is executed on 370
2^k subtrees. Therefore, $2^{\log n - j} \cdot 2^k$ instances of the $L\mathbf{a}$ / $U\mathbf{a}$ 371
construction algorithm can be executed in parallel dur- 372
ing that stage. On a stream architecture, this potential 373
parallelism can be exposed by allocating a stream con- 374
sisting of $2^{\log n - j + k}$ elements and executing a so-called 375
kernel on each element. 376

The $L\mathbf{a}$ / $U\mathbf{a}$ construction algorithm consists of $j - k$ 377
phases, where each phase reads and modifies a pair 378
of nodes, (p, q), of a bitonic tree. Assume that a ker- 379
nel implementation performs the operation of a single 380
phase of this algorithm. (How such a kernel implemen- 381
tation is realized without random-access writes will be 382
described below.) The temporary data that have to be 383

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 $L\mathbf{a}/U\mathbf{a}$ 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 can-
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
420 down along a path, changing some of the nodes as nec-
421 essary. The strategy is simple: simply output every node
422 visited along this path to a stream. Since the data lay-
423 out is fixed and predetermined, the kernel can store the
424 index of the children with the node as it is being writ-
425 ten to the output stream. One child address remains
426 the same anyway, while the other is determined when
427 the kernel is still executing for the current node. Fig-
428 ure 11 demonstrates the operation of the stream pro-
429 gram using the described stream output technique.

Complexity 430

431 A simple implementation on the GPU would need
432 $O(\log^2 n)$ phases (or "passes" in GPU parlance) in
433 total for adaptive bitonic sorting, which amounts to
434 $O(\log^3 n)$ operations in total.

435 This is already very fast in practice. However, the
436 optimal complexity of $O(\log n)$ passes can be achieved
437 exactly as described in the original work [5], that is,
438 phase i of a stage k can be executed immediately after
439 phase $i + 1$ of stage $k - 1$ has finished. Therefore, the exe-
440 cution of a new stage can start at every other step of the
441 algorithm.

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

GPU-Specific Details 445

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

Preconditioning the Input 450

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

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

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

469 After the local sort, a further stream operation
470 converts the resulting sorted subsequences of length
471 8 pairwise to bitonic trees, each containing 16 nodes.
472 Thereafter, the GPU-ABiSort approach can be applied
473 as described above, starting with $j = 4$.
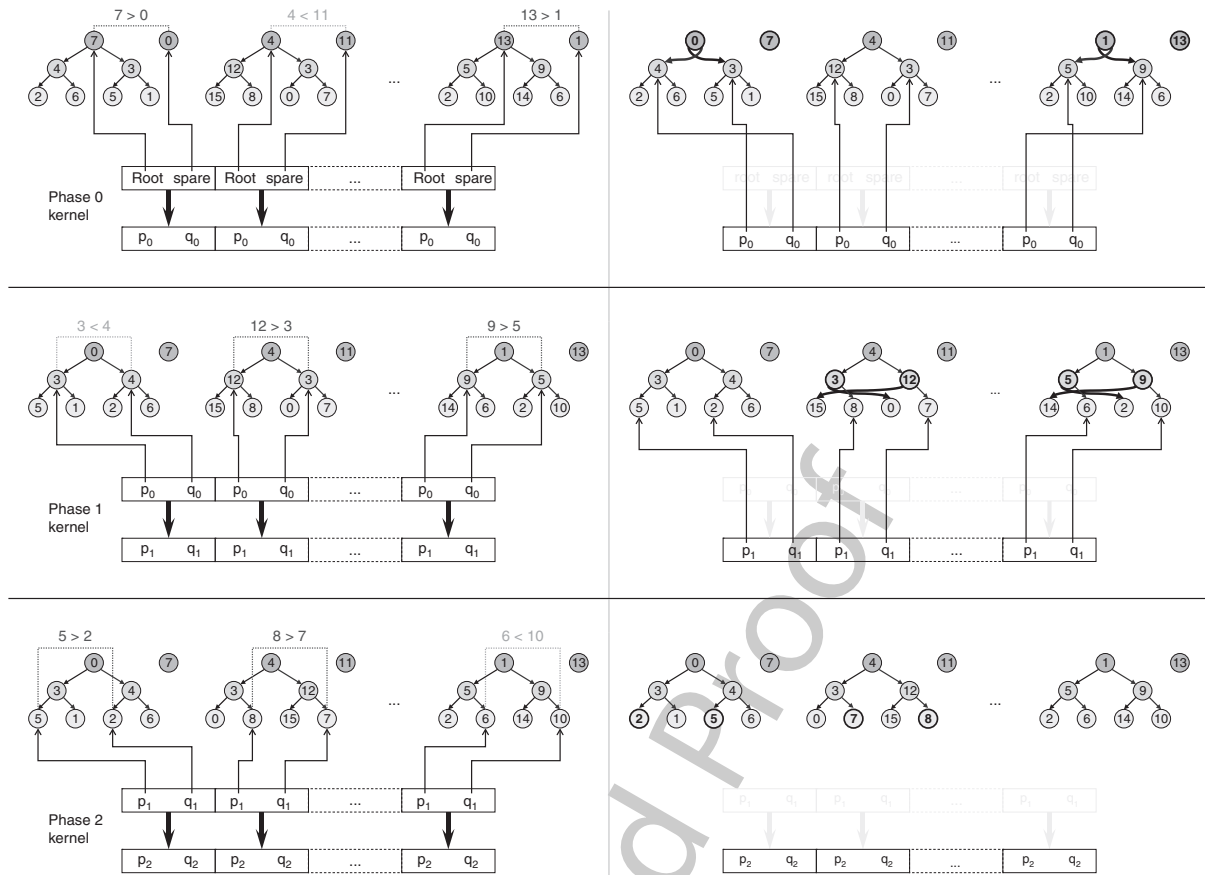
Adaptive Bitonic Sorting. Fig. 11 To execute several instances of the adaptive $L\mathbf{a}/U\mathbf{a}$ construction algorithm in parallel, where each instance operates on a bitonic tree of 2^3 nodes, three phases are required. This figure illustrates the operation of these three phases. On the left, 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

474 The Last Stage of Each Merge

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

480 Timings

481 The following experiments were done on arrays consist-
482 ing of key/pointer pairs, where the key is a uniformly
483 distributed random 32-bit floating point value and the
484 pointer a 4-byte address. Since one can assume (without
485 loss of generality) that all pointers in the given array are

unique, these can be used as secondary sort keys for the 486
adaptive bitonic merge. 487

The experiments described in the following com- 488
pare the implementation of GPU-ABiSort of [11, 12] with 489
sorting on the CPU using the C++ STL sort function (an 490
optimized quicksort implementation) as well as with the 491
(nonadaptive) bitonic sorting network implementation 492
on the GPU by Govindaraju et al., called GPUSort [10]. 493

Contrary to the CPU STL sort, the timings of GPU- 494
ABiSort do not depend very much on the data to be 495
sorted, because the total number of comparisons per- 496
formed by the adaptive bitonic sorting is not data- 497
dependent. 498

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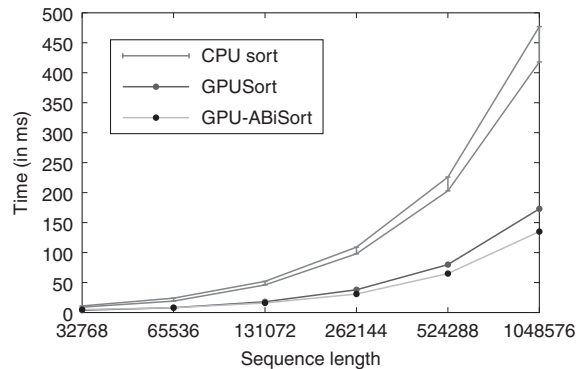
Adaptive Bitonic Sorting. Fig. 12 Timings on a GeForce 7800 system. (There are two curves for the CPU sort, so as to visualize that its running time is somewhat data-dependent)

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

558   4. Batcher KE (1968) Sorting networks and their applications. In:
559   Proceedings of the 1968 Spring joint computer conference (SJCC),
560   Atlanta City, NJ, volume 32, pp 307–314
561   5. Bilardi G, Nicolau A (1989) Adaptive bitonic sorting: An optimal
562   parallel algorithm for shared-memory machines. SIAM J Comput
563   18(2):216–228
564   6. Cole R (1988) Parallel merge sort. SIAM J Comput 17(4):770–785.
565   see Correction in SIAM J. Comput. 22, 1349
566   7. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduc-
567   tion to algorithms, 3rd edn. MIT Press, Cambridge, MA
568   8. Gibbons A, Rytter W (1988) Efficient parallel algorithms. Cam-
569   bridge University Press, Cambridge, England
570   9. Govindaraju NK, Gray J, Kumar R, Manocha D (2006) GPUT-
571   eraSort: high performance graphics coprocessor sorting for
572   large database management. Technical Report MSR-TR-2005-183,
573   Microsoft Research (MSR), December 2005. In: Proceedings of
574   ACM SIGMOD conference, Chicago, IL
575 10. Govindaraju NK, Raghuvanshi N, Henson M, Manocha D (2005)
576   A cachee efficient sorting algorithm for database and data min-
577   ing computations using graphics processors. Technical report,
578   University of North Carolina, Chapel Hill
579 11. Greß A, Zachmann G (2006) GPU-ABiSort: optimal parallel
580   sorting on stream architectures. In: Proceedings of the 20th
581   IEEE international parallel and distributed processing sympo-
582   sium (IPDPS), Rhodes Island, Greece, p 45
583 12. Greß A, Zachmann G (2006) Gpu-abisort: Optimal parallel
584   sorting on stream architectures. Technical Report IfI-06-11, TU
585   Clausthal, Computer Science Department, Clausthal-Zellerfeld,
586   Germany
587 13. Kipfer P, Westermann R (2005) Improved GPU sorting. In:
588   Pharr M (ed) GPU Gems 2: programming techniques for
589   high-performance graphics and general-purpose computation.
590   Addison-Wesley, Reading, MA, pp 733–746
591 14. Leighton T (1984) Tight bounds on the complexity of parallel
592   sorting. In: STOC '84: Proceedings of the sixteenth annual ACM
593   symposium on Theory of computing, ACM, New York, NY, USA,
594   pp 71–80
595 15. Natvig L (1990) Logarithmic time cost optimal parallel sorting is
596   not yet fast in practice! In: Proceedings supercomputing '90, New
597   York, NY, pp 486–494
598 16. Purcell TJ, Donner C, Cammarano M, Jensen HW, Hanrahan P
599   (2003) Photon mapping on programmable graphics hardware. In:
600   Proceedings of the 2003 annual ACM SIGGRAPH/eurographics
601   conference on graphics hardware (EGGH '03), ACM, New York,
602   pp 41–50
603 17. Satish N, Harris M, Garland M (2009) Designing efficient sorting
604   algorithms for manycore gpus. In: Proceedings of the 2009 IEEE
605   International Symposium on Parallel and Distributed Process-
606   ing (IPDPS), IEEE Computer Society, Washington, DC, USA, pp
607   1–102
608 18. Schnorr CP, Shamir A (1986) An optimal sorting algorithm for
609   mesh connected computers. In: Proceedings of the eighteenth
610   annual ACM symposium on theory of computing (STOC), ACM,
611   New York, NY, USA, pp 255–263
612 19. Sintorn E, Assarsson U (2008) Fast parallel gpu-sorting using a
613   hybrid algorithm. J Parallel Distrib Comput 68(10):1381–1388