A novel sorting algorithm for many-core architectures based on adaptive bitonic sort

Hagen Peters, Ole Schulz-Hildebrandt, Norbert Luttenberger
Department of Computer Science
CAU Kiel
Kiel, Germany
{hap,osh,nl}@informatik.uni-kiel.de

Abstract—Adaptive bitonic sort is a well known merge-based parallel sorting algorithm. It achieves optimal complexity using a complex tree-like data structure called a bitonic tree.

Due to this, using adaptive bitonic sort together with other algorithms usually implies converting bitonic trees to arrays and vice versa. This makes adaptive bitonic sort inappropriate in the context of hybrid sorting algorithms where frequent switches between algorithms are performed.

In this article we present a novel optimal sorting algorithm that is based on an approach similar to adaptive bitonic sort. Our approach does not use bitonic trees but uses the input array together with some additional information. Using this approach it is trivial to switch between adaptive bitonic sort and other algorithms.

We present an implementation of a hybrid algorithm for GPUs based on bitonic sort and our novel algorithm. This implementation turns out to be the fastest comparison-based sorting algorithm for GPUs found in literature.

Keywords-sorting; parallel; many-core; CUDA; bitonic sort

I. INTRODUCTION

Sorting is one of the most fundamental problems in computer science. Many sorting algorithms for different architectures and with distinct properties (e.g. comparison-based, in-place, stable etc.) have been developed.

Adaptive bitonic sort [3] is an optimal comparison-based sorting algorithm for shared-memory architectures. Its time complexity is \(O(N \cdot \log N)\). This optimal complexity is achieved by using a complex tree-like data structure called a bitonic tree.

Although adaptive bitonic sort has optimal complexity, it performs relatively slowly for shorter sequences. In such cases it is an often used approach to combine an optimal algorithm with another algorithm (not necessarily optimal) that is fast on shorter sequences, i.e. to develop a hybrid algorithm. Unfortunately, developing hybrid algorithms based on adaptive bitonic sort using other algorithms like bitonic sort or odd-even sort is not trivial. These algorithms operate on arrays instead of bitonic trees and thus bitonic trees have to be converted to arrays before these algorithms can be used.

In this contribution we present a novel optimal sorting algorithm. Our algorithm is based on the same idea as adaptive bitonic sort but in contrast our approach does not use a special data structure. Our algorithm uses the array representation of the input sequence together with some additional information instead, thus it is trivial to switch between our novel algorithm and other algorithms.

We present an implementation of a hybrid algorithm based on bitonic sort and our algorithm for GPUs. This implementation turns out to be the fastest comparison-based sorting algorithm for GPUs known to the authors.

The report is organized as follows: We first give short introductions to bitonic sort and adaptive bitonic sort. In section III we present our novel sorting algorithm, IBR bitonic sort, and we prove its correctness in section IV. The hybrid GPU implementation is given in V and its performance is evaluated in section VI. We conclude this article with an outlook in section VII.

II. BASICS

A. Bitonic Sort

Adaptive bitonic sort is based on bitonic sort which is a sorting network algorithm developed by Batcher [2]. A sorting network is a sorting algorithm where the sequence of comparisons (i.e. the order, direction and number of comparisons) is pre-determined and data-independent. Therefore the algorithm can be described by a static network of compare-exchange operations (COEX-operations), connected among each other by data-lanes.

Bitonic sort uses a divide-and-conquer approach similar to merge sort: Sorting a sequence of \(N\) elements consists of \(\log N\) stages. In the first stage pairs of subsequences of length 1 (which are trivially sorted) are merged which results in sorted subsequences of length 2. In the second stage pairs of these sorted subsequences are merged resulting in sorted subsequences of length 4, etc. However, bitonic sort differs from merge sort in the way the subsequences are merged.

In contrast to merge sort in each stage bitonic sequences (cf. Figure 1) are sorted. A bitonic sequence is a concatenation...
A fixed set of COEX-operations. Due to this the number of operations needed to apply one step of a bitonic sequence of length $M$ can be sorted within $\log M$ steps each consisting of \( \frac{M}{2} \) COEX-operations by recursively applying the lemma to the resulting subsequences $L(E)$ and $U(E)$.

Figure 2 shows the corresponding network for sorting a bitonic sequence (i.e. a bitonic merger) of length 8. Note that ring-shifts of $L(E)$ and $U(E)$ can also be used in the recursion. In fact, using any other way to apply a step, i.e. for splitting $E$ in two subsequences, could be used as long as properties (1) and (2) from lemma 1 hold for the two resulting subsequences. Figure 3 shows the bitonic sorting network for arbitrary sequences of length 8.

In the $i$th stage of bitonic sort \( \frac{N}{2^i} \) bitonic sequences of length $2^i$ are sorted within $i$ steps, $i=1,\ldots,\log N$. For every stage $i \geq s$, \( \frac{N}{2^i} \) bitonic sequences of length $2^i$ are processed, whereat each bitonic sequence is turned into two bitonic subsequences (each of length $2^{i-1}$) for which lemma 1 holds. The number of operations needed to apply one step to a single bitonic subsequence of length $M$ is defined to be $C(M)$ in the following. Thus the total number of processed operations for sorting a sequence of length $M$ using bitonic sort based approaches is

\[
\sum_{i=1}^{\log N} \sum_{s=1}^{i} \frac{N}{2^s} \cdot C(2^s)
\]

Original bitonic sort closely follows the construction of $L(E)$ and $U(E)$ from lemma 1, thus applying a step on a bitonic sequence of length $2^s$ consists of $C_{bit}(2^s) = \frac{N}{2^s}$ COEX-Operations. Due to this the number of operations needed to process all \( \frac{N}{2^s} \) bitonic sequences of step $s$ is constantly

\[
\frac{N}{2^s} \cdot C_{bit}(2^s) = \frac{N}{2^s} \cdot 2^s = \frac{N}{2}
\]

Therefore the total number of COEX-Operations is

\[
\sum_{i=1}^{\log N} \sum_{s=1}^{i} \frac{N}{2^s} \cdot C_{bit}(2^s) = \sum_{i=1}^{\log N} i \cdot \frac{N}{2} = O(N \cdot \log^2 N)
\]

Thus the complexity of bitonic sort is $O(N \cdot \log^2 N)$. This is not optimal.
B. Lemma 1 and complexity

As shown constructing \( L(E) \) and \( U(E) \) of a subsequence of length \( M \) using \( C_{bit}(M) = \frac{M}{2} \) COEX-operations results in a total complexity \( O(N \cdot \log^2 N) \). However, if constructing \( L(E) \) and \( U(E) \) would require only \( O(\log M) \) operations, the total complexity would be:

\[
\sum_{i=1}^{\log N} \sum_{s=1}^{2^s} N \cdot C_{ada}(2^s) \\
\leq \log N \cdot \left( \sum_{s=1}^{\log (N)} \frac{N}{2^s} \cdot C_{ada}(2^s) \right) \\
= \log N \cdot \left( \sum_{s=1}^{\log (N)} \frac{N}{2^s} \cdot \log 2^s \right) \\
= N \cdot \log N \cdot \left( \sum_{s=1}^{\log (N)} \left( \frac{1}{2} \right)^s \cdot s \right) \\
= N \cdot \log N \cdot \left( \frac{-1 - \frac{1}{2} \cdot \log N}{(\frac{1}{2})^2} + \frac{1 + \log N}{(\frac{1}{2})^2} + \frac{1}{2} \right) \\
= N \cdot \log N \cdot \left( -2 - \log N \cdot \frac{1}{2} + 2 \right) \\
= N \cdot \log N \cdot \left( -2 - \log N \cdot \frac{1}{N} + 2 \right) \\
= O(N \cdot \log N)
\]

In the following we show how in adaptive bitonic sort \( L(E) \) and \( U(E) \) are constructed using \( C_{ada}(M) = O(\log M) \) operations. Afterwards we introduce our approach (IBR bitonic sort) and prove that \( C_{ibr}(M) \) is also in \( O(\log M) \).

C. Adaptive Bitonic Sort

Figure 4 (a) shows the approach of bitonic sort. Applying the COEX-operations from lemma 1 results in rearrangement of intervals of elements: the elements in solid boxes are not exchanged while elements in dotted/dashed boxes are exchanged. Adaptive bitonic sort is based on this observation (Figure 4 (b)). It efficiently \( O(\log M) \) computes which intervals of elements have to be rearranged. Furthermore it rearranges these intervals with \( O(\log M) \) operations using bitonic trees.

The following lemma is the basis for adaptive bitonic sort. It shows that in order to determine the intervals of elements to be rearranged only a single value \( q \) has to be computed. The lemma expresses \( L(E) \) and \( U(E) \) as two ring-shifts by \( q \) and \( -q \mod M/2 \).

Lemma 2: Let \( E = (E_0, E_1, \ldots, E_{M-1}) \), \( M = 2^k \), \( k \in \mathbb{N}, M \in \mathbb{N} \) be a bitonic sequence. There exists \( q \in \mathbb{Z} \), such that with \( (F_0, F_1, \ldots, F_{M-1}) := S_q(E) \):

- \( L(E) = S_{-q \mod M/2}(F_0, \ldots, F_{M/2-1}) \)
- \( U(E) = S_{-q \mod M/2}(F_{M/2}, \ldots, F_{M-1}) \)

Proof: cf. [3]

As shown in [3] \( q \) can be found with \( \log M \) comparisons using a kind of binary search.

Thus it is possible to compute in time complexity \( O(\log M) \) which intervals of elements have to be rearranged. If one implemented the rearrangement by copying each element, the complexity would be \( O(M) \) again. However in [3] bitonic sequences are represented using bitonic trees which allow the rearrangement of subsequences with complexity \( O(\log M) \). Figure 5 gives an example of a bitonic tree.

Using bitonic trees, the rearrangement of intervals of elements can be processed using \( O(\log M) \) exchanges of values and subtrees (i.e. pointers). Thus the total number of operations \( C_{ada}(M) \) for constructing \( L(E) \) and \( U(E) \) is in \( O(\log M) + O(\log M) = O(\log M) \).

III. IBR BITONIC SORT

In this section our approach, called Interval Based Rearrangement bitonic sort (IBR bitonic sort), is shown. Analog to the previous section we show that \( C_{ibr}(M) \) is
Input: index i
array of keys A
list of intervals
\([0_0, l_0], (0_1, l_1), \ldots, (0_k-1, l_{k-1})\)
\(j = 0\)
while \(i \geq l_j\)
\(i = i - l_j\)
\(j = j + 1\)
end
return \(A[l_j + 1]\)

Listing 1: Pseudocode get(i)

\[
A = \begin{bmatrix}
index: 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
key: 2 & 3 & 5 & 7 & 8 & 10 & 13 & 15 & 17 & 14 & 12 & 11 & 9 & 7 & 3 & 1
\end{bmatrix}
\]

Figure 6. Example IBR bitonic sort

in \(O(\log M)\), i.e. we show that the complexity of applying a step on a single bitonic subsequence (cf. section II-B) is \(O(\log M)\).

We use the same approach for computing the intervals of elements of the input array that form \(L(E)\) and \(U(E)\) as in adaptive bitonic sort. In contrast to adaptive bitonic sort we do not rearrange the intervals physically in memory after each step. Instead we only store the information on which intervals of the input array the subsequences consist of. In the following, an interval of the input array is given by its starting index (offset \(o\)) and its length \(l: [o, l]\).

In Figure 4 (b) the upper resulting bitonic sequence \(2, 3, 3, 1\) can be represented by the intervals \([0, 2], [6, 2]\) of elements of the input sequence. The lower resulting bitonic sequence \(8, 7, 5, 7\) can be represented by \([4, 2], [2, 2]\).

Given the intervals for a subsequence, the \(n\)th element of this sequence can be accessed using the \(get\)-function shown in listing 1. The time complexity of the \(get\)-function scales linearly in the number of intervals.

The representation of \(L(E)\) (\(U(E)\)) using intervals can be used to process the next step directly. Figure 6 shows an example. The Figure depicts the first two steps of the fourth stage of sorting a sequence of length 16. The bitonic input sequence \(E^{1,0}\) of the fourth stage is stored in the array \(A\). This sequence is represented by \([0, 8], [8, 8]\). The output of the first step are two bitonic sequences \((E^{2,0}, E^{2,1})\) each of length 8 represented by \([0, 5], [13, 3]\) and \([8, 5], [5, 3]\). These two sequences are used as input for the second step of the stage, using the above introduced \(get\)-function to access elements. The output of the second stage are four bitonic sequences \((E^{3,0}, E^{3,1}, E^{3,2}, E^{3,3})\) of length 4 each represented by up to three intervals. This example shows that (using this basic approach) the number of intervals per subsequence might increase in each step.

Listing 2 shows pseudocode for sorting a bitonic sequence using interval representation. In each iteration of the loops

Input: bitonic sequence of length \(N\)
in array \(A\)

\[
// Initial interval \([0, \frac{N}{2}], \left[\frac{N}{2}, \frac{N}{2}\right]\)
for \(i = \log(N)\) to 1
forall sequences of step \(i\)
// Determine \(q\) using \(get\)-function
// Calculate interval representation
end
end

// Transform array \(A\) into an output array \(B\)
// according to its interval representation
for \(j = 1\) to \(N\)
\(B[j] = get(1)\)
end

Listing 3: IBR bitonic sort
q is determined. As said above, q can be computed using \(O(\log M)\) accesses to the corresponding subsequence. The accesses to the elements of the subsequence are processed using the \(\text{get}\)-function, thus total time complexity for computing \(q\) is \(O(\log M)\) times the complexity of \(\text{get}\). The time complexity of the \(\text{get}\)-function scales linearly in the number of intervals by which a sequence is represented. In the following sections we show that using a slightly modified approach the number of intervals is bounded by 2 and thus the time complexity of the \(\text{get}\)-function is \(O(1)\). In that case it holds \(C_{ibr}(M) = \log M\).

**IV. Proof**

As one can see in listing 3 at the beginning of each phase, each bitonic sequence is represented by two intervals. Furthermore for each of these bitonic sequences it holds, that one of the intervals is ascending and the other interval is descending, i.e. that either the first interval is ascending and the second interval is descending or the first interval is descending and the second interval is ascending. In the following lemma we prove that such bitonic sequences can be transformed into two subsequences that hold (1) and (2) from lemma 1. Furthermore we prove that each of those subsequences can be also represented using at most two intervals whereby again one of the intervals is ascending and the other one is descending. This is achieved by doing this transformation in a way that the two resulting subsequences are ring-shifts of \(L(E)\) and \(U(E)\) from lemma 1. Thus by computing the transformations as given in lemma 3, at each step of IBR bitonic sort each bitonic sequence is represented by at most two intervals. Because the only difference between bitonic sort and approaches like adaptive bitonic sort or IBR bitonic sort is the way a bitonic sequence is transformed into two subsequences that hold (1) and (2) from lemma 1, this also proves the correctness of IBR bitonic sort.

**Lemma 3:** Let \(E\) be a bitonic sequence of length \(2^k = M\), \(k \in \mathbb{N}\), represented by two intervals \([o_0, l_0],[o_1, l_1]\) whereby one of the intervals is ascending and the other interval is descending.

Then one can find in time complexity \(O(\log M)\) a representation of two subsequences \(U_E\) and \(L_E\) by two intervals each

\[
L_E \simeq [o_{0,0}, l_{0,0}];[o_{0,1}, l_{0,1}]
\]
\[
U_E \simeq [o_{1,0}, l_{1,0}];[o_{1,1}, l_{1,1}]
\]

such that \(g, h \in \mathbb{N}\) exist for which it holds that \(L(E)\) is \(L_E\) ring-shifted by \(g\) and \(U(E)\) is \(U_E\) ring-shifted by \(h\)

\[
L(E) = S_g(L_E)
\]
\[
U(E) = S_h(U_E)
\]

Furthermore for both \(L_E\) and \(U_E\) it holds again that one of the intervals is ascending and the other interval is descending.
\[
s = 0
\]
\[
e = l_0
\]
while \((s < e)\) {
    \[
    \text{mid} = s + \text{floor}\left(\frac{e-s}{2}\right)
    \]
    if \(\text{get}(A, \text{mid}) < \text{get}(A, \text{mid} + M/2)\) {
        \[
        s = \text{mid} + 1
        \]
    } else {
        \[
        e = \text{mid}
        \]
    }
}\[
q = s
\]

Listing 4: Find \(q\) in the third case

**Proof:** We first note that because \(E\) is represented by at most two intervals and these intervals are ascending or descending, \(E\) is a trivial bitonic sequence which consists of an ascending and a descending subsequence. Thus there are six different cases for \(E\) that we consider, which are all shown in Figure 7.

The first two are special cases where the whole sequence is sorted and represented by one interval of length \(M\) \([0,0]\) and one interval of length 0. In both cases the computation of \(L_E\) and \(U_E\) is trivial and \(L_E\) and \(U_E\) can be represented by \([0, \frac{M}{2}]\) and \([\frac{M}{2}, M]\) and \(y = 0, h = 0\).

In any other case, \(E\) consists either of an ascending interval and a descending interval (cases 3 and 4) or a descending interval and an ascending interval (cases 5 and 6). The cases 3 and 4 differ in the length of the first interval. In case 3 it is \(l_0 \leq \frac{M}{2}\). In case 4 it is \(l_0 > \frac{M}{2}\). The same holds for cases 5 and 6.

We now consider the third case in which \([0, l_0]\) is ascending, \([l_1, l]\) descending and \(l_0 \leq \frac{M}{2}\). This case is depicted in Figure 8. In order to construct \(L_E\) and \(U_E\) we determine \(q\) for which holds

\[
\min (E_i, E_{i+M/2}) = E_i \quad \forall \ i \in \{0,\ldots,q-1\}
\]

\[
\min (E_i, E_{i+M/2}) = E_{i+M/2} \quad \forall \ i \in \{q,\ldots,\frac{M}{2}-1\}
\]

The algorithm in listing 4 determines such \(q\) using a kind of binary search in time complexity \(O(\log M)\). In order to prove this we first show that if \(E_j < E_{i+M/2}\) holds, then also \(E_j < E_{j+M/2}\) holds for \(j < i\) and that, vice versa, if \(E_i \geq E_{i+M/2}\) holds, then also \(E_j \geq E_{j+M/2}\) holds for \(j > i\).

Consider \(i \in \{0,\ldots,l_0-1\}\) for which holds \(E_i < E_{i+M/2}\). It is obvious that the \(i\)th element is within the first interval. Since it holds that \(l_0 \leq \frac{M}{2}\) (third case) it follows, that the \((i + \frac{M}{2})\)th element is within the second interval. For \([0, l_0]\) is ascending it follows that

\[
E_j \leq E_i < E_{i+M/2} \quad \forall \ j \in \{0,\ldots,i\}.\]

Since \([0, l_1]\) is descending it follows that

\[
E_{i+M/2} \leq E_{j+M/2} \quad \forall \ j \in \{0,\ldots,i-1\}.
\]

Combining this equations implies

\[
E_j \leq E_i < E_{i+M/2} \leq E_{j+M/2} \quad \forall \ j \in \{0,\ldots,i-1\}
\]

and thus

\[
E_j < E_{j+M/2} \quad \forall \ j \in \{0,\ldots,i-1\}.
\]

In listing 4 \(s\) is always assigned to \(q\) in the end. Furthermore \(s\) holds the above requirements for \(i\) (since \(s\) is changed to \(\text{mid} + 1\) if and only if \(E_{\text{mid}} < E_{\text{mid}+M/2}\), so it is guaranteed that

\[
E_j < E_{j+M/2} \quad \forall \ j \in \{0,\ldots,q-1\}.
\]

Thus the first of the above properties for \(q\) is proved.

Let \(i \in \{0,\ldots,l_0-1\}\) for which holds \(E_i \geq E_{i+M/2}\). For \([0, l_0]\) is ascending it follows that

\[
E_j \geq E_i \geq E_{i+M/2} \quad \forall \ j \in \{i+1,\ldots,l_0-1\}.
\]

Since \([0, l_0]\) is descending it follows that

\[
E_{i+M/2} \geq E_{j+M/2} \quad \forall \ j \in \{i+1,\ldots,l_0-1\}.
\]

Combining this equation implies

\[
E_j \geq E_i \geq E_{i+M/2} \geq E_{j+M/2}
\]

and thus

\[
E_j \geq E_{j+M/2} \quad \forall \ j \in \{i+1,\ldots,l_0-1\}.
\]

The loop in listing 4 terminates if and only if \(s \geq e\). Initially it is \(e = l_0\) and \(e\) is changed to \(\text{mid}\) if and only if \(E_{\text{mid}} \geq E_{\text{mid}+M/2}\), so again it is guaranteed that

\[
E_j \geq E_{j+M/2} \quad \forall \ j \in \{q,\ldots,l_0-1\}.
\]

Let \(i \in \{l_0,\ldots,\frac{M}{2}-1\}\). For \([0, l_1]\) is descending it holds that

\[
E_i \geq E_{i+M/2} \quad \forall \ i \in \{l_0,\ldots,\frac{M}{2}-1\}.
\]

Thus the second of the above properties for \(q\) is proved.

We now show how to construct the two resulting subsequences \(L_E\) and \(U_E\) using at most two intervals. Because \(q\) holds both properties it is

\[
L(E) = E_0; E_1; \ldots; E_{q-1}; E_{q+M/2}; E_{q+1+M/2}; \ldots; E_{M-1}
\]

\[
U(E) = E_{0+M/2}; E_{1+M/2}; \ldots; E_{q-1+M/2}; E_{q}; E_{q+1}; \ldots; E_{M/2-1}.
\]

Thus a representation of \(L(E)\) and \(U(E)\) by intervals is

\[
L(E) \simeq [0, q], [0 + (l_1 - \frac{M}{2}) + q, \frac{M}{2} - q]
\]

\[
U(E) \simeq [0 + (l_1 - \frac{M}{2}), q, \frac{M}{2} - q, l_0 - q], [0, l_1 - \frac{M}{2}].
\]

This representation of \(L(E)\) by intervals consists of two intervals whereby the first \(([0, q])\) is ascending and the
second \((o_1 + (l_1 - M/2) + q, M/2 - q)\) descending. Thus \(L(E)\) with this representation can be used as \(L_E\). However, the representation of \(U(E)\) consists of three intervals. This problem can be solved by a ring-shift of \(U(E)\) by \(l_1 - M/2\). Set \(U_E = S_{l_1-M/2}(U(E))\). It follows

\[
U_E = E_{M/2-(l_1-M/2)}; E_{M/2-(l_1-M/2)+1}; \ldots; E_{M/2-1}; E_{0+M/2}; \ldots; E_{q-1+M/2}; E_q; \ldots; E_{M/2-(l_2-M/2); E_{l_0-1}}.
\]

Then it is

\[
U_E \simeq [o_1, q + l_1 - M/2], [o_0 + q, l_0 - q].
\]

This is a valid representation, since \(U_E\) is represented by two intervals and the first interval \([o_1, q + l_1 - M/2]\) is descending and the second \([o_0 + q, l_0 - q]\) is ascending. As mentioned, the total time complexity to determine \(q\) is \(O(\log M)\). Since the total time complexity for building the representations afterwards is \(O(1)\), the total time complexity is \(O(\log M)\). Due to this, \(C_M(M)\) is also in \(O(\log M)\), thus IBR bitonic sort has optimal complexity.

The proof for the other cases is similar to the proof for the third case.

V. IMPLEMENTATION FOR GPUs

In this section we present a novel hybrid sorting algorithm for GPUs using IBR bitonic sort as its basis. Meanwhile there are many high-performing sorting algorithms for GPUs. The fastest algorithms for 32 bit integer sequences are radix sort from Merrill et al. [10] and Satish et al. [13]. However since IBR bitonic sort is comparison-based we decided to combine IBR bitonic sort with an existing comparison-based algorithm for GPUs.

Many different approaches for comparison-based algorithms have been developed: GPU quick sort from Cederman et al. [5], merge sort from Satish et al. [13], [14], warp sort from Ye et al. [15] or bitonic sort from Buck et al. [4], Purcell et al. [12], Kipfer et al. [8] or Govindaraju [6]. Greß and Zachmann implemented a hybrid algorithm using adaptive bitonic sort, odd even sort and bitonic sort [7]. In contrast to our approach they used bitonic trees.

The fastest comparison-based sorting algorithms are bitonic sort from Peters et al. [11] and sample sort from Leischnier et al. [9]. However, for smaller sequences bitonic sort from Peters et al. is significantly faster than sample sort from Leischnier et al. Furthermore two bitonic sort based approaches can be combined straightforward. Particularly it is possible to switch between both algorithms within a stage. Thus we decided to combine IBR bitonic sort with bitonic sort from Peters et al.

Bitonic sort from Peters et al. performs best when processing small sequences but suffers from its \(O(N \cdot \log^2 N)\) complexity when sorting larger sequences. Pseudocode of the algorithm is shown in listing 5. The first \(firstStages\) stages are executed using one single kernel \(BS_firstStages\) and using shared memory. In the following stages up to four consecutive steps are executed using \(BS_steps\) until step \(lastSteps\). Afterwards the last \(lastSteps\) steps are executed using one kernel \(BS_lastSteps\) and shared memory. The variables \(firstStages\) and \(lastSteps\) depend on several parameters, e.g. the specification of the used GPU, the key size and whether or not key-value pairs are to be sorted. A more detailed description can be found in [11].

Implementing IBR bitonic sort for GPUs is straightforward. Listing 6 shows pseudocode for the resulting hybrid algorithm. It closely follows the implementation of bitonic sort in listing 5, but replaces the inner loop, i.e. it processes \(stage - lastSteps\) steps in a single kernel launch using the interval representation of the sequence.

This implies the creation of an interval representation of the sequence using \(BS_2_IBR\) and the rearrangement of the sequence according to the interval representation using
IBR\_2\_BS in each stage. The interval representation at the beginning of a stage $s$ only depends on the length of the sequence and the value of $s$ (cf. listing 3), so that BS\_2\_IBR does not induce memory transactions or any computations depending on the input sequence.

IBR\_2\_BS is explicitly used in the pseudocode to rearrange the sequence after applying IBR\_stages, but in our implementation this is implicitly done by using a modified version of BS\_lastSteps, in which elements are accessed using the get-function from listing 1. Thus the rearrangement induces only a small computational overhead and no additional memory transactions compared to bitonic sort. Nevertheless, in contrast to bitonic sort from Peters et al. the memory access pattern might not be optimal, so that some memory transactions might not be coalesced.

We used a basic approach for parallelization of IBR bitonic sort in our hybrid implementation: We do not parallelize the computation of $L(E)$ and $U(E)$ (i.e. determining $q$) of a single bitonic sequence, but we process bitonic sequences in parallel. When sorting a sequence of length $2^k$, the total number $T$ of bitonic subsequences only depends on the currently processed step, not the stage (cf. Figure 3: there are two bitonic subsequences in step 2 of stage 2 as well as in step 2 of stage 3). As mentioned in section II-A, in step $s$ the total number of bitonic subsequences (all of length $2^s$) is $T(s) = \frac{2^s}{s}$. Thus in most of the steps during the whole sorting process there are enough bitonic subsequences to achieve a fair parallelization using this basic approach. Only if $s$ is large, i.e. it is close to $k$, the number of bitonic subsequences might be too small. But the number of operations to compute $L(E)$ and $U(E)$, $C_{ada}(2^s)$, equals $\log 2^s = s$. Due to this the total number of operations in step $s$, $C_{ada}(2^s)\cdot T(s)$, is lesser in larger steps than in smaller ones:

$$C_{ada}(2^{(s-1)})\cdot T(s-1)$$

$$= C_{ada}(2^{(s-1)}) \cdot \frac{2^k}{2(s-1)} = (s-1) \cdot \frac{2^k}{2s}$$

$$= (s-1) \cdot 2 \cdot \frac{2^k}{2s} = 2 \cdot (C_{ada}(2^s) - 1) \cdot T(s)$$

That means, that in steps which cannot be parallelized very well, the total number of operations is small, while a good parallelization is achieved in steps with a high total number of operations. Therefore even with the basic approach of parallelization good performance results can be achieved.

VI. Evaluation

We tested our hybrid implementation using a GTX280, a GTX480 and a GTX580 GPU, using 32 bit integers as well as 64 bit integers. The sequences of integers were randomly generated using a uniform distribution. We tested sequences with lengths ranging from $2^{15}$ elements to the maximum length so that the tested algorithms still could sort the sequences in the GPU memory.

Since sorting on GPUs is usually considered to be just a part of a larger computation, we did not include the time needed to transfer data to or from the GPU in our tests. We compared our algorithm to the best performing comparison-based algorithms: bitonic sort from Peters et al. [11], sample sort from Leischner et al. [9], the merge sort from Satish et al. from the thrust library [1], [13], GPU quick sort from Cederman et al. [5] and warp sort from Ye et al. [15].

Satish et al. presented in [14] a novel sorting algorithm which is based on merge sort. Since the source code is not available we could not include this algorithm in our tests. According to [14] the performance obtained with their merge sort seems to be similar to the performance of bitonic sort from Peters et al. [11].

In all Figures the x-axis shows the length of the input sequence and the y-axis presents the sorting rate, i.e. the runtime of the sorting algorithm divided by the length of the input sequence.

Figure 9 presents the results of the test for 32-bit integers. The bitonic sort from Peters et al. and our hybrid algorithm perform best for all GPUs and all input sizes. As expected our hybrid algorithm performs similarly to the bitonic sort from Peters et al. for smaller sequences. As mentioned, the main disadvantage of bitonic sort is its complexity $O(N \cdot \log^2 N)$. Thus the length of the input sequences has much more influence on the performance of bitonic sort than on the performance of IBR bitonic sort. In the case of a GTX580 and sorting a sequence of length $2^{26}$, the sorting rate of our hybrid algorithm is close to 500 mio/s whereas the sorting rate of the bitonic sort from Peters et al. is about 325 mio/s. Thus the speedup is greater than 1.5.

Figure 10 shows the results for 64bit integers. Again our hybrid algorithm performs best for all GPUs and all input sizes. In contrast to 32bit integers the bitonic sort from Peters et al. is outperformed by the sample sort from Leischner et al. for the GTX480 and GTX580 GPU when sorting sequences which are larger than $2^{23}$. For the largest sequences in our tests the performance of sample sort comes very close to the performance of IBR bitonic sort.

The speedup of our hybrid algorithm over the bitonic sort from Peters et al. is similar to the speedup in the case of 32 bit integers.

VII. Conclusion

In this paper we presented IBR bitonic sort, a comparison-based sorting algorithm for parallel architectures. Our algorithm uses an approach which is similar to adaptive bitonic sort. We proved that IBR bitonic sort has also optimal complexity $O(N \cdot \log N)$, but in contrast to adaptive bitonic sort it does not need any special datastructure for storing sequences. We implemented a hybrid algorithm based on bitonic sort and IBR bitonic sort for NVIDIA’s GPUs. In our tests IBR bitonic sort turns out to be the fastest comparison-based algorithm found in literature.
Figure 9. 32 bit-integer
Figure 10. 64 bit-integer
REFERENCES


