High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs

Xiaochun Ye[1], Dongrui Fan[1], Wei Lin[1], Nan Yuan[1], Paolo Ienne[1, 2]

[1] Key Laboratory of Computer System and Architecture
Institute of Computing Technology (ICT)
Chinese Academy of Sciences, Beijing, China
{yexiaochun, fandr, linwei, yuannan}@ict.ac.cn

[2] Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
Paolo.Ienne@epfl.ch

***Abstract* - Sorting is a kernel algorithm for a wide range of applications. In this paper, we present a new algorithm, GPU-Warpsort, to perform comparison-based parallel sort on *Graphics Processing Units* (GPUs). It mainly consists of a bitonic sort followed by a merge sort.**

Our algorithm achieves high performance by efficiently mapping the sorting tasks to GPU architectures. Firstly, we take advantage of the synchronous execution of threads in a warp to eliminate the barriers in bitonic sorting network. We also provide sufficient homogeneous parallel operations for all the threads within a warp to avoid branch divergence. Furthermore, we implement the merge sort efficiently by assigning each warp independent pairs of sequences to be merged and by exploiting totally coalesced global memory accesses to eliminate the bandwidth bottleneck.

Our experimental results indicate that GPU-Warpsort works well on different kinds of input distributions, and it achieves up to 30% higher performance than previous optimized comparison-based GPU sorting algorithm on input sequences with millions of elements.

Keywords - Sorting Algorithm; Many-Core; GPU; CUDA; Bitonic Network; Merge Sort

I. INTRODUCTION

Parallel algorithms for sorting have been studied since at least the 1960s. Bitonic sort [1] was one of the first algorithms designed for parallel machines, and since then hundreds of articles on parallel sorting have appeared. However, as the development and variety of parallel architectures expands, these algorithms need to be mapped to different kinds of new platforms.

Recently, *Graphics Processing Units* (GPUs) have been at the leading edge of many-core parallel architectures. They are able to provide considerably higher peak computing power than CPUs [2]. For example, current NVIDIA's GTX285 GPUs contain up to 240 processing units per chip and provide a peak performance of 1 TFLOPS [3]. Moreover, AMD/ATI's HD4890 GPUs attain more than 1.3 TFLOPS [4]. In addition to the high computing power, modern GPUs also introduce efficient features for general-purpose computation, such as scatter and atomic operations, thread synchronization, and on-chip shared memory. So far, GPUs have become very important parallel computing platforms for various kinds of applications [23].

In this paper we present an efficient comparison-based sorting algorithm for CUDA-enabled GPUs [5]. It achieves higher performance than prior comparison-based GPU sorting algorithms. Our algorithm is mainly composed of two parts: Firstly, we divide the input sequence into equal sized subsequences and use a bitonic network to sort each of them. After that, a merge sort follows to merge all small subsequences into the final result. In order to map the sorting tasks to the GPU architecture efficiently, our algorithm takes advantage of the synchronous execution of threads in a warp, and organizes the bitonic sort and merge sort using one warp as a unit. Thus, we call it GPU-Warpsort.

Some of the contributions of our work include: (1) We produce a fast bitonic sort with no barriers by assigning independent bitonic sorting networks to each warp. (2) We provide sufficient homogeneous parallel comparisons to all the threads in a warp to avoid branch divergence in the bitonic sort. (3) Our merge sort assigns to each warp independent pairs of sequences to be merged, making use of the barrier-free bitonic network. At the same time, it maintains coalesced global memory access to eliminate bandwidth bottlenecks. (4) We demonstrate that a bitonic network is faster than rank-based method [17] when used in our warp-based merge sort.

We implement our sorting algorithm on a PC with a modern NVIDIA 9800GTX+ GPU. Our results indicate up to 30% higher performance than state-of-the-art optimized comparison-based algorithms.

The rest of the paper is organized as follows: In Section Ⅱ, we present related work. In Section Ⅲ, we give an overview of the GPU computation model and highlight some of the features critical to our algorithm. We present our GPU-Warpsort in Section Ⅳ and give a complexity analysis in Section Ⅴ. In Section Ⅵ, we describe the experimental results. We summarize the paper and present future work in Section Ⅶ.

II. RELATED WORK

In this section, we briefly survey related work in GPU sorting algorithms and analyze the limitations of prior implementations.

A. Sorting on GPUs

Because of the restrictions of GPU hardware and programming mode, early sorting implementations were often based on Batcher's bitonic sort [1]. Purcell *et al*. [6] presented an implementation of bitonic sort on GPUs based on an implementation by Kapasi *et al*. [7]. Kiper *et al*. [8] showed an improved version of the bitonic sorting network as well as an odd-even merge sort. Later, Greß *et al*. [9]

978-1-4244-6443-2/10/$26.00 ©2010 IEEE

designed the GPU-abisort utilizing adaptive bitonic sorting [10], improving the complexity of bitonic sort from $O(n(\log n)^2)$ to $O(n \log n)$. The results of GPU-abisort show that it is slightly faster than the bitonic sort system GPUSort of Govindaraju *et al*. [11].

All the sorting algorithms mentioned above were implemented using traditional graphics-based *General-Purpose computing on GPUs* (GPGPU) programming interfaces, such as OpenGL and DirectX API. With the introduction of new general-purpose programming model such as CUDA [5], sorting algorithms can be implemented more efficiently on modern GPUs.

Harris *et al*. [13] described a split-based radix sort followed by a parallel merge sort. Yet, the parallelism of merge sort decreases geometrically and the authors did not address the problem.

Le Grand [15] and He *et al*. [16] implemented a similar radix sort based on histograms. Their schemes did not make efficient use of memory bandwidth and were not competitive for large arrays [17].

Sengupta *et al*. [12] presented a radix sort and a quicksort implemented based on segmented scan primitives. Their radix sort was obviously faster than that of Harris *et al*. [13]. However, their GPU quicksort produced poor performance.

Sintorn *et al*. [14] designed a fast hybrid algorithm that combines bucket sort and merge sort. Their algorithm was further adapted to use dual graphics cards to achieve another 1.8 times speedup.

Later, Cederman *et al*. [18] developed a more efficient implementation of GPU quicksort by using explicit partitioning for large sequences coupled with bitonic sort for small sequences. The performance of GPU quicksort is comparable to that of Sintorn's hybrid sort [14].

Recently, Satish *et al*. [24, 17] described a new radix sort and a rank-based merge sort for many-core GPUs. The results demonstrated theirs to be the fastest GPU sort and the fastest comparison-based GPU sorting algorithm, respectively. Their radix sort is also available in the NVIDIA CUDA SDK [19] (version 2.2).

Another sorting algorithm available in NVIDIA CUDA SDK is bitonic sort. However, this version of bitonic sort is single-block only and not very useful in practice. Barajlia *et al*. [20] presented a practical bitonic sorting network implemented with CUDA, and the results showed its competitiveness with respect to the quicksort of Cederman *et al*. [18] for large arrays.

B. Limitations of Prior Implementations

Although they have taken advantage of some new features provided by the CUDA programming model, previous GPU sorting algorithms still have several limitations.

Radix sort has been proven to be the fastest GPU sorting algorithm [17]. And some other sorting algorithms also use radix sort as a component part [14, 15]. However, radix sort is less generic since it is not comparison based.

Among the comparison-based sorting algorithms, bitonic sort has been implemented with CUDA by Baraglia *et al*. [20]. However, the relatively high complexity limits its efficiency for large arrays. Besides, whenimplemented on GPUs, this high complexity also introduces a large number of memory accesses and thus bandwidth bound.

Quick sort is normally seen as one of the fastest algorithms on traditional CPUs, but it has not shown advantageous performances on GPUs [17,18].

Merge sort is one of the most widely used sorting algorithms on modern GPUs [13,14,17]. Although it is a fast sort, current implementations still have some limitations. Harris *et al*. [13] do not address the problem that the number of pairs to be merged decreases geometrically and the parallelism will be insufficient in many iterations towards the end. Sintorn *et al*. [14] map an independent merge sort to each thread and the input sequence being sorted has to be divided into thousands of independent subsequences in order to keep the parallelism and maintain high efficiency. However, this is not easy. Their results show that the bucket sort used to divide the input sequence costs more than half of the total execution time. Another problem of this algorithm is that the global memory accesses are not coalesced in merge sort and it induces a bandwidth bottleneck. Satish *et al*. [17] use an odd-even network to sort the small subsequences in the beginning and implement a rank-based merge sort to merge these sorted tiles. Their odd-even network sorts t values with a t-thread block. In this case, barriers are necessary between different stages and half of the threads are idle during the compare-and-swap operations. Besides, in their merge sort, splitting operations are frequently performed to expose fine-grain parallelism, costing nearly 1/5 of the total execution time [24].

In this paper, we address these problems and present an efficient algorithm that achieves better performance than prior comparison-based GPU sorting algorithms.

III. GPU COMPUTATION MODEL

Our algorithm is based on the NVIDIA GPU architecture [21] and the CUDA programming model [5].

The GPU is a massively multi-threaded data-parallel architecture, which contains hundreds of processing cores. Eight cores are grouped into a *Streaming Multiprocessor* (SM), and cores in the same SM execute instructions in a *Single Instruction Multiple Thread* (SIMT) fashion [5, 21].

To support general-purpose computation, the CUDA programming model introduces several new features, such as scatter operation, thread synchronization, and on-chip shared memory. It considers the programs executed on GPU as kernels, which usually consist of thousands of threads. Threads have a three-level hierarchy: A grid is a set of blocks that execute a kernel, and each block consists of hundreds of threads. All threads within a block can share the same on-chip memory and can be synchronized at a barrier. Each block can only execute on one SM.

When we designed the sorting algorithm on GPUs, we paid most careful attention to the following aspects: warp-based thread scheduling, SIMT execution model, and global memory access pattern.

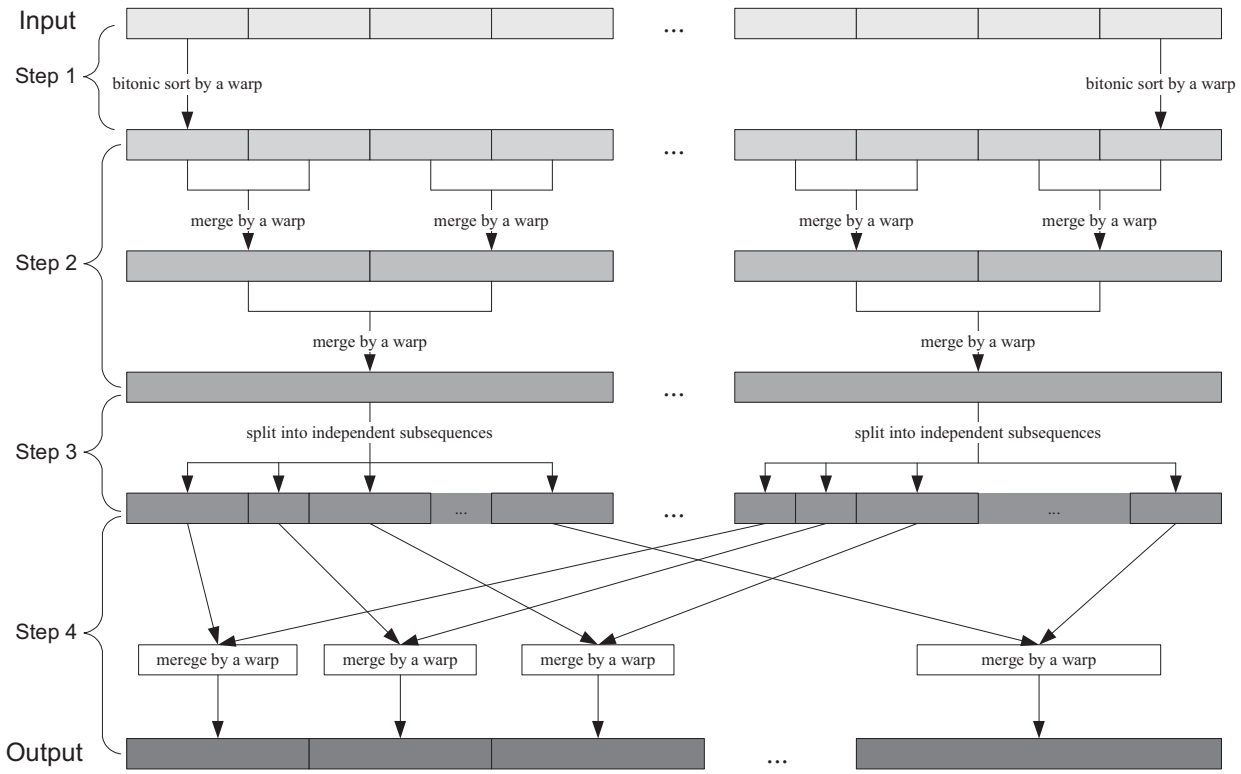


Figure 1. Overview of GPU-Warpsort. It mainly consists of four steps: (1) Use warp-based bitonic networks to sort the equal-sized subsequences. (2) Merge the subsequences with warp-based bitonic network until the parallelism is insufficient. (3) Split the merged sequences into smaller independent subsequences. (4) Merge the small subsequences into the final results.

A. *Warp-Based Scheduling and SIMT Execution Model*

During execution, 32 threads from a continuous section are grouped into a warp, which is the scheduling unit on each SM. Different warps are free to branch and execute different instructions independently. However, threads within a warp execute in a SIMT model, which is similar to SIMD vector organizations. Only a single instruction controls all the processing elements and they execute the same instruction at any instant in time. If threads of a warp diverge due to a data-dependent condition branch, the warp serially executes each taken path, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

According to the warp-based scheduling method and SIMT execution model, the following features can be easily concluded: (1) Threads within a warp execute instructions synchronously, thus there is no need to use barriers with a warp. This feature is also used in the scan primitive by Sengupta *et al*. [22]. (2) Full efficiency can be realized when all 32 threads of a warp agree on their execution paths. However, it is the responsibility of programmers to avoid divergence in a warp whenever possible.

B. *Coalesced and Uncoalesced Memory Accesses*

Modern GPU provides a very high memory bandwidth. But whether the program can fully utilize it or not depends on the global memory access pattern within a warp. For example, if the accessed memory addresses are sequential words and the start address is aligned, sixteen memory requests within a warp can be coalesced into a single memory transaction. Otherwise, uncoalesced memory accesses take place and several different transactions will be generated. Coalesced memory accesses deliver a much higher efficient bandwidth than uncoalesced accesses, thus greatly affecting the performance of bandwidth-bound programs. Detailed coalesced memory access requirements for different CUDA compute capabilities can be found in the CUDA programming guide [5].

IV. Implementation of GPU-Warpsort

Because of the relatively high complexity, bitonic sort usually performs more efficiently when sorting small sequences. Thus in our implementation, bitonic sort is only used to sort small independent subsequences.

As mentioned in section Ⅱ, taking threads or blocks as a unit to organize the merge sort has some issues, such as the difficulty to provide enough parallelism or the need of frequent barrier operations. Instead of that, GPU-Warpsort uses a warp-based method to map sorting tasks to GPUs efficiently. It mainly consists of two novel sorting implementations: The first part is a barrier-free and divergence-free bitonic sort, and the second one is a fast merge sort with totally coalesced global memory accesses.

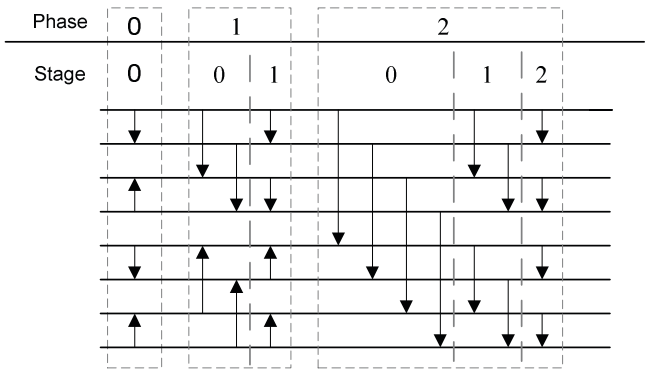


Figure 2. Bitonic sorting network for 8-element sequence. It consists of 3 phases, and phase k ($0 \le k \le 3$) includes k+1 stages.

Fig. 1 gives an overview of our sorting algorithm. It mainly consists of 4 steps as follows:

(1) Divide the input sequence into equal-sized subsequences. Each subsequence will be sorted by an independent warp using the bitonic network.
(2) Merge all the subsequences produced in step 1 until the parallelism is insufficient.
(3) Split the large subsequences produced in step 2 into small ones that can be merged independently.
(4) Merge the small subsequences produced in step 3 by different wraps or blocks independently.

A. Barrier-Free Bitonic Sort (Step 1)

We assume that the input sequence is a continuous array, so it can be easily divided into equal-sized subsequences. Then, each subsequence can be sorted independently using a bitonic network.

Normally, bitonic sorting network of n elements consists of $\log n$ phases and phase k ($0 \le k \le \log n$) includes k+1 stages (see Fig. 2). In each stage, pairs of two elements are selected and followed by compare-and-swap operations. A global synchronization is needed between adjacent stages. To avoid this synchronization overhead, our algorithm sorts each subsequence by a warp, rather than more commonly by a block. This exploits the advantage of synchronous execution of threads in a warp to eliminate the need for barriers.

There are some other features which should be noticed: Firstly, n elements correspond to only n/2 compare-and-swap operations in each stage. Secondly, half of the compare-and-swap operations are to form ascending pairs, while the other half are to form descending pairs, excepting in the steps of the last phase, where all the comparisons form the same sequence order. In order to avoid idle threads' or divergent operations' appearing within a warp, we ensure that 32 threads in a warp perform 32 distinct compare-and-swap operations at the same time, and all these operations form the sorted pairs with the same order. This means that at least 128 elements are needed for each warp.

As a result, we divide the input sequence into equal-sized subsequences which contain at least 128 elements. The subsequences are loaded into shared memory with coalesced memory accesses and sorted by different warps independently. Threads within a warp are synchronized by hardware; there is no need to use the barrier, which is the *__syncthreads()* function in CUDA.

```
bitonic_warp_128_(key_t *keyin, key_t *keyout) {
  //phase 0 to log(128)-1
  for(i=2;i<128;i*=2){
    for(j=i/2;j>0;j/=2){
      k0 ?  position of preceding element in each pair
            to form ascending order
      if(keyin[k0]>keyin[k0+j])
        swap(keyin[k0],keyin[k0+j]);
      k1 ?  position of preceding element in each pair
            to form descending order
      if(keyin[k1]<keyin[k1+j])
        swap(keyin[k1],keyin[k1+j]);
    }
  }
  //special case for the last phase
  for(j=128/2;j>0;j/=2){
    k0 ?  position of preceding element in the thread's
          first pair to form ascending order
    if(keyin[k0]>keyin[k0+j])
      swap(keyin[k0],keyin[k0+j]);
    k1 ?  position of preceding element in the thread's
          second pair to form ascending order
    if(keyin[k1]>keyin[k1+j])
      swap(keyin[k1],keyin[k1+j]);
  }
}
```

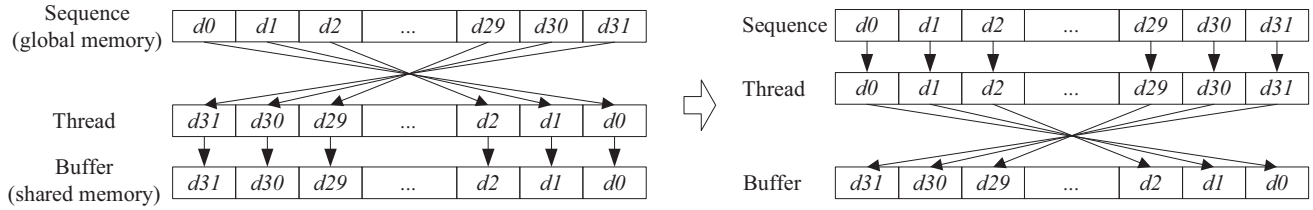
Figure 3. Pseudo code of sorting 128 elements with bitonic network by a warp in step1. We ensure that 32 threads in a warp perform 32 distinct compare-and-swap operations at the same time, and all these operations form the sorted pairs with the same order.

The pseudo code of sorting 128 elements with bitonic network by a warp is shown in Fig. 3. Two points need to be noted: First, during the last phase, all the comparisons form the same sequence order, which is ascending in this example. Second, the if-swap pairs in the pseudo code can be implemented more efficiently using the *max()* and *min()* functions provided by CUDA, thus avoiding the branch divergence and resulting in a slightly better performance.

B. Bitonic-Based Merge Sort (Step 2)

In this step, each pair of sorted subsequences is merged by a warp, rather than a thread or a block, in order to eliminate the frequent synchronization while at the same time maintaining coalesced global memory accesses. This warp-based merge sort is similar to the strategy we used in step 1.

Assume that two input sequences A and B are merged by a warp (Fig. 4 gives an example of merging two 8-element sequences). The warp fetches t/2 elements from sequence A



(a) uncoalesced global memory access

(b) coalesced global memory access

Figure 5. Global memory access patterns when fetching sequence elements.

and $t/2$ elements from sequence B every time (t = 8 in Fig. 4). Our bitonic network contains t elements and a t-element buffer in shared memory is allocated to store them. First, the warp fetches the $t/2$ smallest elements from sequence A and the $t/2$ smallest elements from sequence B, storing them into the buffer. After a bitonic merge sort, the smallest $t/2$ elements of the result are output, and the remaining $t/2$ elements remain in the buffer. Next, the lower $t/2$ entries of the buffer are filled by new $t/2$ elements from sequence A or B depending on the result of comparing the maximum elements fetched last time. Assume $a_{\max}$ is the maximum of $t/2$ elements fetched from sequence A last time, and $b_{\max}$ is the correspondence from B. If $a_{\max} \le b_{\max}$, the warp fetches t elements from sequence A; otherwise, it fetches them from sequence B. Note that it is different from the method used in vector-mergesort [14]. Vector-mergesort fetches the next vectors from both sequences A and B, and decides which one to be used by comparing the minimum elements of the two vectors. As a result, vector-mergesort has a problem of bandwidth waste which does not exist with our strategy.

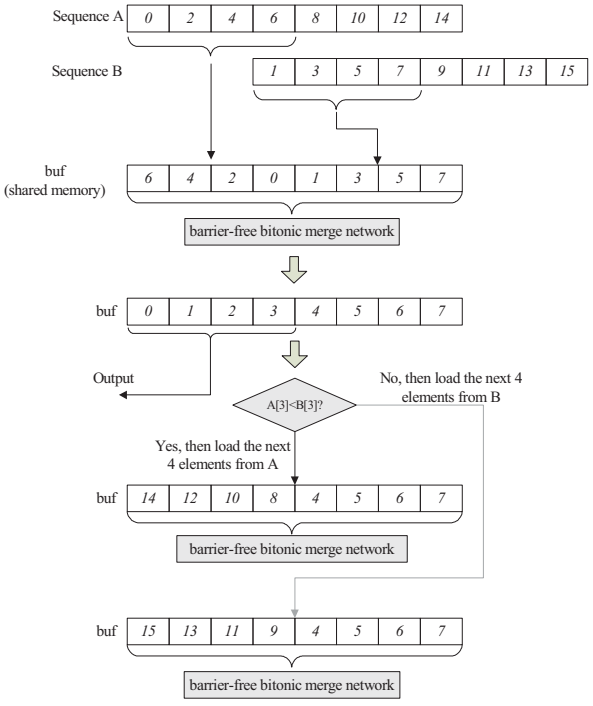


Figure 4. An example of merge sort for two eight-element sequences by a warp. Each time, the warp fetches $t/2$ (t = 8 in this case) elements from sequence A or B and stores them into the buffer in shared memory. It uses a barrier-free bitonic merge network to sort them and outputs the lower half part. The others are sorted with another t elements from either sequence A or B.

Similar to step 1, the bitonic merge network is also barrier-free. However, it should be noticed that this bitonic network is much simpler than that used in step 1. The two $t/2$-element input sequences of the bitonic network have already been sorted. According to the definition of bitonic sequence, we can easily assemble the two $t/2$-element input sequences into a t-element bitonic array. For example, the first $t/2$-element sequence is stored in the buffer in a descending order and the second one in an ascending order. Therefore, only the last phase of bitonic network is necessary and each warp only contains $\log t$ stages.

To avoid thread idling, t should be at least 64. Fortunately, the shared memory is large enough to provide 64-element buffers for all the warps executing simultaneously on the same SM.

Because the $t/2$ elements stored in the lower buffer entries are in a descending order, which is the opposite of the order in the input sequence, an intuitive method is that all the threads in a warp fetch elements in a reverse order. For example, if $t/2$ equals 32, then thread i reads the (31-i)-th element and writes it to i-th buffer entry, as shown in Fig. 5(a). Unfortunately, this method causes uncoalesced global memory access. Instead, each thread can fetch the elements sequentially according to its thread identifier, and reverse them before writing them to the buffer in shared memory. As shown in Fig. 5(b), thread i reads the i-th element, but writes it to (31-i)-th buffer entry. Thus, the global memory accesses can be coalesced. We find that this method eliminates the bandwidth bottleneck and makes our algorithm computation-bound.

Satish *et al*. [17] implement the merge sort by computing the rank of each element. This method is also applicable in our case. To do this, each warp simply computes the rank of elements in the two $t/2$-element input sequences and writes them to the final positions of sorted sequence. However, in section Ⅵ, our results demonstrate that it is not as fast as the bitonic merge sort (see Fig. 13).

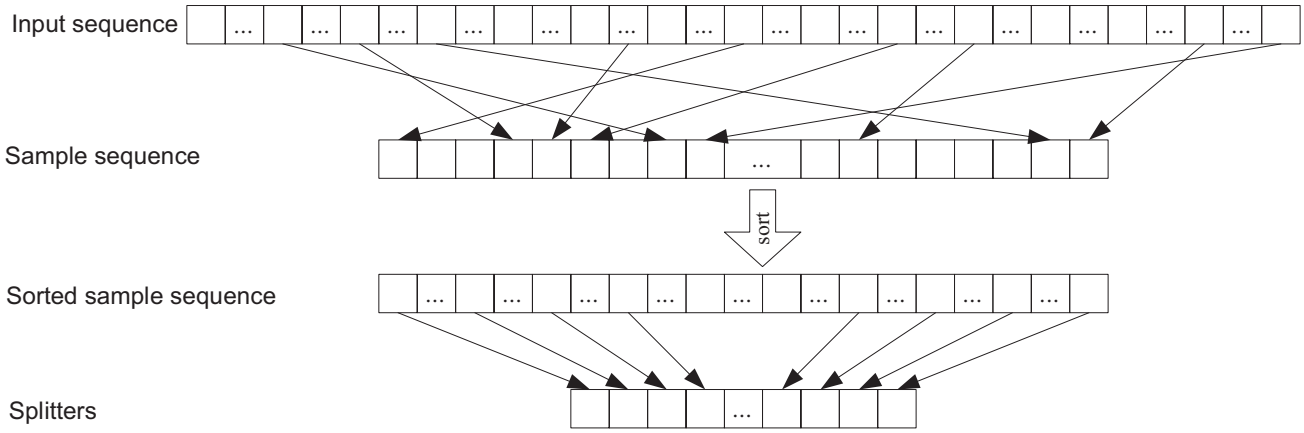


Figure 7. Sample the input data to get the splitters. We sample the input sequence randomly to get $s \cdot k$ sample elements. Then we sort the sample sequence and construct the splitters by selecting every k-th element of the sorted sample sequence.

C. Split into Small Tiles (Step 3)

As we know, in merge sort, the number of pairs to be merged will decrease geometrically. When there are not enough pairs of sequences to be merged, parallelism is lost and it is difficult to achieve high efficiency on massively parallel platforms such as many-core GPUs. As a result, large sequences have to be divided into smaller tiles once they are not able to provide sufficient parallelism.

This operation is quite intuitive. Assume that we have l equal sequences of size n before the split operation, and each sequence will be divided into s subsequences, as show in Fig. 6. After splitting, all of the elements satisfy that the following relation:

$$\forall a \in subsequence(x,i), \forall b \in subsequence(y,j) : a \le b,$$

where

$$0 \le x < l, 0 \le y < l, 0 \le i < j < s.$$

To maintain sufficient parallelism in step 2, l must be larger than the number of SMs. Our experiments show that 64 is a good choice for 16-SM GPUs. Similarly, s must be larger than the number of SMs to provide high parallelism for next step.

In order to construct even splitters, we sample the original input sequence randomly to get $s*k$ sample elements. Then we sort the sample sequence and construct the splitters by selecting every k-th element of the sorted sample sequence, as shown in Fig. 7. Choosing a larger k consumes more time, but produces better splitters. Note that this sample operation is performed before step 1.

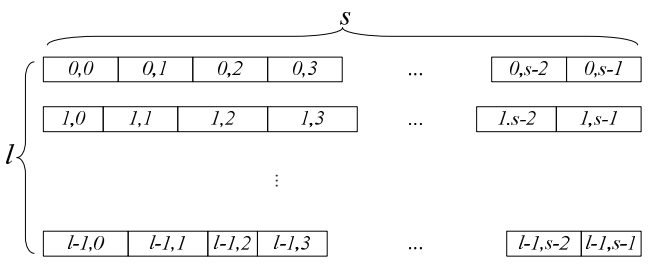


Figure 6. Split each of the l large sequence into s small subsequences to maintain sufficient parallelism.

D. Final Merge Sort (Step 4)

At the end of step 3, we have l large sequences and each of them consists of s subsequences. To get the final sorted result, subsequences $(0,i)$, $(1,i)$,…, $(l\text{-}1,i)$ need to be merged into one sequence S_i. It is obvious that S_0, S_1,…, S_l can be merged independently, thus regenerating sufficient parallelism for many-core GPUs.

Similar to the method used in step 2, each pair of subsequences is merged by a warp. Finally, S_0, S_1,…, S_l are assembled into a totally sorted array.

Note that the start and end addresses of the subsequences after division may not be aligned. In order to avoid uncoalesced global memory access, we patch the start and end of each subsequence by inserting some special keys, if they are misaligned.

V. Complexity Analysis

Our sorting algorithm mainly consists of two parts: bitonic sort in step 1 and merge sort in the following steps. We can discuss the complexity of each part independently first. Then we will get the total complexity of our algorithm.

A. Time Complexity

In the bitonic sort of step 1, we assume the input sequence contains n elements and each equal-sized subsequence has w elements. Then, the average time complexity is $O(n(\log w)^2)$. Because w is a constant in our implementation, the time complexity of step 1 is $O(n)$.

As for the merge sort in the following steps, the merge tree contains $\log(n/w)$ levels. In each level, the bitonic sorting network consists of t elements. Thus, every element needs $O((\log t)^2)$ compare operations in each level, and

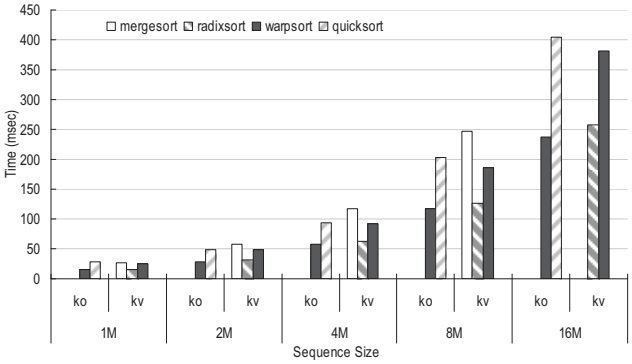


Figure 8. Execution time comparison with other implementations. (ko: key-only; kv: key-value)

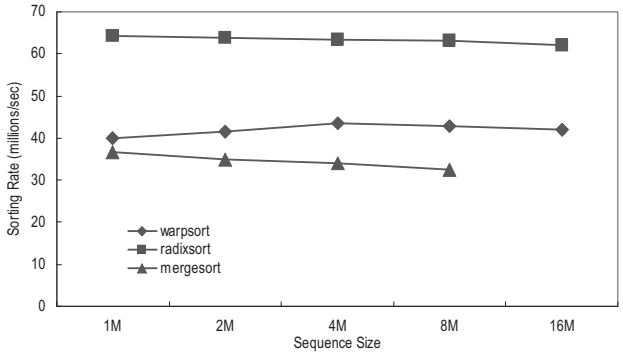


Figure 9. Sorting rate comparison with other GPU-based implementations. (key-value pairs)

the time complexity of merge sort is $O(n\log(n/w)\bullet(\log t)^2)=O(n\log n)$

As a result, the time complexity of our GPU-warpsort is $O(n)+O(n\log n)=O(n\log n)$

B. *Space Complexity*

In our algorithm, bitonic sort works in-place; thus, there is no need to use auxiliary buffer in step 1. However, the merge sort does not work in-place: we use two buffers and switch each other in different levels of the merge tree. In addition, we need a little more auxiliary space for the patching of misaligned address. Thus the space requirement of our algorithm is $2n+c$. The constant c depends on the number of splitters used in step 3.

VI. Experimental Results and Discussion

We have implemented our GPU-Warpsort using the CUDA programming API. We study its performance on a PC with dual core AMD Opteron880 at 2.4 GHz, 2GB RAM and a NVIDIA 9800GTX+ with 512 MB of on board Graphics RAM. The host is running Red Hat Linux. We test our implementation with key-only and key-value configurations where both key and values are 32-bit words.

A. *Performance Evaluation*

We compare the performance of GPU-Warpsort to state-of-the-art implementations on GPUs. To be best of our knowledge, Satish *et al*. [17] have implemented both the fastest published sorting algorithm and the fastest comparison-based sorting algorithm for modern GPU processors so far. As a result, the algorithms that have been demonstrated to be slower than Satish's implementation are not compared in this paper: this includes the radix sort published by Le Grand in GPU Gems 3 [15], the radix sort implemented by Sengupta *et al*. [12], and the bitonic sort based GPUSort of Govindaraju *et al*. [11]. In addition to Satish's implementation, we also compare our algorithm to two other comparison-based algorithms: the quick sort designed by Cederman *et al*. [18] and the bitonic sort of Baraglia *et al*. [20]. We measure the GPU execution time only and do not include the data transfer time between the host and GPU memory. The input arrays are randomly generated sequences whose lengths range from 1M elements to 16M elements. Unless otherwise specified, we report results for sequences with uniform random distribution.

Fig. 8 and Fig. 9 report the execution time and sorting rate of different algorithms on the 9800GTX+. Among these algorithms, GPU-Warpsort, quicksort and merge sort are comparison-based. It is obviously that GPU-Warpsort is substantially faster than quicksort. For key-only sequences, GPU-Warpsort achieves 1.7 times higher performance on average. Cederman *et al*. did not implement a key-value variant of their quicksort, but, according to Fig. 8, the result of key-value warpsort is even faster than that of key-only quicksort. Therefore, we believe that our algorithm will also be much faster than quicksort for key-value sequences. When compared to the merge sort of Satish *et al*., which was the fastest comparison-based GPU sorting algorithm, GPU-Warpsort sports a performance over 20% higher on average; for sequences larger than 4M, GPU-Warpsort even achieves up to 30% higher performance.

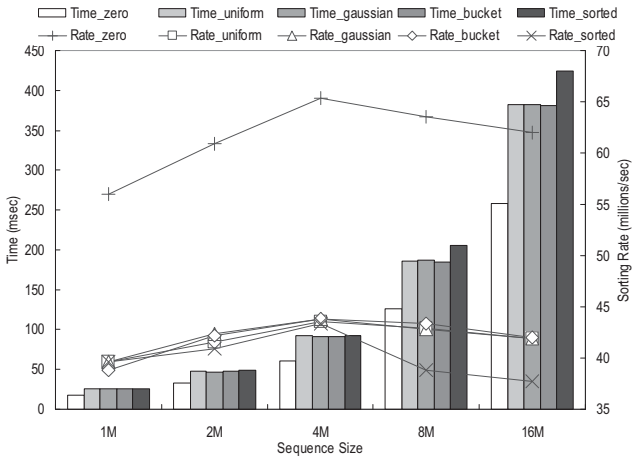


Figure 10. Sort performance for different distributions.

(key-value pairs)

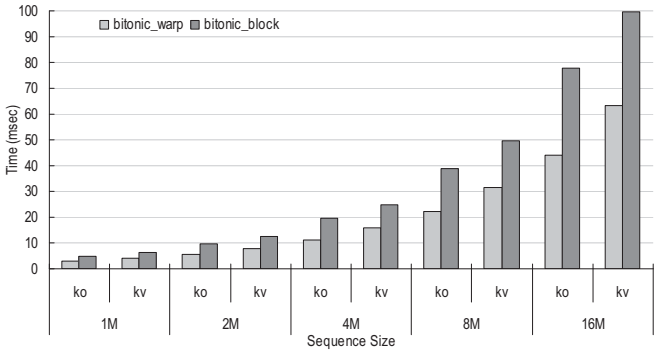


Figure 11. Performance of bitonic sort in step 1

(ko: key-only; kv: key-value)

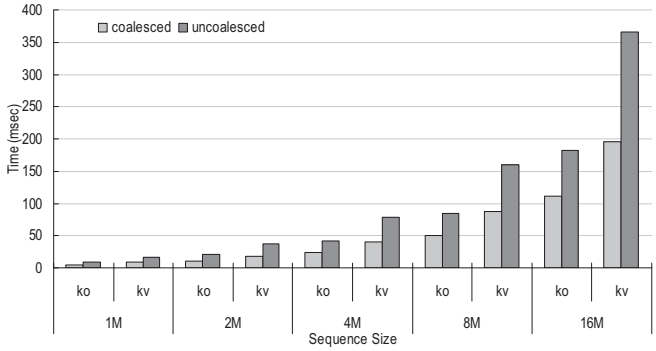


Figure 12. Performance of merge sort in step 2

(ko: key-only; kv: key-value)

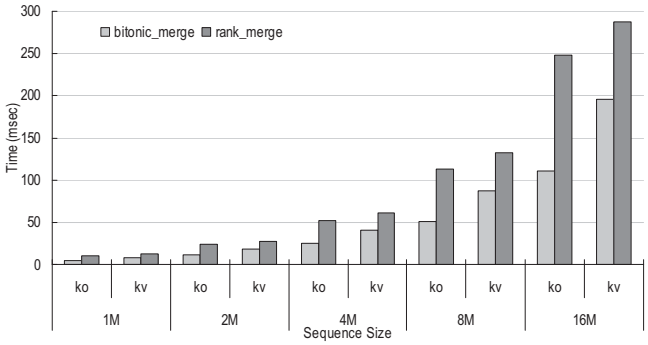


Figure 13. Performance of rank-merge and bitonic-merge

(ko: key-only; kv: key-value)

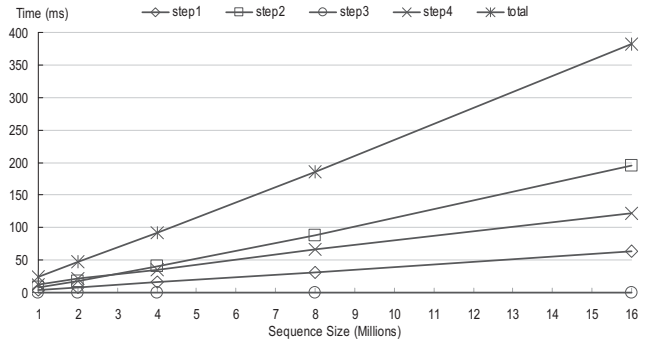


Figure 14. Time taken by different steps

(use key-value pairs, uniform distribution)

Nevertheless, our results show that radix sort is still the fastest sorting algorithm on GPUs. Our key-only warpsort is only as fast as the key-value radix sort in most cases and our key-value warpsort is about 40% slower. But radix sort is not comparison-based, thus not as generic as ours.

We could not find the source code available for the bitonic sort of Baraglia, but, according to their reported results, the performance of their bitonic sort is similar to that of the quicksort designed by Cederman *et al*. [18]. Thus GPU-Warpsort should perform substantially better than bitonic sort.

We also tested our sort on different key distributions, including zero, sorted, uniform, bucket, and Gaussian distributions (see Fig. 10). GPU-Warpsort shows about the same performance on uniform, bucket, and Gaussian distributions. Besides, applying it to zero distribution sequences does result in obviously shorter execution time. On average, it achieves 15% higher performance for key-only sequences and 46% higher performance for key-value sequences. This improved result can be mainly attributed to three reasons: First, when sorting zero-distributed sequence with a bitonic network, all the comparison operations produce false results and no swap operation is performed. Secondly, after the split operation in step 3, all large sequences will be divided into equal-sized subsequences. This produces a balanced task for each block or thread. Third, there is no necessity to patch the borders of subsequences in step 4, because zero-distributed sequences are always split into aligned subsequences. We also note that our algorithm becomes worse when the input sequence is already sorted. This is mainly because of the load imbalance occurring in the last step. Our algorithm does not excel on nearly sorted sequences.

Next, we evaluate the impact of warp-based bitonic sort and global memory access patterns. For comparison, we implement a block-based bitonic sort in step 1 and a merge sort with uncoalesced global memory accesses in step 2. The block-based bitonic sort assigns t elements to a t-thread block and uses barriers to synchronize with other threads. The merge sort is implemented according to Fig. 5(a). Figs. 11 and 12 show the performance results. Because warp-based bitonic avoids barrier operations and thread idling, it achieves 76% and 57% higher performance than block-based bitonic sort for key-only and key-value input sequences, respectively. As for merge sort, the one with coalesced memory accesses also shows an obvious advantage over that with uncoalesced accesses. According to Fig. 12, the former is 74% and 94% faster than the latter for key-only and key-value sequences, respectively. Further analysis shows that

coalesced global memory accesses have changed our sorting algorithm from bandwidth-bound to computation-bound.

As mentioned earlier, both the rank-based merge sort and bitonic merge sort can be used in step 2 of GPU-Warpsort. However, Fig. 13 shows that bitonic merge sort is clearly faster than rank-based merge sort in our situation.

Finally, we examine the breakdown of execution time in Fig. 14. The split step (step 3), which takes nearly 1/5 of the total time in Satish's radix sort, costs only a negligible part of time in our algorithm. The execution time of other three steps scales well as the sequence size increases. The input sequence can be easily divided into equal-sized subsequences, thus both steps 1 and 2 show steady execution time in proportion to the sequence sizes. The result of step 4 shows that it is more efficient when sorting large arrays. Thus, one can hope that the advantage could be even larger over other algorithms when sorting larger input sequences.

B. *Discussion*

Our results clearly show that GPU-Warpsort delivers substantially higher performance than other comparison-based sorting algorithms. Based on the experimental observations, we get the following insights that are helpful for performance tuning on modern many-core GPUs:

- First, although bitonic sort has a relatively high computation complexity, it is still very efficient when sorting small sequences. Besides, it can provide high parallelism even for a small number of elements. This is very beneficial for massively parallel platforms such as many-core GPUs.
- Second, GPUs usually only provide a simple barrier scheme for synchronization. Thus, programs show very low performance when frequent synchronizations are required. Taking advantage of the hardware synchronization among threads within a warp to eliminate barriers, as we have used, is a practical alternative for solving this problem.
- Third, coalesced global memory is a critical issue for the performance of algorithms on GPUs, especially for bandwidth-bound algorithms such as sort. In these situations, it is wise to avoid uncoalesced global memory at the cost of introducing more computation or consuming more memory capacity.

VII. Conclusions and Future Work

In this paper, we present an efficient comparison-based sorting algorithm for many-core GPUs. We compare our results to state-of-the-art algorithms. In practice, our results demonstrate up to 30% better performance than previous optimized comparison-based algorithms for input sequences with millions of elements. As far as we know, ours is the fastest comparison-based sorting algorithm for modern many-core GPUs.

We achieve the outstanding performance by efficiently mapping the sorting tasks to GPU architecture. Taking advantage of the synchronous execution of threads in a warp, we carefully organize the tasks for each warp to eliminate the need for barriers. We also provide sufficient homogeneous parallel operations for each thread within a warp to avoid thread idling or thread divergence. Thus, we believe that we make the best use of all the computation resources on the chip. Our merge sort uses totally coalesced global memory accesses when fetching and storing the sequence elements; this leads to highly efficient bandwidth utilization.

As part of our planned future work, we hope to implement our sorting algorithm on dual graphics cards. We would also like to perform more experiments and study the scalability issues on newer generation GPUs and on other data-parallel architectures.

Acknowledgment

We would like to thank all reviewers for their insightful feedback. This work is supported by the National Grand Fundamental Research 973 Program of China (No. 2005CB321600), the National Natural Science Foundation of China (No. 60736012 and No. 60752001), the National High-Tech Research and Development Plan of China (No.2009AA01Z103), the National Science Foundation for Distinguished Young Scholars of China (No.60925009), the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (No. 60921002), and the Beijing Natural Science Foundation (No.4092044).

References

[1] K. E. Batcher, "Sorting networks and their applications," in AFIPS Spring Joint Computer Conference, vol. 32, 1968, pp. 307–314.

[2] J. D. Owens, D. Luebke, N. K. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In Eurographics 2005, State of the Art Reports, August 2005, pp. 21-51.

[3] NVIDIA Geforce series GTX285. http://www.nvidia.com/geforce.

[4] ATI Mobility RadeonTM HD4890 Graphics-Overview. http://ati.amd.com/products.

[5] NVIDIA CUDA Programming Guide, NVIDIA Corporation, Jun. 2008, version 2.0.

[6] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In Proceedings of the ACM SIGGRAPH Conference on Graphics Hardware, pages 41–50. Eurographics Association, 2003.

[7] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture, pages 159–170. ACM Press, 2000.

[8] P. Kipfer and R. Westermann. Improved GPU sorting. In M. Pharr, editor, GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, pages 733–746. Addison-Wesley, 2005.

[9] A. Greß and G. Zachmann. Gpu-abisort: Optimal parallel sorting on stream architectures. In The 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS '06), page 45, Apr. 2006.

[10] G. Bilardi, A. Nicolau, Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines, SIAM Journal of Computation 18 (2) (1989) 216–228.

[11] N. K. Govindaraju, J. Gray, R. Kumar, D. Manocha, Gputerasort: High performance graphics coprocessor sorting for large database management, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2006.

[12] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics hardware, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007.

[13] M. Harris., S. Sengupta, J. D. Owens.: Parallel prefix sum (scan) with CUDA. In GPU Gems 3, Nguyen H., (Ed.). Addison Wesley, Aug. 2007, ch. 31.

[14] E. Sintorn and U. Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. Journal of Parallel and Distributed Computing, 2008.

[15] S. Le Grand, "Broad-phase collision detection with CUDA," in GPU Gems 3, H. Nguyen, Ed. Addison-Wesley Professional, Jul. 2007, ch. 32, pp. 697–721.

[16] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in Proc. ACM/IEEE Conference on Supercomputing, 2007, pp. 1–12.

[17] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In IPDPS, 2009.

[18] D. Cederman and P. Tsigas, "A practical quicksort algorithm for graphics processors," in Proc. 16th Annual European Symposium on Algorithms (ESA 2008), Sep. 2008, pp. 246–258.

[19] "NVIDIA CUDA SDK," http://www.nvidia.com/cuda, 2009.

[20] R. Baraglia, G. Capannini, F. M. Nardini, and F. Silvestri. Sorting using Bitonic Network with CUDA. In the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR) , Boston, USA, July, 2009.

[21] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," IEEE Micro, vol. 28, no. 2, pp. 39–55, Mar/Apr 2008.

[22] S. Sengupta, M. Harris, and M. Garland. Efficient parallel scan algorithms for GPUs. NVIDIA Technical Report NVR-2008-003, December 2008.

[23] M. Garland, S. Le Grand, J. Nickolls, *et al*.. Parallel Computing Experiences with CUDA," IEEE Micro, vol. 28, no. 4, pp. 13–27, 2008.

[24] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. NVIDIA Technical Report NVR-2008-001, Sep. 2008.