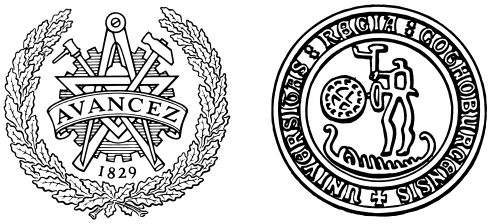
Technical Report no. 2008-01

A Practical Quicksort Algorithm for Graphics Processors

Daniel Cederman[*] *Philippas Tsigas*[†]

CHALMERS | GÖTEBORG UNIVERSITY



Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2008

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

Technical Report in Computer Science and Engineering at
Chalmers University of Technology and Göteborg University

Technical Report no. 2008-01
ISSN: 1650-3023

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2008

Abstract

In this paper we describe GPU-Quicksort, an efficient Quicksort algorithm suitable for highly parallel multi-core graphics processors. Quicksort has previously been considered an inefficient sorting solution for graphics processors, but we show that in CUDA, NVIDIA's programming platform for general purpose computations on graphical processors, GPU-Quicksort 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.

Keywords: Sorting, Multi-Core, CUDA, Quicksort, GPGPU

1 Introduction

In this paper, we describe an efficient parallel algorithmic implementation of Quicksort, GPU-Quicksort, designed to take advantage of the highly parallel nature of the graphics cards (GPUs) and their limited cache memory. Quicksort has long been considered one of the fastest sorting algorithms in practice for single processor systems and is also one of the most studied sorting algorithms, but until now it has not been considered an efficient sorting solution for GPUs [23]. 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 i) using a two-phase design to keep the inter-thread synchronization low, ii) coalescing read operations and constraining threads so that memory accesses are kept to a minimum. It can also take advantage of the atomic synchronization primitives found on newer hardware to, when available, further improve its performance.

Today's graphics cards contain very powerful multi-core processors, for example, NVIDIA's highest-end graphics processor currently boasts 128 cores. Outside of gaming this computational power goes mostly unused. But since the processors are specialized for compute-intensive, highly parallel computations, they could be used to assist the CPU in solving problems that can be efficiently data-parallelized.

Previous work on general purpose computation on GPUs have used the OpenGL interface, which is a rather clunky way, being primarily designed for performing graphics operations and gives a poor abstraction to the programmer that wishes to use it for non-graphics related tasks. NVIDIA is attempting to remedy this situation by providing programmers with CUDA[1], a programming platform for doing general purpose computation on GPUs. ATI has similar project called *Close to the Metal*.

Although simplifying programming a lot, one still needs to be aware of the strengths and limitations of the new platform to be able to take full advantage of it. Algorithms that work great on standard single processor systems most likely need to be altered extensively to perform well on GPUs, which have limited cache memory and instead uses massive parallelism to hide memory latency.

This means that directly porting efficient sorting algorithms from the single processor domain to the GPU domain would most likely yield very poor performance. This is unfortunate, since the sorting problem is very well suited to be solved in parallel and is an important kernel for sequential and multiprocessing computing and a core part of database systems. Being one of the most basic computing problems, it also plays a vital role in plenty of algorithms commonly used in graphics applications, such as visibility ordering or collision detection.

Quicksort was presented by C.A.R. Hoare in 1961 and uses a divide-and-conquer method to sort data [13]. A sequence is sorted by recursively dividing it into two subsequences, one with values lower and one with values higher than specific pivot value that is selected in each iteration. This is done until all elements are sorted.

[1] Compute Unified Device Architecture

1.1 Related Work

With Quicksort being such a popular sorting algorithm, there have been a lot of different attempts to create an efficient parallelization of it. The obvious way is to take advantage of its inherent parallelism by just assigning a new processor to each new subsequence. This means, however, that there will be very little parallelization in the beginning, when the sequences are few and large [5].

Another approach has been to divide each sequence to be sorted into blocks that can then be dynamically assigned to available processors [11, 26]. However, this method requires extensive use of atomic FAA[2] which makes it too expensive to use on graphics processors.

Blelloch suggested using prefix sums to implement Quicksort and recently Sengupta et al. used this method to make an implementation for CUDA [2, 23]. The implementation was done as a demonstration of their segmented scan primitive, but it performed quite poorly and was an order of magnitude slower than their radix-sort implementation in the same paper.

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. [21] have presented an implementation of bitonic merge sort on GPUs based on an implementation by Kapasi et al. [15]. Kipfer et al. [16, 17] 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 found in the Bilardi et al. paper [1]. Govindaraju et al. [8] implemented a sorting solution based on the periodic balanced sorting network method by Dowd et al. [4] and one based on bitonic sort [6]. 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 [7]. Sengupta et al. [23] have presented a radix-sort and a Quicksort implementation. Recently, Sintorn et al. [25] presented a hybrid sorting algorithm which splits the data with a bucket sort and then uses merge sort on the resulting blocks. The implementation requires atomic primitives that are currently not available on all cards.

In the following section we present the system model. In Section 3.1 we give an overview of the algorithm and in Section 3.2 we go through it in detail. We prove its time and space complexity in Section 4. In Section 5 we show the results of our experiments and in Section 5.4 and Section 6 we discuss the result and conclude.

2 The System Model

CUDA is NVIDIA's initiative to bring general purpose computation to 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. Also included are high performance numerical libraries for FFT and linear algebra.

General Architecture The high range graphics cards 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, has 16KiB of fast local memory and a maximum of 8192 available registers that can be divided between the threads.

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

[2]Fetch-And-Add reads an integer from the memory, increments it by a given amount and writes it back to the memory, all in one atomic step.

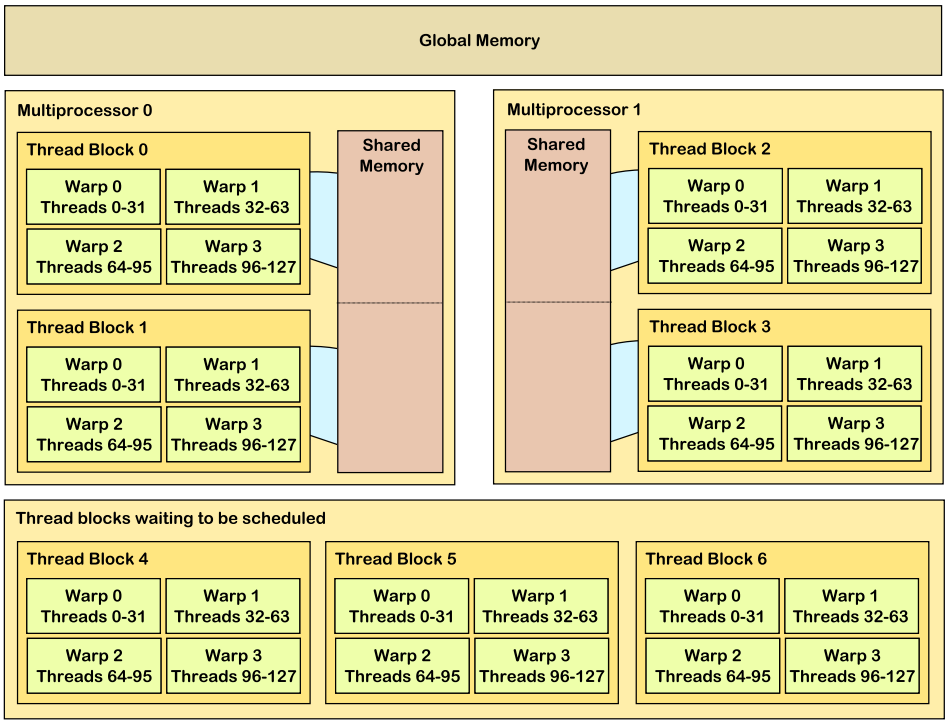


Figure 1: A graphical representation of the CUDA hardware model

threads requires, there could be multiple blocks assigned to a single multiprocessor. If more blocks are needed than there is room for on any of the multiprocessors, the leftover blocks will be run sequentially.

The GPU schedules threads depending on which *warp* they are in. Threads with *id* 0..31 are assigned to the first warp, threads with *id* 32..63 to the next and so on. When a warp is scheduled for execution, the threads which perform the same instructions are executed concurrently (limited by the size of the multiprocessor) whereas threads that deviate are executed sequentially. Hence it's important to try to make all threads in the same warp perform the same instructions most of the time. See Figure 1 for a graphical description of the way threads are grouped together and scheduled.

Two warps cannot execute simultaneously on a single multiprocessor, so one could see the warp as the counter-part of the thread in a conventional SMP system. All instructions on the GPU are SIMD, so the threads that constitute a warp can be seen as a way to simplify the usage of these instructions. Instead of each thread issuing SIMD instructions on 32-word arrays, the threads are divided into 32 sub-threads that each works on its own word.

Synchronization Threads within a thread block can use the multiprocessors shared local memory and a special thread barrier-function to communicate with each other. The hardware thread barrier-function is possible since the thread scheduling is done in hardware and not in software, as is usually the case on conventional SMP systems.

There is however no barrier-function for threads from different blocks. The reason for this is that when more blocks are executed than there is room for on the multiprocessors, the scheduler will wait for a thread block to finish executing before it swaps in a new block. This makes it impossible to implement a barrier function in software and the only solution is to

wait until all blocks have completed.

Some newer cards support atomic instructions such as CAS (Compare-And-Swap) and FAA.

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

This is in direct contrast with the conventional SMP systems, where one should try to let each thread access its own part of the memory so as to not thrash the cache.

Because of the lack of caching, a high number of threads are needed to hide the memory latency. These threads should preferably have a high ratio of arithmetic to memory operations to be able to hide the latency well.

The shared memory is divided into memory banks that can be accessed in parallel. If two threads write to or read from the same memory bank, the accesses will be serialized. Due to this, one should always try make threads in the same warp write to different banks. If all threads read from the same memory bank, a broadcasting mechanism will be used, making it just as fast as a single read. A normal access to the shared memory takes the same amount of time as accessing a register.

3 The algorithm

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

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. 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 needs 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 cards and there is still the need to have a thread block barrier-function between the partition iterations, something that cannot be implemented using the available atomic primitives.

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 run. So the only way to synchronize thread blocks is to wait until all blocks have finished executing. Then one can assign new sequences to them.

Exiting and reentering the GPU is not expensive, but it's also not delay-free since parameters needs 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, 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 [14] 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.

In-place On conventional SMP systems it's favorable to perform the sorting in-place, since that gives good cache behavior. But on the GPUs with 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 fade. Here it's 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's 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 and the primary becomes the auxiliary and vice versa.

3.1.1 Partitioning

The principle of two phase partitioning is outlined in Figure 2. A sequence to be partitioned is selected and it's 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 to the right of the pivot. The problem here is how to synchronize this in an efficient way. How do we make sure that each thread knows where to write in the auxiliary buffer? It should also be noted that it's important to minimize the amount of synchronization communication between threads since it will be quite expensive as we have so many threads.

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[3] of the number of threads that want to write to the left and that wants to write to the right of the pivot, each thread would know that x threads with a lower thread id than its own are going

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

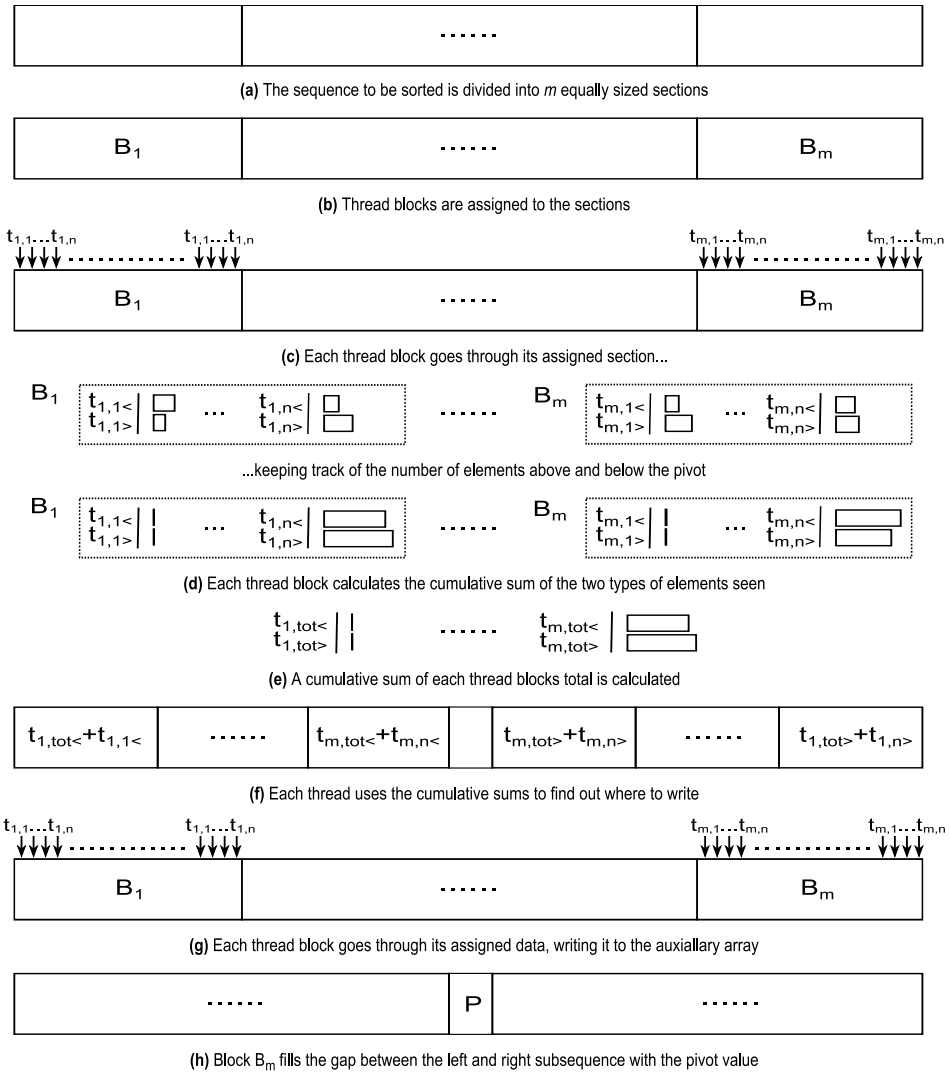


Figure 2: Partitioning a sequence

to write to the left and that y threads are going to write to the right. 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, were 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).

Now 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.

Algorithm 1 Parallel Quicksort

procedure PQSORT($size, d^{prim}, d^{aux}$)
$minlength, flip \leftarrow \frac{size}{maxseq}, false$
$work, done \leftarrow \{(0, size, flip, piv)\}, \emptyset$
while $work \neq \emptyset \wedge |work| + |done| < maxseq$ **do**
  $ws, xs \leftarrow \sum_{v \in work} \frac{v_{end} - v_{start}}{maxseq}, \emptyset$
  for all $v \in work$ **do**
    $x \leftarrow (v_{start}, v_{end}, v, \lceil \frac{v_{end} - v_{start}}{ws} \rceil)$
    $xs \cup \{x\}$
    for $i \leftarrow 0, i < x^c - 1, i \leftarrow i + 1$ **do**
      $start \leftarrow x^s + ws \cdot i$
      $bl \leftarrow bl \cup \{(x, start, start + ws)\}$
    $bl \leftarrow bl \cup \{(x, x^s + ws \cdot (x^c - 1), x^e)\}$
  gqsort(bl, d^{prim}, d^{aux});
  for all $x \in xs$ **do**
    if $x^s - x^{v_{start}} < minlength$ **then**
      $done \leftarrow done \cup \{(x^{v_{start}}, x^s, flip, piv)\}$
    else
      $work \leftarrow work \cup \{(x^{v_{start}}, x^s, flip, piv)\}$
    if $x^{v_{end}} - x^e < minlength$ **then**
      $done \leftarrow done \cup \{(x^e, x^{v_{end}}, flip, piv)\}$
    else
      $work \leftarrow work \cup \{(x^e, x^{v_{end}}, flip, piv)\}$
  $d^{prim}, d^{aux}, flip \leftarrow d^{aux}, d^{prim}, \neg flip$
if $flip$ **then**
  $d^{prim}, d^{aux} \leftarrow d^{aux}, d^{prim}$
$done \leftarrow done \cup work$
lqsort($done, d^{prim}, d^{aux}$);

3.2 Detailed Description

The following subsection describes the algorithm in more detail.

3.2.1 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's better to have some extra sequences and let the scheduler balance the workload. Based on that observation, a good way to partition is to only partition

Algorithm 2 First Phase Kernel

procedure GQSORT(bl, d^{prim}, d^{aux})
$b \leftarrow bl_{bid}$
$lt_{tid}, gt_{tid} \leftarrow 0, 0$
$i \leftarrow b^{start} + tid$
for $i < b^{end}, i \leftarrow i + T$ **do**
  if $d_i^{prim} < b^{x^p}$ **then**
    $lt_{tid} \leftarrow lt_{tid} + 1$
  if $d_i^{prim} > b.x.p$ **then**
    $gt_{tid} \leftarrow gt_{tid} + 1$
$lt, gt \leftarrow accum(lt), accum(gt)$
$lstart \leftarrow FAA(b^{x^s}, lt_T)$
$gstart \leftarrow FAA(b^{x^e}, -gt_T)$
$lfrom_{tid} = lstart + lt_{tid}$
$gfrom_{tid} = gstart + gt_{tid}$
$i \leftarrow b^{start} + tid$
for $i < b^{end}, i \leftarrow i + T$ **do**
  if $d_i^{prim} < pivot$ **then**
    $d_{lfrom}^{aux} \leftarrow d_i^{prim}$
    $lfrom \leftarrow lfrom + 1$
  if $ld > pivot$ **then**
    $d_{gfrom}^{aux} \leftarrow d_i^{prim}$
    $gfrom \leftarrow gfrom - 1$
if $FAA(b^{x^c}, -1) = 1$ **then**
  for $i \leftarrow b^{x^s}, i < b^{x^e}, i \leftarrow i + 1$ **do**
    $d_i^{aux} \leftarrow b^{x^p}$

subsequences that are longer than $minlength = n/maxseq$ and to stop when we have $maxseq$ number of sequences.

In the beginning of each iteration, all sequences that are larger than $minlength$ are assigned thread blocks relative to their size. In the first iteration, the original sequence will be assigned all available thread blocks. The sequences are divided so that each thread block gets an equally large section to sort, as can be seen in Figure 2 (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. This information is stored in two arrays in the shared local memory; with each thread in a half warp[4] accessing different memory banks to increase performance.

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

[4]A warp is 32 consecutive threads that are always scheduled together.

Algorithm 3 Second Phase Kernel

procedure LQSORT(sl, d^{true}, d^{false})
$wset = \{sl_{bid}\}$
while $wset \neq \emptyset$ **do**
  $v \leftarrow minsize(wset)$ **where** $v = (v^s, v^e, v^b)$
  $pivot \leftarrow median(d^{v^b}_{v^s}, d^{v^b}_{v^e}, d^{v^b}_{(v^s+v^e)/2})$
  $i, lt_{tid}, gt_{tid} \leftarrow v^s + tid, 0, 0$
  for $i < v^e, i \leftarrow i + T$ **do**
    if $d^{v^b}_i < pivot$ **then**
      $lt_{tid} \leftarrow lt_{tid} + 1$
    if $d^{v^b}_i > pivot$ **then**
      $gt_{tid} \leftarrow gt_{tid} + 1$
  $alt, agt \leftarrow accum(lt), accum(gt)$
  $alt_{tid}, agt_{tid} \leftarrow v^s + alt_{tid}, v^e - agt_{tid}$
  $i \leftarrow v^s + tid$
  for $i < v^e, i \leftarrow i + T$ **do**
    if $d^{v^b}_i < pivot$ **then**
      $d^{\neg v^b}_{alt_{tid}}, alt_{tid} \leftarrow d^{v^b}_i, alt_{tid} - 1$
    if $d^{v^b}_i > pivot$ **then**
      $d^{\neg v^b}_{agt_{tid}}, agt_{tid} \leftarrow d^{v^b}_i, agt_{tid} + 1$
  $i \leftarrow v^s + alt_T + tid$
  for $i < v^s - agt_T, i \leftarrow i + T$ **do**
    $d^{false} \leftarrow pivot$
  $r \leftarrow \{(v^s, alt_T), (v^e - agt_T, agt_T)\}$
  for all $s \in r$ **do**
    if $s^{size} < MINSIZE$ **then**
      $altsort(s^{start}, s^{size}, d^{v^b}, d^{false})$
    else
      $wset \leftarrow wset \cup \{(s^f, s^f + s^{size}, \neg v^b)\}$

each thread a place to store its result, as can be seen in Figure 2 (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 cards, is 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* sequences. If we do, the next phase begins. Otherwise we go through

another iteration. (See Algorithm 1 and 2).

3.2.2 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. (See Algorithm 3).

Stack To minimize the amount of fast local memory used, there being a very limited supply of it, we always recurse on the smallest subsequence. By doing that, Hoare have showed [14] 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 in [22], always storing the smallest subsequence at the top.

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. See Section 5.2 for more information about algorithm used.

3.2.3 Cumulative sum

When calculating the cumulative sum, it would be possible to use a simple sequential implementation, since the sequences are so short (≤ 512). But it's calculated so often that every performance increase counts, so we decided to use the parallel cumulative sum implementation described in [10] which is based on [2]. Their implementation was an *exclusive* cumulative sum so we had to modify it to include the total sum. We also modified it so that it accumulated two arrays at the same time. By using this method, the speed of the calculation of the cumulative sum was increased by 20% compared to using a sequential implementation.

Another alternative would have been to let each thread use FAA to create a cumulative sum, but that would have been way too expensive, since all the threads would have been writing to the same variable, leading to all additions being serialized. Measurements done using 128 threads show that it would be more than ten times slower than the method we decided to use.

4 Complexity

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

Proof. For the analysis we combine phase one and two since there is no difference between them from a complexity perspective. Each partition iteration requires going through the data, calculating the cumulative sum and going through the data again writing the result to its correct position. Going through the data twice takes $O(\frac{2n}{TB})$ steps, where T is the number of threads per thread block and B is the number of blocks.

According to [2] the accumulate function has a time complexity of $O(\frac{n_{acc}}{T} + \log(T))$. Since we only calculate the cumulative sum on arrays the size of T, we can simplify it to $O(\log(T))$.

Assuming that all elements are equally likely to be picked as a pivot we get an average running time of

$$T(n) = \begin{cases} O(\frac{2n}{TB} + \log(T)) + \frac{2}{n}\sum_{i=0}^{n-1} T(i) & n > m, \\ T_{altsort}(n) & n \leq m. \end{cases}$$

where m is the maximum size of the sequences that can be sorted by the alternative sorting method. The value of m is constant and is limited by the maximum amount of available shared memory on the card, which means that the worst case complexity of the alternative sort is not dependent on n. The alternative sort can thus be seen as having complexity $O(1)$.

This together with the fact that both T and B are constants independent of n gives us

$$T(n) = \begin{cases} O(n) + \frac{2}{n}\sum_{i=0}^{n-1} T(i) & n > m, \\ O(1) & n \leq m. \end{cases}$$

Assuming that $m \ll n$ we can set $m = 1$ which gives us the standard Quicksort recurrence relation, which is proved to be $O(n \log(n))$ in e.g. [18]. □

THEOREM 2. *The space complexity for GPU-Quicksort is $O(5B + 2n)$, where B is the amount of thread blocks used.*

Proof. The first phase needs more global memory than the second phase, since it has to keep track on which thread blocks are accessing the same sequence, something which is not needed in the second phase where all thread blocks have their own sequence.

To keep track of which part of the sequence each thread block is partitioning, two words are required for every block (start of sequence and end of sequence). Each block also needs to know the index to the tuple containing the start and end of the shared sequence. This index takes up one word and there is a maximum of B tuples each being two words in size (start and end of sequence).

If all these values are summed up, it gives us a space complexity of $O(5B + 2n)$ for GPU-Quicksort. □

5 Experimental Evaluation

5.1 Hardware

We ran the experiments on a dual-processor dual-core AMD Opteron 1.8GHz machine. Two different graphics cards were used, the low-end 8600GTS 256MiB NVIDIA graphics card with 4 multiprocessors and the high-end 8800GTX 768MiB NVIDIA graphics card with 16 multiprocessors, each multiprocessor having 8 processors each.

The 8800GTX provides no support for atomic operations. Because of this, we used an implementation of the algorithm that exits to the CPU for block-synchronization, instead of using FAA.

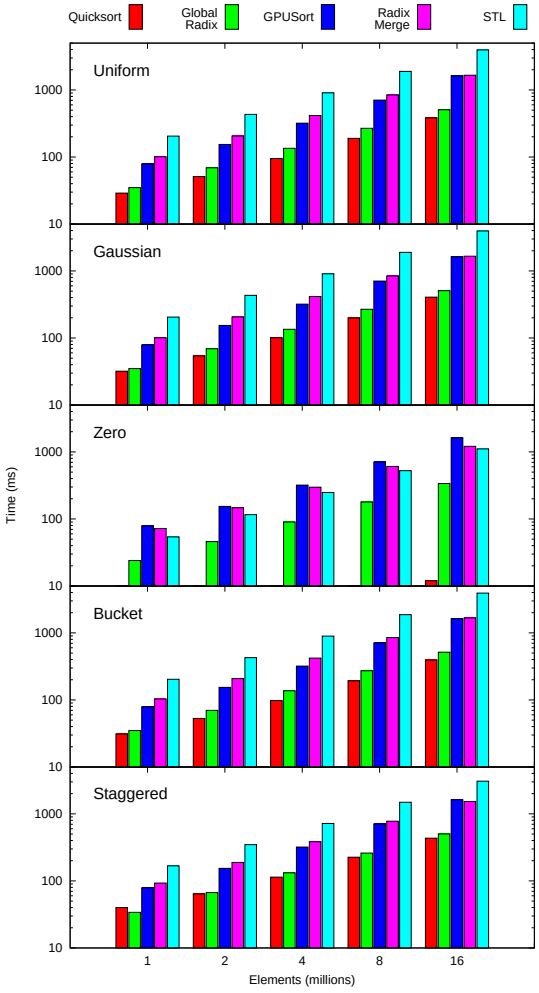


Figure 3: Results on the 8800GTX

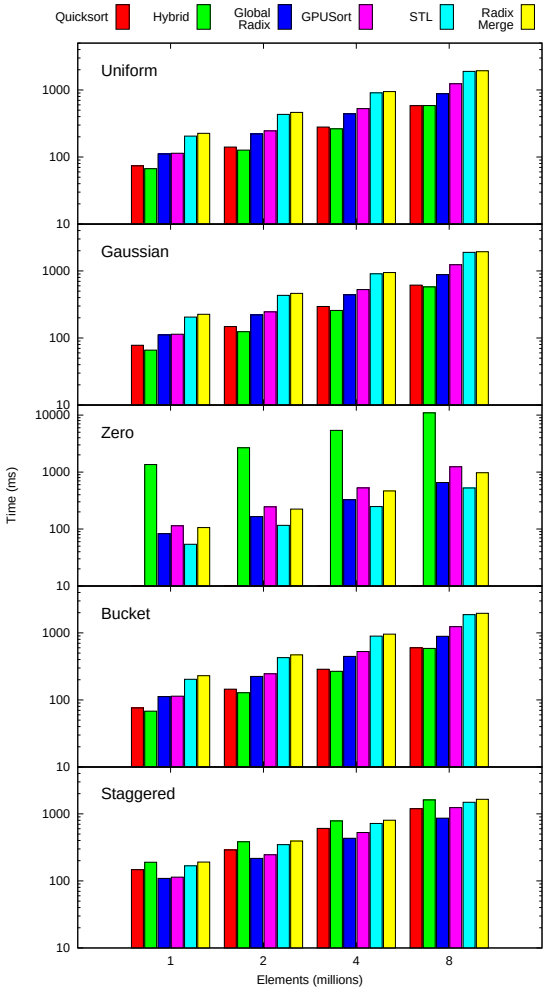


Figure 4: Results on the 8600GTS

5.2 Algorithms used

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

GPUSort Uses bitonic merge sort. By Govindaraju et. al. [6].

Radix-Merge Uses radix sort to sort blocks that are then merged. By Harris et. al. [10].

Global Radix Uses radix sort on the entire sequence. By Sengupta et. al. [23].

Hybridsort Splits the data with a bucket sort and uses merge sort on the resulting blocks. By Sintorn and Assarsson [25].

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 subsequences get smaller than a certain value. Since it's 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 [20].

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 an 8800GTX, which makes it much slower than STL sort [23].

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 card. 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 is already available 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 sequence got 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. We used this more computationally expensive method here because it's vital to have an even size of the subsequences that are assigned to each multiprocessor in the next phase.

In phase two it is not as important to have evenly sized partitions since everything is being run locally on the GPU, but with better pivot selection we achieve better performance. Keeping track of the maximum and minimum elements became too expensive in this phase, so instead we picked the median of the first, middle and last element as the pivot, a method suggested by Singleton [24]. This gives a good pivot even when faced with totally sorted data.

The full source code of GPU-Quicksort can be downloaded for free for non-commercial use [3].

5.3 Input Distributions

For benchmarking we used the following distributions which are defined and motivated in [12]. The source of the random uniform values is the Mersenne Twister [19].

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 reason for this is that when we started doing experiments we used 128 thread blocks and later on we could not detect any performance difference for any algorithm when changing this value.

The results presented in Figure 3 and 4 are based on experiments sorting sequences of integers. We have done experiments using floats instead, but found no difference in performance.

5.4 Discussion

In this section we discuss GPU sorting in the light of the experimental result.

Since sorting on GPUs has received a lot of attention it might be reasonable to start with the following question; **is there really a point in sorting on the GPU?** If we take a look at the radix-merge sort in Figure 4 we see that it performs comparable to the CPU reference implementation. Considering that we can run the algorithm concurrently with other operations on the CPU, it makes perfect sense to sort on the GPU.

If we look at the other algorithms we see that they perform at twice the speed and more compared to Introsort, the CPU reference. On the faster GPU in Figure 3, the difference in speed can be up to 8 times the speed of the reference! Even if one includes the time it takes to transfer data back and forth to the GPU, less than 8ms per 1M element, it is still a massive performance gain that can be made by sorting on the GPU. Clearly there are good reasons to use the GPU as a general purpose co-processor.

But why should one use Quicksort? 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. As can be seen when comparing the performance on the two GPUs, GPU-Quicksort shows a speedup by around 3 times on the faster GPU. The faster GPU has a memory bandwidth that is 2.7 times higher but has a slightly slower clock speed, indicating that the algorithm is bandwidth bound and not computation bound, which was the case with the Quicksort in [23].

Is it better than radix? 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 $O(32n)$, which is higher than $O(n \log(n))$ for $n < 2^{32}$. Optimizations are possible to lower this constant, such as constantly checking if the sequence has been sorted, but when dealing with longer keys that can be expensive. Quicksort being a comparison sort also means that it is easier to modify it to handle different key types.

Is the hybrid approach better? 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. In the paper by Sintorn and Assarsson they state that the algorithm drops in performance when faced with already sorted data, so they suggest randomizing the data first. This however lowers the performance and wouldn't affect the result in the zero distribution.

This hybrid approach is useful when one knows that the distribution will be uniform and not partially sorted, but Quicksort is more general and is thus more practical when the distribution is unknown.

How are the algorithms affected by the faster card? GPUSort doesn't increase as much in performance as the other algorithms when run on the faster card. 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 and even a bit slower than it.

The global radix sort showed a 3x speed improvement, as did GPU-Quicksort. As mentioned earlier, this shows that the algorithms most likely are bandwidth bound.

How are the algorithms affected by the different distributions? 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 slower 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 it always got even partitions regardless of the pivot chosen and never had to swap elements positions. GPUSort performs the same number of operations regardless of the distribution, so there was no change there. The hybrid sort didn't like this distribution at all and showed terrible performance. The reason for this is that all elements ended up in the same bucket. GPU-Quicksort shows the best performance since it will pick the only value that is available in the distribution as the pivot value, which will then be marked as already sorted. This means that it just have to do two passes through the data and can sort the zero distribution in $O(n)$ time.

6 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.

The bookkeeping is minimized by constraining all thread blocks to work with only one (or part of a) sequence of data at a time. This way pivot values do not need to be distributed to all thread blocks and thus no extra information needs to be written to the global memory.

The two-pass design of GPU-Quicksort has been introduced to keep the inter-thread synchronization low. First the algorithm goes through the sequence to sort, counting the number of elements that each thread sees that have a higher (or lower) value than the pivot. By calculating a cumulative sum of these sums, in the second phase, each thread will know where to write its assigned elements without any extra synchronization. The small amount of inter-block synchronization that is required between the two passes of the algorithm can be reduced further by taking advantage of the atomic synchronization primitives that are available on newer hardware.

A previous implementation of Quicksort for GPUs by Sengupta et. al. turned out not to be competitive enough in comparison to radix sort or even CPU based sorting algorithms [23]. According to the authors this was due to it being more dependent on the processor speed than on the bandwidth. This is in contrast with GPU-Quicksort which is bandwidth bound, which means that it gains significantly from the high bandwidth of GPUs and scales in performance as bandwidth increases.

In experiments we compared GPU-Quicksort with some of the fastest known sorting algorithms for GPUs, as well as with the C++ Standard Library sorting algorithm, Introsort, for reference. We used several input distributions and two different graphics processors, the low-end 8600GTS with 32 cores and the high-end 8800GTX with 128 cores, both from NVIDIA. What we could observe was that GPU-Quicksort performed better on all distributions on the high-end processor and on par with or better on the low-end processor.

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 and Marina Papatriantafilou for their valuable comments during the writing of this report. 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] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting. An optimal parallel algorithm for shared-memory machines. *SIAM J Comput*, 18(2):216–228, 1989.

[2] Guy E. Blelloch. Prefix Sums and Their Applications. In John H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[3] Daniel Cederman and Philippas Tsigas. GPU Quicksort Library source code. *www.cs.chalmers.se/~dcs/gpuqsortdcs.html*, 2007.

[4] Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. The periodic balanced sorting network. *J. ACM*, 36(4):738–757, 1989.

[5] D. J. Evans and R. C. Dunbar. The parallel Quicksort algorithm Part 1 - Run time analysis. *Int. J. Comput. Math.*, 12:19–55, 1982.

[6] N. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina-Chapel Hill, 2005.

[7] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In *ACM SIGMOD International Conference on Management of Data*, Chicago, United States, June 2006.

[8] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622, New York, NY, USA, 2005. ACM Press.

[9] Alexander Greß and Gabriel Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In *Proc. 20th IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 2006.

[10] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[11] P. Heidelberger, A. Norton, and John T. Robinson. Parallel Quicksort Using Fetch-And-Add. *IEEE Trans. Comput.*, 39(1):133–138, 1990.

[12] David R. Helman, David A. Bader, and Joseph JáJá. A randomized parallel sorting algorithm with an experimental study. *J. Parallel Distrib. Comput.*, 52(1):1–23, 1998.

[13] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.

[14] C. A. R. Hoare. Quicksort. *Computer J.*, 5(4):10–15, 1962.

[15] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 159–170, New York, NY, USA, 2000. ACM Press.

[16] Peter Kipfer, Mark Segal, and Rüdiger Westermann. UberFlow: a GPU-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[17] Peter Kipfer and Rüdiger Westermann. Improved GPU Sorting. In Matt Pharr, editor, *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 733–746. Addison-Wesley, 2005.

[18] Udi Manber. *Introduction to Algorithms - A Creative Approach.* Addison-Wesley, 1989.

[19] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[20] David R. Musser. Introspective Sorting and Selection Algorithms. *Software - Practice and Experience*, 27(8):983–993, 1997.

[21] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[22] Robert Sedgewick. Implementing Quicksort programs. *Commun. ACM*, 21(10):847–857, 1978.

[23] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John 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. Eurographics Association.

[24] Richard C. Singleton. Algorithm 347: an efficient algorithm for sorting with minimal storage. *Commun. ACM*, 12(3):185–186, 1969.

[25] Erik Sintorn and Ulf Assarsson. Fast Parallel GPU-Sorting Using a Hybrid Algorithm. In *Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[26] Philippas Tsigas and Yi Zhang. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, page 372, 2003.