A Practical Quicksort Algorithm for Graphics Processors

Daniel Cederman* and Philippas Tsigas**

Department of Computer Science and Engineering
Chalmers University of Technology, SE-412 96 Göteborg, Sweden
{cederman,tsigas}@chalmers.se

Abstract. In this paper we present GPU-Quicksort, an efficient Quicksort algorithm suitable for highly parallel multi-core graphics processors. Quicksort has previously been considered as an inefficient sorting solution for graphics processors, but we show that GPU-Quicksort often performs better than the fastest known sorting implementations for graphics processors, such as radix and bitonic sort. Quicksort can thus be seen as a viable alternative for sorting large quantities of data on graphics processors.

1 Introduction

In this paper, we present an efficient parallel algorithmic implementation of Quicksort, GPU-Quicksort, designed to take advantage of the highly parallel nature of graphics processors (GPUs) and their limited cache memory. Quicksort has long been considered as one of the fastest sorting algorithms in practice for single processor systems, but until now it has not been considered as an efficient sorting solution for GPUs [1]. We show that GPU-Quicksort presents a viable sorting alternative and that it can outperform other GPU-based sorting algorithms such as GPUSort and radix sort, considered by many to be two of the best GPU-sorting algorithms. GPU-Quicksort is designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and inter-thread synchronization needed. It achieves this by using a two-phase design to keep the inter-thread synchronization low and by steering the threads so that their memory read operations are performed coalesced. It can also take advantage of the atomic synchronization primitives found on newer hardware, when available, to further improve its performance.

The obvious way to parallelize Quicksort is to take advantage of its inherent parallelism by just assigning a new processor to each new sequence created in the partitioning step. This means, however, that there will be very little parallelization at the beginning, when the sequences are few and long [2].

Another approach has been to divide each sequence to be sorted into blocks that can then be dynamically assigned to available processors [3, 4]. However,

* Supported by Microsoft Research through its European PhD Scholarship Programme
** Partially supported by the Swedish Research Council (VR)

this method requires extensive use of the atomic synchronization primitive Fetch-And-Add (FAA) which makes it too expensive to use on graphics processors.

Since most sorting algorithms are memory bandwidth bound, there is no surprise that there is currently a big interest in sorting on the high bandwidth GPUs. Purcell et al. [5] have presented an implementation of bitonic merge sort on GPUs based on an implementation by Kapasi et al. [6]. Kipfer et al. [7, 8] have shown an improved version of the bitonic sort as well as an odd-even merge sort. Greß et al. [9] introduced an approach based on the adaptive bitonic sorting technique presented by Bilardi et al. [10]. Govindaraju et al. [11] implemented a sorting solution based on the periodic balanced sorting network method by Dowd et al. [12] and one based on bitonic sort [13]. They later presented a hybrid bitonic-radix sort that used both the CPU and the GPU to be able to sort vast quantities of data [14]. Sengupta et al. [1] have presented a radix-sort and a Quicksort implementation. Recently, Sintorn et al. [15] presented a sorting algorithm that combines bucket sort with merge sort.

2 The System Model

The algorithm has been implemented in CUDA, which is NVIDIA's initiative to enable general purpose computations on their graphics processors. It consists of a compiler for a C-based language which can be used to create kernels that can be executed on the GPU.

General Architecture The high range graphics processors from NVIDIA that supports CUDA currently boasts 16 multiprocessors, each multiprocessor consisting of 8 processors that all execute the same instruction on different data in lock-step. Each multiprocessor supports up to 768 threads and has 16 KiB of fast local memory.

Scheduling Threads are logically divided into *thread blocks* that are assigned to a specific multiprocessor. Depending on how many registers and how much local memory the block of threads requires, there could be multiple blocks running concurrently on a single multiprocessor. If more blocks are needed than there is room for, on any of the multiprocessors, the leftover blocks will be scheduled sequentially.

Synchronization Threads within a thread block can use the multiprocessors local memory and a special thread barrier-function to communicate with each other. The barrier-function forces all threads in the same block to synchronize. Some newer graphics processors support atomic instructions such as Compare-And-Swap and FAA.

Memory Data is stored in a large, global memory that supports both gather and scatter operations. There is no caching available when accessing this memory, but each thread block can use its own, very fast, shared local memory to temporarily copy and store data from the global memory and use it as a manual cache. By letting each thread access consecutive memory locations, it is possible to allow read and write operations to coalesce, which will increase performance.

3 The Algorithm

The following subsection gives an overview of GPU-Quicksort. Section 3.2 will then go into the algorithm in more detail.

3.1 Overview

The method used by the algorithm is to recursively *partition* the sequence to be sorted, i.e. to move all elements that are lower than a specific pivot value to a position to the left of the pivot and to move all elements with a higher value to the right of the pivot. This is done until the entire sequence has been sorted.

In each partition iteration a new pivot value is picked and as a result two new subsequences are created that can be sorted independently. After a while there will be enough subsequences available that each thread block can be assigned one of them. But before that point is reached, the thread blocks need to work together on the same sequences. For this reason, we have divided up the algorithm into two, albeit rather similar, phases.

First Phase In the first phase, several thread blocks might be working on different parts of the same sequence of elements to be sorted. This requires appropriate synchronization between the thread blocks, since the results of the different blocks need to be merged together to form the two resulting subsequences.

Newer graphics processors provide access to atomic primitives that can aid somewhat in this synchronization, but they are not yet available on the high-end graphics processors. Because of that, there is still a need to have a thread block barrier-function between the partition iterations.

The reason for this is that the blocks might be executed sequentially and we have no way of knowing in which order they will be executed. The only way to synchronize thread blocks is to wait until all blocks have finished executing. Then one can assign new subsequences to them. Exiting and reentering the GPU is not expensive, but it is also not delay-free since parameters need to be copied from the CPU to the GPU, which means that we want to minimize the number of times we have to do that.

When there are enough subsequences so that each thread block can be assigned its own subsequence, we enter the second phase.

Second Phase In the second phase, each thread block is assigned its own subsequence of input data, eliminating the need for thread block synchronization. This means that the second phase can run entirely on the graphics processor. By using an explicit stack and always recurse on the smallest subsequence, we minimize the shared memory required for bookkeeping.

Hoare suggested in his paper [16] that it would be more efficient to use another sorting method when the subsequences are relatively small, since the overhead of the partitioning gets too large when dealing with small sequences. We decided to follow that suggestion and sort all subsequences that can fit in the available local shared memory using an alternative sorting method.

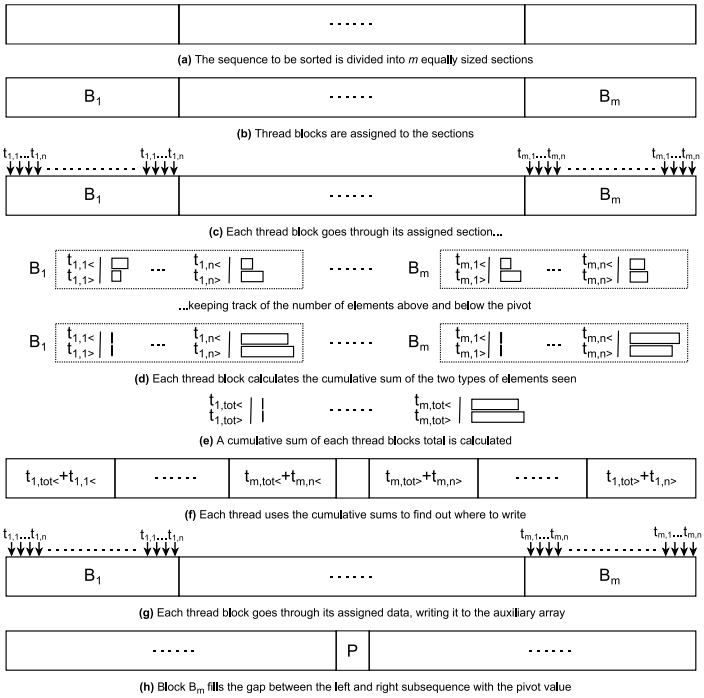

Fig. 1. Partitioning a sequence (m thread blocks with n threads each).

In-place On conventional SMP systems it is favorable to perform the sorting in-place, since that gives good cache behavior. But on GPUs, because of their limited cache memory and the expensive thread synchronization that is required when hundreds of threads need to communicate with each other, the advantages of sorting in-place quickly fades away. Here it is better to aim for reads and writes to be coalesced to increase performance, something that is not possible on conventional SMP systems. For these reasons it is better, performance-wise, to use an auxiliary buffer instead of sorting in-place.

So, in each partition iteration, data is read from the primary buffer and the result is written to the auxiliary buffer. Then the two buffers switch places, with the primary becoming the auxiliary and vice versa.

Partitioning The principle of two phase partitioning is outlined in Figure 1. The sequence to be partitioned is selected and it is then logically divided into m equally sized sections (Step a), where m is the number of thread blocks available. Each thread block is then assigned a section of the sequence (Step b).

The thread block goes through its assigned data, with all threads in the block accessing consecutive memory so that the reads can be coalesced. This is important, since reads being coalesced will significantly lower the memory access time.

Synchronization The objective is to partition the sequence, i.e. to move all elements that are lower than the pivot to a position to the left of the pivot in the auxiliary buffer and to move the elements with a higher value than the pivot to the right of the pivot. The problem here is to synchronize this in an efficient way. How do we make sure that each thread knows where to write in the auxiliary buffer?

Cumulative Sum A possible solution is to let each thread read an element and then synchronize the threads using a barrier function. By calculating a cumulative sum[1] of the number of threads that want to write to the left and to the right of the pivot respectively, each thread would know that x threads with a lower thread id than its own are going to write to the left of the pivot and that y threads are going to write to the right of the pivot. Each thread then knows that it can write its element to either buf_{x+1} or $buf_{n-(y+1)}$, depending on if the element is higher or lower than the pivot.

A Two-Pass Solution But calculating a cumulative sum is not free, so to improve performance we go through the sequence two times. In the first pass each thread just counts the number of elements it has seen that have value higher (or lower) than the pivot (Step c). Then when the block has finished going through its assigned data, we use these sums instead to calculate the cumulative sum (Step d). Now each thread knows how much memory the threads with a lower id than its own needs in total, turning it into an implicit memory-allocation scheme that only needs to run once for every thread block, in each iteration.

In the first phase, where we have several thread blocks accessing the same sequence, an additional cumulative sum need to be calculated for the total memory used by each thread block (Step e).

When each thread knows where to store its elements, we go through the data in a second pass (Step g), storing the elements at their new position in the auxiliary buffer. As a final step, we store the pivot value at the gap between the two resulting subsequences (Step h). The pivot value is now at its final position which is why it doesn't need to be included in any of the two subsequences.

3.2 Detailed Description

The First Phase The goal of the first phase is to divide the data into a large enough number of subsequences that can be sorted independently.

Work Assignment In the ideal case, each subsequence should be of the same size, but that is often not possible, so it is better to have some extra subsequences and let the scheduler balance the workload. Based on that observation, a good way to partition is to only partition subsequences that are longer than $minlength = n/maxseq$ and to stop when we have $maxseq$ number of subsequences.

In the beginning of each iteration, all subsequences that are larger than the $minlength$ are assigned thread blocks relative to their size. In the first iteration,

[1] The terms prefix sum or sum scan are also used in the literature.

Fig. 2. Pseudocode for the first phase.

```
procedure GPUQSORT(size, d^prim, d^aux)
    minlength, flip ← size/maxseq, false
    work, done ← {(0, size, flip, piv)}, ∅
    while work ≠ ∅ ∧ |work| + |done| < maxseq
do
        ws, xs ← Σ_{v∈work} (v.end − v.beg)/maxseq, ∅

        for all v ∈ work do
            x ← (v.beg, v.end, v, ⌈(v.end − v.beg)/ws⌉)
            xs ∪ {x}
            for i ← 0, i < x^c − 1, i ← i + 1 do
                beg ← x^s + ws · i
                bl ← bl ∪ {(x, beg, beg + ws)}
            bl ← bl ∪ {(x, x^s + ws · (x^c − 1), x^e)}

        gqsort(bl, d^prim, d^aux);

        for all x ∈ xs do
            ns_1 ← {(x.beg, x^s, flip, piv)}
            ns_2 ← {(x^e, x.end, flip, piv)}
            if x^s − x.beg < minlength then
                done ← done ∪ ns_1
            else
                work ← work ∪ ns_1
            if x.end − x^e < minlength then
                done ← done ∪ ns_2
            else
                work ← work ∪ ns_2

        d^prim, d^aux, flip ← d^aux, d^prim, ¬flip

    if flip then
        d^prim, d^aux ← d^aux, d^prim
    done ← done ∪ work
    lqsort(done, d^prim, d^aux);
```

```
procedure GQSORT(bl, d^prim, d^aux)
    b ← bl_bid
    lt_tid, gt_tid ← 0, 0
    i ← b^beg + tid

    for i < b^end, i ← i + T do
        if d_i^prim < b^{x^p} then
            lt_tid ← lt_tid + 1
        if d_i^prim > b.x.p then
            gt_tid ← gt_tid + 1

    lt, gt ← accum(lt), accum(gt)
    lbeg ← FAA(b^{x^s}, lt_T)
    gbeg ← FAA(b^{x^e}, −gt_T)
    lfrom_tid = lbeg + lt_tid
    gfrom_tid = gbeg + gt_tid
    i ← b^beg + tid

    for i < b^end, i ← i + T do
        if d_i^prim < pivot then
            d^aux_lfrom ← d_i^prim
            lfrom ← lfrom + 1
        if ld > pivot then
            d^aux_gfrom ← d_i^prim
            gfrom ← gfrom − 1

    if FAA(b^{x^c}, −1) = 1 then
        for i ← b^{x^s}, i < b^{x^e}, i ← i + 1 do
            d_i^aux ← b^{x^p}
```

the original subsequence will be assigned all available thread blocks. The subsequences are divided so that each thread block gets an equally large section to sort, as can be seen in Figure 1 (Step a and b).

First Pass When a thread block is executed on the GPU, it will iterate through all the data in its assigned sequence. Each thread in the block will keep track of the number of elements that are greater than the pivot and the number of elements that are smaller than the pivot. The data is read in chunks of T words, where T is the number of threads in each thread block. The threads read consecutive words so that the reads coalesce as much as possible.

Space Allocation Once we have gone through all the assigned data, we calculate the cumulative sum of the two arrays. We then use the atomic FAA-function to calculate the cumulative sum for all blocks that have completed so far. This information is used to give each thread a place to store its result, as can be seen in Figure 1 (Step c-f).

FAA is as of the time of writing not available on all GPUs. An alternative, if one wants to run the algorithm on the older, high-end graphics processors, is

Fig. 3. Pseudocode for the second phase.

```
procedure LQSORT(sl, d^true, d^false)

    wset = {sl_bid}

    while wset ≠ ∅ do
        v ← minsize(wset)
        where v = (v^s, v^e, v^b)
        pivot ← med(d^{v^b}_{v^s}, d^{v^b}_{v^e}, d^{v^b}_{(v^s+v^e)/2})
        i, lt_tid, gt_tid ← v^s + tid, 0, 0

        for i < v^e, i ← i + T do
            if d_i^{v^b} < pivot then
                lt_tid ← lt_tid + 1
            if d_i^{v^b} > pivot then
                gt_tid ← gt_tid + 1

        alt, agt ← accum(lt), accum(gt)
        alt_tid, agt_tid ← v^s + alt_tid, v^e − agt_tid
        i ← v^s + tid

    for i < v^e, i ← i + T do
        if d_i^{v^b} < pivot then
            d^{¬v^b}_{alt_tid}, alt_tid ← d_i^{v^b}, alt_tid − 1
        if d_i^{v^b} > pivot then
            d^{¬v^b}_{agt_tid}, agt_tid ← d_i^{v^b}, agt_tid + 1
    i ← v^s + alt_T + tid

    for i < v^s − agt_T, i ← i + T do
        d^false ← pivot

    r ← {(v^s, alt_T), (v^e − agt_T, agt_T)}

    for all s ∈ r do
        if s^len < MINSIZE then
            altsort(s^beg, s^len, d^{v^b}, d^false)
        else
            wset ← wset ∪ {(s^f, s^f + s^len, ¬v^b)}
```

to divide the kernel up into two kernels and do the block cumulative sum on the CPU instead. This would make the code more generic, but also slightly slower on new hardware.

Second Pass Using the cumulative sum, each thread knows where to write elements that are greater or smaller than the pivot. Each block goes through its assigned data again and writes it to the correct position in the current auxiliary array. It then fills the gap between the elements that are greater or smaller than the pivot with the pivot value. We now know that the pivot values are in their correct final position, so there is no need to sort them anymore. They are therefore not included in any of the newly created subsequences.

Are We Done? If the subsequences that arise from the partitioning are longer than *minlength*, they will be partitioned again in the next iteration, provided we don't already have more than *maxseq* subsequences. If we do have more than *maxseq* subsequences, the next phase begins. Otherwise we go through another iteration. (See Algorithm 1).

The Second Phase When we have acquired enough independent subsequences, there is no longer any need for synchronization between blocks. Because of this, the entire phase two can be run on the GPU entirely. There is however still the need for synchronization between threads, which means that we will use the same method as in phase one to partition the data. That is, we will count the number of elements that are greater or smaller than the pivot, do a cumulative sum so that each thread has its own location to write to and then move all elements to their correct position in the auxiliary buffer.

Stack To minimize the amount of fast local memory used, there is a very limited supply of it, we always recurse on the smallest subsequence. By doing that, Hoare have showed [16] that the maximum recursive depth can never go

below $\log_2(n)$. We use an explicit stack as suggested by Hoare and implemented by Sedgewick, always storing the smallest subsequence at the top [17].

Overhead When a subsequence's size goes below a certain threshold, we use an alternative sorting method on it. This was suggested by Hoare since the overhead of Quicksort gets too big when sorting small sequences of data. When a subsequence is small enough to be sorted entirely in the fast local memory, we could use any sorting method that can be made to sort in-place, doesn't require much expensive thread synchronization and performs well when the number of threads approaches the length of the sequence to be sorted.

Theorem 1. *The average time complexity for GPU-Quicksort is* $O(n \log(n))$.

Theorem 2. *The space complexity for GPU-Quicksort is* $2n + c$*, where* c *is a constant.*

The proofs of the theorems above are simple and are not included in this version of the paper due to space constraints.

4 Experimental Evaluation

We ran the experiments on a dual-processor dual-core AMD Opteron 1.8GHz machine. Two different graphics processors were used, the low-end NVIDIA 8600GTS 256MiB with 4 multiprocessors and the high-end NVIDIA 8800GTX 768MiB with 16 multiprocessors. Since the 8800GTX provides no support for the atomic FAA operation we instead used an implementation of the algorithm that exits to the CPU for block-synchronization.

We compared GPU-Quicksort to the following state-of-the-art GPU sorting algorithms:

GPUSort Uses bitonic merge sort [13].

Radix-Merge Uses radix sort to sort blocks that are then merged [18].

Global Radix Uses radix sort on the entire sequence [1].

Hybridsort Uses a bucket sort followed by a merge sort [15].

STL-Introsort This is the Introsort implementation found in the C++ Standard Library. Introsort is based on Quicksort, but switches to heap-sort when the recursion depth gets too large. Since it is highly dependent on the computer system and compiler used, we only included it to give a hint as to what could be gained by sorting on the GPU instead of on the CPU [19].

We could not find an implementation of the Quicksort algorithm used by Sengupta et al., but they claim in their paper that it took over 2 seconds to sort 4M uniformly distributed elements on a 8800GTX [1].

We only measured the actual sorting phase, we did not include in the result the time it took to setup the data structures and to transfer the data on and off the graphics memory. The reason for this is the different methods used to transfer data which wouldn't give a fair comparison between the GPU-based algorithms. Transfer times are also irrelevant if the data to be sorted are already available

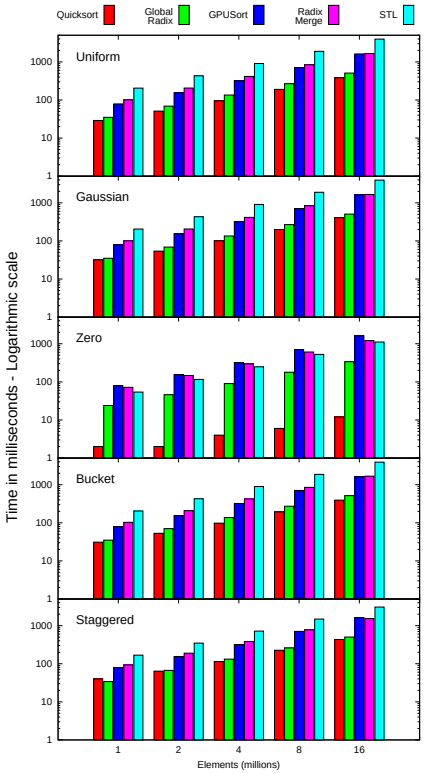

Fig. 4. Results on the 8800GTX.

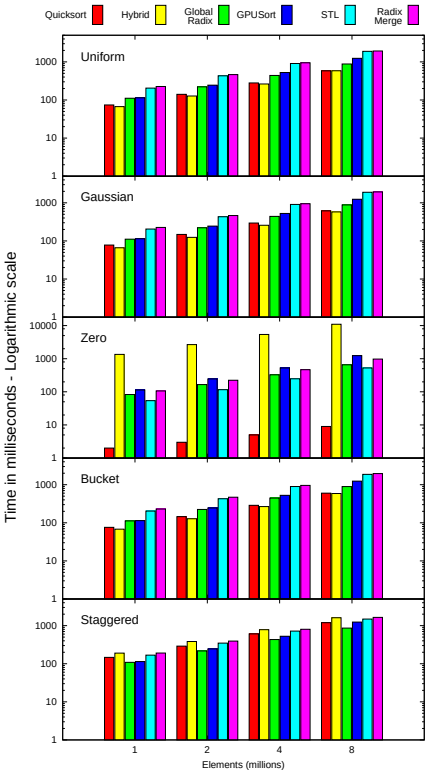

Fig. 5. Results on the 8600GTS.

on the GPU. Because of those reasons, this way of measuring has become a standard in the literature.

On the 8800GTX we used 256 thread blocks, each block having 256 threads. When a subsequence dropped below 1024 elements in size, we sorted it using bitonic sort. On the 8600GTS we lowered the amount of thread blocks to 128 since it has fewer multiprocessors. All implementations were compiled with the -O3 optimization flag.

We used different pivot selection schemes for the two phases. In the first phase we took the average of the minimum and maximum element in the sequence and in the second we picked the median of the first, middle and last element as the pivot, a method suggested by Singleton[20].

The source code of GPU-Quicksort is available for non-commercial use [21].

For benchmarking we used the following distributions which are defined and motivated in [22]. These are commonly used yardsticks to compare the performance of different sorting algorithms. The source of the random uniform values is the Mersenne Twister [23].

Uniform Values are picked randomly from $0 - 2^{31}$.

Zero A constant value is used. The actual value is picked at random.

Bucket The data set is divided into p blocks, where $p \in \mathbb{Z}^+$, which are then each divided into p sections. Section 1 in each block contains randomly selected values between 0 and $\frac{2^{31}}{p} - 1$. Section 2 contains values between $\frac{2^{31}}{p}$ and $\frac{2^{32}}{p} - 1$ and so on.

Gaussian The Gaussian distribution is created by always taking the average of four randomly picked values from the uniform distribution.

Staggered The data set is divided into p blocks, where $p \in \mathbb{Z}^+$. The staggered distribution is then created by assigning values for block i, where $i \leq \lfloor \frac{p}{2} \rfloor$, so that they all lie between $((2i-1)\frac{2^{31}}{p})$ and $((2i)(\frac{2^{31}}{p} - 1))$. For blocks where $i > \lfloor \frac{p}{2} \rfloor$, the values all lie between $((2i-p-2)\frac{2^{31}}{p})$ and $((2i-p-1)\frac{2^{31}}{p} - 1)$.

We decided to use a p value of 128. The results presented in Fig. 4 and 5 are based on experiments sorting sequences of integers. We have done experiments using floats instead, but found no difference in performance.

4.1 Discussion

Quicksort has a worst case scenario complexity of $O(n^2)$, but in practice, and on average when using a random pivot, it tends to be close to $O(n \log(n))$, which is the lower bound for comparison sorts. In all our experiments GPU-Quicksort has shown the best performance or been among the best. There was no distribution that caused problems to the perfomance of GPU-Quicksort. As can be seen when comparing the performance on the two GPUs, GPU-Quicksort shows a speedup of approximately 3 on the higher-end GPU. The higher-end GPU has a memory bandwidth that is 2.7 times higher and has four times the number of multiprocessors, indicating that the algorithm is bandwidth bound and not computation bound, which was the case with the Quicksort in [1].

On the CPU, Quicksort is normally seen as a faster algorithm as it can potentially pick better pivot points and doesn't need an extra check to determine when the sequence is fully sorted. The time complexity of radix sort is $O(n)$, but that hides a potentially high constant which is dependent on the key size. Optimizations are possible to lower this constant, such as constantly checking if the sequence has been sorted, but that can be expensive when dealing with longer keys. Quicksort being a comparison sort also means that it is easier to modify it to handle different key types.

The hybrid approach uses atomic instructions that were only available on the 8600GTS. We can see that it outperforms both GPU-Quicksort and the global radix sort on the uniform distribution. But it loses speed on the staggered distributions and becomes immensely slow on the zero distribution. The authors state that the algorithm drops in performance when faced with already sorted data, so they suggest randomizing the data first, but this wouldn't affect the result in the zero distribution.

GPUSort doesn't increase as much in performance as the other algorithms when executed on the higher-end GPU. This is an indication that the algorithm is more computationally bound than the other algorithms. It goes from being

much faster than the slow radix-merge to perform on par with and even a bit slower than it. The global radix sort showed a 3x speed improvement, as did GPU-Quicksort.

All algorithms showed about the same performance on the uniform, bucket and Gaussian distributions. GPUSort always shows the same result independent of distributions since it is a sorting network, which means it always performs the same number of operations regardless of the distribution. The staggered distribution was more interesting. On the low-end GPU the hybrid sorting was more than twice as slow as on the uniform distribution. GPU-Quicksort also dropped in speed and started to show the same performance as GPUSort. This can probably be attributed to the choice of pivot selection which was more optimized for uniform distributions. The zero distribution, which can be seen as an already sorted sequence, affected the algorithms to different extent. The STL reference implementation increased dramatically in performance since its two-way partitioning function always returned even partitions regardless of the pivot chosen. GPU-Quicksort shows the best performance as it does a three-way partitioning and can sort the sequence in $O(n)$ time.

5 Conclusions

In this paper we present GPU-Quicksort, a parallel Quicksort algorithm designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and inter-thread synchronization needed. A significant conclusion, we think, that can be drawn from this work, is that Quicksort is a practical alternative for sorting large quantities of data on graphics processors.

Acknowledgements We would like to thank Georgios Georgiadis, Marina Papatriantafilou and the anonymous referees for their valuable comments. We would also like to thank Ulf Assarsson and Erik Sintorn for insightful discussions regarding CUDA and for providing us with the source code to their hybrid sort.

References

1. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan Primitives for GPU Computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware. (2007) 97–106
2. Evans, D.J., Dunbar, R.C.: The Parallel Quicksort Algorithm Part 1 - Run Time Analysis. International Journal of Computer Mathematics **12** (1982) 19–55
3. Heidelberger, P., Norton, A., Robinson, J.T.: Parallel Quicksort Using Fetch-And-Add. IEEE Transactions on Computers **39**(1) (1990) 133–138
4. Tsigas, P., Zhang, Y.: A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. In: Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network-based Processing. (2003) 372–381

5. Purcell, T.J., Donner, C., Cammarano, M., Jensen, H.W., Hanrahan, P.: Photon Mapping on Programmable Graphics Hardware. In: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware. (2003) 41–50
6. Kapasi, U.J., Dally, W.J., Rixner, S., Mattson, P.R., Owens, J.D., Khailany, B.: Efficient Conditional Operations for Data-parallel Architectures. In: Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture. (2000) 159–170
7. Kipfer, P., Segal, M., Westermann, R.: UberFlow: A GPU-based Particle Engine. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. (2004) 115–122
8. Kipfer, P., Westermann, R.: Improved GPU Sorting. In Pharr, M., ed.: GPUGems 2. Addison-Wesley (2005) 733–746
9. Greß, A., Zachmann, G.: GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium. (2006)
10. Bilardi, G., Nicolau, A.: Adaptive Bitonic Sorting. An Optimal Parallel Algorithm for Shared Memory Machines. SIAM Journal on Computing **18**(2) (1989) 216–228
11. Govindaraju, N.K., Raghuvanshi, N., Manocha, D.: Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data. (2005) 611–622
12. Dowd, M., Perl, Y., Rudolph, L., Saks, M.: The Periodic Balanced Sorting Network. Journal of the ACM **36**(4) (1989) 738–757
13. 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)
14. Govindaraju, N.K., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. (2006) 325–336
15. Sintorn, E., Assarsson, U.: Fast Parallel GPU-Sorting Using a Hybrid Algorithm. In: Workshop on General Purpose Processing on Graphics Processing Units. (2007)
16. Hoare, C.A.R.: Quicksort. Computer Journal **5**(4) (1962) 10–15
17. Sedgewick, R.: Implementing Quicksort Programs. Communications of the ACM **21**(10) (1978) 847–857
18. Harris, M., Sengupta, S., Owens, J.D.: Parallel Prefix Sum (Scan) with CUDA. In Nguyen, H., ed.: GPU Gems 3. Addison Wesley (August 2007)
19. Musser, D.R.: Introspective Sorting and Selection Algorithms. Software - Practice and Experience **27**(8) (1997) 983–993
20. Singleton, R.C.: Algorithm 347: an Efficient Algorithm for Sorting with Minimal Storage. Communications of the ACM **12**(3) (1969) 185–186
21. Cederman, D., Tsigas, P.: GPU Quicksort Library. `www.cs.chalmers.se/~dcs/gpuqsortdcs.html` (December 2007)
22. Helman, D.R., Bader, D.A., JáJá, J.: A Randomized Parallel Sorting Algorithm with an Experimental Study. Journal of Parallel and Distributed Computing **52**(1) (1998) 1–23
23. Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. Transactions on Modeling and Computer Simulation **8**(1) (1998) 3–30