Fast in-place sorting with CUDA based on bitonic sort

Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger

Research Group for Communication Systems
Department of Computer Science
Christian-Albrechts-University Kiel, Germany

Abstract. State of the art graphics processors provide high processing power and furthermore, the high programmability of GPUs offered by frameworks like CUDA increases their usability as high-performance coprocessors for general-purpose computing. Sorting is well-investigated in Computer Science in general, but (because of this new field of application for GPUs) there is a demand for high-performance parallel sorting algorithms that fit to the characteristics of modern GPU-architecture. We present a high-performance in-place implementation of Batcher's bitonic sorting networks for CUDA-enabled GPUs. We adapted bitonic sort for arbitrary input length and assigned compare/exchange-operations to threads in a way that decreases low-performance global-memory access and thereby greatly increases the performance of the implementation.

Keywords: GPU, GPGPU, CUDA, Parallel, Sorting, Multicore

1 Introduction

The processing power of modern desktop-GPUs has greatly increased during the last years. In fact, it exceeds the processing power (measured in FLOP/s) of desktop-CPUs by far. Since the release of frameworks like NVIDIA's CUDA, that provide free programmability of GPUs, the processing power of GPUs is easily available for General Purpose Computing on GPU's (GPGPU).

Many general purpose applications require high-performance sorting algorithms, therefore many sorting algorithms have been explicitly developed for GPUs. Early implementations ([1], [2], [3], [4], [5]) often based on bitonic sort [6]. Although bitonic sort is an in-place sorting algorithm, early implementations (due to the limitations of early programming frameworks and the underlying hardware) were not in-place. Meanwhile many sorting algorithms with a wide range in design were developed for GPUs, many of them using CUDA. Harris et al. [7], Grand [8] and He et al. [9] implemented radix sort for GPUs. Sengupta et al. [10] were the first to implement quicksort, later Cederman et al. [11] developed a more efficient implementation of quicksort. Sintorn et al. [12] presented a hybrid algorithm that combines merge sort and bucket sort. Recently, Satish et al. [13] developed new implementations of radix sort and merge sort for CUDA. None of these sorting algorithms is both in-place and comparison-based.

In this contribution we present an implementation of Batcher's bitonic sort for NVIDIA's CUDA. In opposite to other implementations of bitonic sort this implementation is highly competitive to other sorting algorithms on GPUs. Furthermore, it keeps the benefits of using sorting networks, i.e. it works comparison-based and in place. We give a short introduction to bitonic sort in section 2 and present a trivial implementation of bitonic sort for CUDA in section 3.1. Our optimization of bitonic sort for NVIDIA's GPUs by reducing access to global memory is shown in section 3.2. Section 3.3 introduces a modification of bitonic sort for sequences of arbitrary length. In section 4 we show performance measurements and comparisons to other GPU sorting algorithms. We summarize the results of this contribution in section 5.

2 Introduction to bitonic sort

Bitonic sort is based on bitonic sequences, i.e. concatenations of two subsequences sorted in opposite directions.

Given a bitonic sequence B with length $m = 2^r$, B can be sorted ascending (analog descending) in r steps. In the first step the pairs of elements $(B[0], B[\frac{m}{2}])$, $(B[1], B[\frac{m}{2}+1])$,...,$(B[\frac{m}{2}-1], B[m-1])$ are compared and exchanged if the first element is smaller than the second element. This results (as shown by Batcher[6]) in two bitonic subsequences, $(B[0], \ldots, B[\frac{m}{2}-1])$ and $(B[\frac{m}{2}], \ldots, B[m-1])$, whereas all elements in the first subsequence are smaller than any element in the second subsequence. In the second step the same procedure is applied to both subsequences, resulting in four bitonic subsequences. All elements in the first subsequence are smaller than any element in the second, all elements in the second subsequence are smaller than any element in the third and all elements in the third subsequence are smaller than any element in the fourth subsequence. The third, fourth,...,r-th step are analog. Processing the r-th step results in 2^r subsequences of length 1, thus the sequence B is sorted.

Let A be the input array to sort and let $n = 2^k$ be the length of A. The process of sorting A consists of k phases. The subsequences of length 2 $(A[0], A[1])$, $(A[2], A[3])$,..., $(A[n-2], A[n-1])$ are bitonic sequences by definition. In the first phase these subsequences are sorted (as shown above) alternating descending and ascending[1], what makes the subsequences of length 4, $(A[0], A[1], A[2], A[3])$,...,$(A[n-4], A[n-3], A[n-2], A[n-1])$, bitonic sequences. In the second phase these subsequences of length 4 are sorted alternating descending and ascending, resulting in subsequences of length 8 being bitonic sequences. In the i-th phase of bitonic sort the total number of subsequences being sorted is 2^{k-i} and the length of each of these subsequences is 2^i, thus the i-th phase consists of i steps. After the $(k-1)$-th phase the array A is a bitonic sequence. A is sorted in the last phase k.

In every step $\frac{n}{2}$ compare/exchange operations are processed, thus the number of compare/exchange operations in bitonic sort, and by this reason the time complexity, is $O(n \cdot \log^2 n)$. This complexity makes bitonic sort less advantageous for

[1] Or ascending/descending, the resulting subsequences are bitonic in both cases.

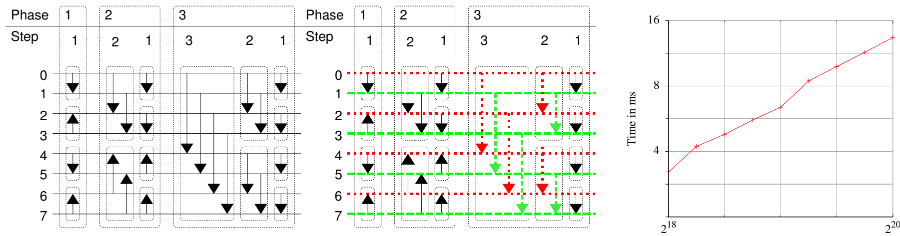


Fig. 1. Sorting network **Fig. 2.** Partition example **Fig. 3.** Absolute runtime

sequential use. The strength of bitonic sort is that it can be easily implemented by comparator networks. Comparator networks can be implemented directly in hardware, but they are also adequate for generic parallel architectures since they work in-place, require less inter process communication and furthermore, can be naturally implemented in SIMD architectures.

Figure 1 shows the bitonic sorting network for an arbitrary input sequence of size $n = 8$. The sorting network consits of $3 = \log 8$ phases, phase p having p steps. Every step consists of $4 = \frac{n}{2}$ compare/exchange operations.

3 Implementation

3.1 Basic implementation of bitonic sort with CUDA

CUDA is a parallel programming model and software environment for NVIDIA's GPU. CUDA allows the programmer to define functions, so called kernels, which are executed on the GPU. One kernel is executed in parallel by a number of CUDA threads. Threads are organized in blocks. The number of blocks and the number of threads per block (and therefore the number of threads) can be specified for each individual launch of this kernel. On up-to-date GPUs the maximum number of blocks is 2^{32} and the maximum number of threads per block is 512.

A thread block has a shared memory visible to all threads of the block. Every thread has its own set of registers and all threads have random access to the same global memory. In general, access to global memory is significantly slower than access to shared memory.

CUDA provides a barrier-mechanism for synchronization of threads of the same block, but does not provide any synchronisation mechanisms for threads of different blocks.

In order to achieve a good utilisation of the GPU one have to take care about the number of launched threads. On up-to-date GPUs the number of threads per block should be at least 64, the number of blocks should be at least 100.

For a trivial implementation of bitonic sort with CUDA one could use an approach that step-wise follows the implementation of bitonic sort by sorting

networks: If the array to sort of length $n = 2^k$ already resides in GPU memory, bitonic sort could be implemented by a sequence of kernel launches, each launch processing all compare/exchange operations of one step of one phase. Every kernel launch would consist of 2^{k-1} threads, each thread processing one compare/exchange operation.

This implementation is optimal in the sense that it provides a good utilisation of the GPU for larger input size and results in a uniform distribution of memory access and workload to threads. The main disadvantage is the excessive use of global memory what significantly increases the runtime of the algorithm. Since all operations of one step are processed in an individual kernel launch, each compare/exchange operation causes at least 2 reading or writing accesses to global memory.

Sorting a sequence of length 2^k requires k phases. The j-th phase consists of j steps. In the trivial solution each step causes $2 \cdot 2^{k-1} = 2^k$ read accesses to global memory. Thus the total number of read accesses in this case is:

$$\sum_{j=1}^{k} j \cdot 2^k = \frac{k \cdot (1+k)}{2} \cdot 2^k = \frac{(\log n) \cdot (1 + \log n)}{2} \cdot n \tag{1}$$

The number of write accesses is at most the same.

3.2 Minimizing access to global memory

Our approach to significantly reduce runtime of the algorithm is to reduce the number of accesses to global memory and the number of kernel launches. Therefore we carefully partition the set of operations into subsets that can be processed by single threads or thread blocks, such that operations of multiple consecutive steps can be processed by one thread in one kernel launch and each element has to be read from or written to global memory at most once.

Figure 2 shows an example of such a partition for phase 3. The partition consists of two subsets, the one marked by red dotted lines and arrows, the other marked by green dashed lines and arrows. All operations of step 3 **and** 2 can be processed in one kernel launch by two threads, because operations in one subset are not affected by operation in the other subset. Note that a thread that processes one of these partitions can keep elements in registers, thus one element is read and written only once (and not twice like in the trivial implementation).

In the following we give a formal definition of the partitions used in our algorithm. Consider an array A of size 2^k to be the input array to sort. The set of indices in A is $I = \{0, \ldots, 2^k - 1\}$. We define $\mathrm{COEX}_{p,s}$ to be the set of compare/exchange operations that belong to step s of phase p of bitonic sort. The commutative representation of one compare/exchange operation in $\mathrm{COEX}_{p,s}$ is $\mathrm{coex}_{p,s}(a, b) = \mathrm{coex}_{p,s}(b, a)$. The whole set of compare/exchange operations processed by bitonic sort for sorting A is:

$$COEX = \bigcup_{p=1}^{k} \left(\bigcup_{s=1}^{p} COEX_{p,s} \right)$$

We define an operator K that maps sets of compare/exchange operations to the sets of array positions, that are affected by these operations. Furthermore, we define a class of operators $N_{p,s}$ that map a set of array positions J to the set of compare/exchange operations that affect array positions in J in step s of phase p:

$$
\begin{aligned}
&K : \mathcal{P}(COEX) \rightarrow \mathcal{P}(I);\ M \mapsto \{i \mid \text{coex}_{\bullet,\bullet}(A[i], \bullet) \in M\} \\
&N_{p,s} : \mathcal{P}(I) \rightarrow \mathcal{P}(COEX_{p,s});\ J \mapsto \{\text{coex}_{p,s}(A[i], \bullet) \in \text{COEX}_{p,s} \mid i \in J\}
\end{aligned}
$$

In a single step s of a phase p any element of A is involved in exactly one compare/exchange operation ($\forall i \in I : |N_{p,s}(\{i\})| = 1$). Thus there are no dependencies between operations in the same step of the same phase, all operations of $\text{COEX}_{p,s}$ can be done (asynchronously) in parallel, as it is done in the trivial implementation.

To ensure correctness bitonic sort requires that before an operation $c \in \text{COEX}_{p,s}$ is processed, both corresponding operations in the preceding step, more precisely, the operations in $N_{p,s+1}(K(\{c\}))$ or $N_{p-1,1}(K(\{c\}))$ (for $p = s$) must be already processed — of course the same requirement holds for elements in that sets, i.e. the four corresponding operations in step $s + 2$ must be already processed and so on. Again have a look at figure 2. In step 2 of phase 3 one red dotted operation corresponds to both red dotted operations in step 3.

A general definition of a partition[2] of I, that ensures a uniform distribution of memory access and workload to threads, is

$$\forall J \in P : |J| = \frac{n}{m}, |N_{p,s}(J)| = \frac{1}{2} \cdot |J| \tag{2}$$

In fact, if P holds (2) then P is a partition of equal sized sets of indices that induce sets (of the half size) of compare/exchange operations, thus

$$\{N_{p,s}(J) \mid J \in P\} \text{ is a partition of } \text{COEX}_{p,s} \tag{3}$$

In simple words, all operations in $\text{COEX}_{p,s}$ that affect elements in $J \in P$ do not affect elements that are not in J. Since the size of induced sets of operations is half the size of sets of indices all operations in $\text{COEX}_{p,s}$ are processed.

We define the degree of P in step s of phase p as the maximum number of successive steps that hold (2) and (3). More precise, the degree $D_{p,s}(P)$ is the maximum number z for which holds:

$$\forall a \in \{1, ..., z\} : \{N_{p,s-a+1}(J) \mid J \in P\} \text{ is a partition of } \text{COEX}_{p,s-a+1} \tag{4}$$

In the example in figure 2 the used partition in step 3 of phase 3 is $\{\{0, 2, 4, 6\}, \{1, 3, 5, 7\}\}$. This partition induces a partition of operations in step 3 of phase 3 **and** in step 2 of phase 3 (red dotted and green dashed arrows) **but not** in step 1 — thus the degree of this partition is two.

[2] $P = \{I_1, \ldots, I_m\}$ is partition of $I \iff \bigcup_{i=1}^{m} I_i = I, \forall i, j \in \{1, \ldots, m\}, i \neq j : I_i \cap I_j = \emptyset$

The trivial implementation of bitonic sort uses partitions of degree 1. Basically, if we could find a partition of degree $d \geq 2$ in step s of phase p, we could process compare/exchange operations of d steps in a single kernel launch. More precise, using a partition P that holds (2), all threads have to access $n/|P|$ elements in memory and can then process compare/exchange operations of $D_{p,s}(P)$ steps nonstop. All operations of $D_{p,s}(P)$ consecutive steps can be done a in single kernel launch without increasing the number of memory accesses per kernel launch. Therefore the overall reading memory access is divided by the degree of P compared to that in (1). This significantly reduces the runtime of the algorithm.

Actually, in step s of phase p there exists a partition with degree d for every $d \leq s$. This partition can be defined as follows:

$$P^d_{p,s} := \bigcup_{i \in I} K(N_{p,s}(K(N_{p,s-1}(\ldots K(N_{p,s-(d-1)}(\{i\}))\ldots)))) \qquad (5)$$

For example, $P^1_{p,s} = \bigcup_{i \in I} K(N_{p,s-(1-1)}(\{i\}))$ is the partition of the trivial implementation and $P^2_{3,3}$ is the partition used in the example in figure 2.

In our implementation of bitonic sort we use partitions of at most degree 4, since the number of registers (or the size of shared memory, respectively) is bounded. Furthermore, an increasing degree results in an increasing workload per thread which decreases the number of threads and blocks. Thus a higher degree would result in under-utilisation of the GPU.

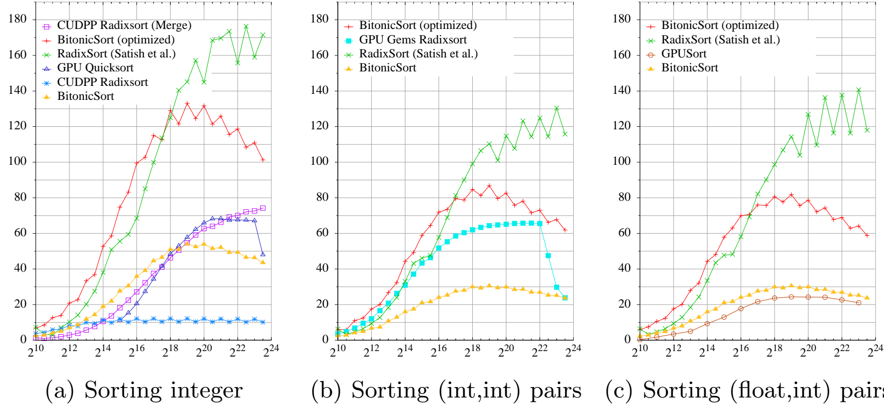
3.3 Bitonic sort for arrays of arbitrary size: Virtual Padding

Bitonic sort as introduced by Batcher can only sort sequences with length being a power of 2. A trivial way to sort sequences of arbitrary length ascending (analog for descending) is to pad the sequence with max-values. Since Batcher's bitonic sort sorts subsequences in alternating directions these elements would be moved while sorting the sequence and therefore have to exist physically. Thus an input sequence of length $2^k + 1$ would result in sorting a sequence of length 2^{k+1}.

Every phase of bitonic sort sorts bitonic sequences — as mentioned in section 2 it makes no difference if the subsequences are sorted ascending/descending or vice versa. Thus bitonic sort can be modified in a way that in each phase the sorting direction in every subsequence containing normal values and max-values (actually there is only one such subsequence in each phase) is ascending. In this modified variant of bitonic sort max-values are never moved and thus do not have to exist physically. Using this modification no additional memory is needed. Figure 3 shows absolute runtimes of our bitonic sort. Of course bitonic sort performs best for input size being a power of 2. Nevertheless there is only a small negative impact when sorting sequences of length being not a power of 2.

4 Evaluation

We compared our algorithm to the bitonic sort based GPUSort [4], the radix sort introduced in the particle demo in the CUDA SDK [8], the quick sort implementation from Cederman et al. [11], both radix sort algorithms in the CUDPP



(a) Sorting integer (b) Sorting (int,int) pairs (c) Sorting (float,int) pairs

Fig. 4. Sorting rate (y-axis) for sorting arrays of size n (x-axis)

library [7], the recently released radix sort algorithm from Satish et al. [13] and to our trivial implementation of bitonic sort. All tests were done on an Intel Core2 Duo 1.86 GHz machine with NVIDIA's GTX280 GPU. We performed 3 different tests: sorting sequences of (a) integers, (b) (int, int) key-value pairs and (c) (int, float) key-value pairs (figure 4). The sequences were randomly generated with length ranging from 2^{10} to 2^{24} elements. Not all of the algorithms were able to sort negative floats and integers, so we used non-negative floats and integers.

We consider sorting to be just one part of larger computation on the GPU, thus we measured the GPU runtime only, not including time for memory transfer between CPU and GPU. In figure 4 we show the sorting rate, i.e. the number of elements divided by the runtime. The sorting rate of our bitonic sort is low for small sequences due to constant start delay and under utilization of the GPU. But it rises with growing number of elements reaching its peak performance for sequences of length 2^{19}. The sorting rate declines for larger sequences due to the the $O(n \cdot \log^2 n)$ complexity of bitonic sort. While the trivial implementation of bitonic sort performs approximately like GPUSort, the results show the much better performance of the optimized implementation. Our optimized bitonic sort is faster than all other sorting algorithms, except for radix sort from Satish et al [13], that is the fastest algorithm for sequences larger than 2^{16}, but neither comparison-based nor in-place. Particularly, bitonic sort is faster than the comparison based GPUSort and GPU Quicksort. There is no sourcecode available for merge sort from Satish et al. [13], but comparing the sorting rate given in [13] to the sorting rate of bitonic sort, bitonic sort seems to perform slightly better.

5 Conclusion

We have presented an efficient implementation of bitonic sort for NVIDIA's GPUs. To achieve this we carefully optimized our implementation in respect to

the number of accesses to global memory. It is the first efficient comparison-based in-place implementation of a sorting algorithm for CUDA we found in literature. Although bitonic sort suffers from complexity $O(n \cdot \log^2 n)$, the results of our tests show that it is highly competitive to all other comparison-based GPU sorting algorithms, and also to most non-comparison-based algorithms. Particularly the performance of our bitonic sort seems to be better than the out-of-place merge sort implemented by Satish et al., which is the fastest comparison-based sorting algorithm for GPUs reported in literature so far.

References

1. Kapasi, U.J., Dally, W.J., Rixner, S., Mattson, P.R., Owens, J.D., Khailany, B.: Efficient conditional operations for data-parallel architectures. In: MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, New York, NY, USA, ACM (2000) 159–170
2. Purcell, T.J., Donner, C., Cammarano, M., Jensen, H.W., Hanrahan, P.: Photon mapping on programmable graphics hardware. In: HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association (2003) 41–50
3. Kipfer, P., Segal, M., Westermann, R.: Uberflow: a gpu-based particle engine. In: HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, New York, NY, USA, ACM (2004) 115–122
4. Govindaraju, N., Raghuvanshi, N., Henson, M., Manocha, D.: A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina-Chapel Hill (2005) (2005)
5. Greb, A., Zachmann, G.: Gpu-abisort: optimal parallel sorting on stream architectures. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. (2006)
6. Batcher, K.: Sorting networks and their applications. In: AFIPS Spring Joint Comput. Conf. (1967)
7. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with cuda. In: GPU Gems 3. Addison-Wesley (2007)
8. Grand, S.L.: Broad-phase collision detection with cuda. In: GPU Gems 3. Addison-Wesley (2007)
9. He, B., Govindaraju, N.K., Luo, Q., Smith, B.: Efficient gather and scatter operations on graphics processors. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM (2007) 1–12
10. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association (2007) 97–106
11. Cederman, D., Tsigas, P.: A practical quicksort algorithm for graphics processors. In: ESA '08: Proceedings of the 16th annual European symposium on Algorithms, Berlin, Heidelberg, Springer-Verlag (2008) 246–258
12. Sintorn, E., Assarsson, U.: Fast parallel gpu-sorting using a hybrid algorithm. Volume 68., Orlando, FL, USA, Academic Press, Inc. (2008) 1381–1388
13. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore gpus. In: Proceedings 23rd IEEE International Parallel and Distributed Processing Symposium. (2009)