GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors

DANIEL CEDERMAN and PHILIPPAS TSIGAS
Chalmers University of Technology

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.

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Sorting and searching; G.4 [**MATHEMATICAL SOFTWARE**]: Parallel and vector implementations

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: 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 graphics processors (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 [Sengupta et al. 2007]. 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-pass design to keep the inter-thread synchronization low, ii) coalescing read operations and constraining threads so that memory accesses are kept

Daniel Cederman was supported by Microsoft Research through its European PhD Scholarship Programme and Philippas Tsigas was partially supported by the Swedish Research Council (VR).
Authors' addresses: Daniel Cederman and Philippas Tsigas, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden; email: {cederman,tsigas}@chalmers.se.

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 processors contain very powerful multi-core processors, for example, NVIDIA's highest-end graphics processor currently boasts 128 cores. These processors are specialized for compute-intensive, highly parallel computations and 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, but since it was primarily designed for performing graphics operations it gives a poor abstraction to the programmer who wishes to use it for non-graphics related tasks. NVIDIA is attempting to remedy this situation by providing programmers with CUDA, a programming platform for doing general purpose computation on their GPUs. A similar initiative to CUDA is OpenCL, which specification has just recently been released [Khronos Group 2008].

Although CUDA simplifies the programming, 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 use 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 [Hoare 1961]. A sequence is sorted by recursively dividing it into two subsequences, one with values lower and one with values higher than the specific pivot value that is selected in each iteration. This is done until all elements are sorted.

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 [Evans and Dunbar 1982].

Another approach has been to divide each sequence to be sorted into blocks that can then be dynamically assigned to available processors [Heidelberger et al. 1990; Tsigas and Zhang 2003]. However, this method requires extensive use of atomic synchronization instructions 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 [Blelloch 1993; Sengupta et al. 2007]. 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. [Purcell et al. 2003] have presented an implementation of bitonic merge sort on GPUs based on an implementation by Kapasi et al. [Kapasi et al. 2000]. Kipfer et al. [Kipfer et al. 2004; Kipfer and Westermann 2005] have shown an improved version of the bitonic sort as well as an odd-even merge sort. Greß et al. [Greß and Zachmann 2006] introduced an approach based on the adaptive bitonic sorting technique found in the Bilardi et al. paper [Bilardi and Nicolau 1989]. Govindaraju et al. [Govindaraju et al. 2005] implemented a sorting solution based on the periodic balanced sorting network method by Dowd et al. [Dowd et al. 1989] and one based on bitonic sort [Govindaraju et al. 2005]. 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 [Govindaraju et al. 2006]. Sengupta et al. [Sengupta et al. 2007] have presented a radix-sort and a Quicksort implementation. Recently, Sintorn et al. [Sintorn and Assarsson 2007] 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 graphics processors.

In the following section, Section 2, 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 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, has 16KiB of fast local memory called the shared memory and have 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 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 is important to

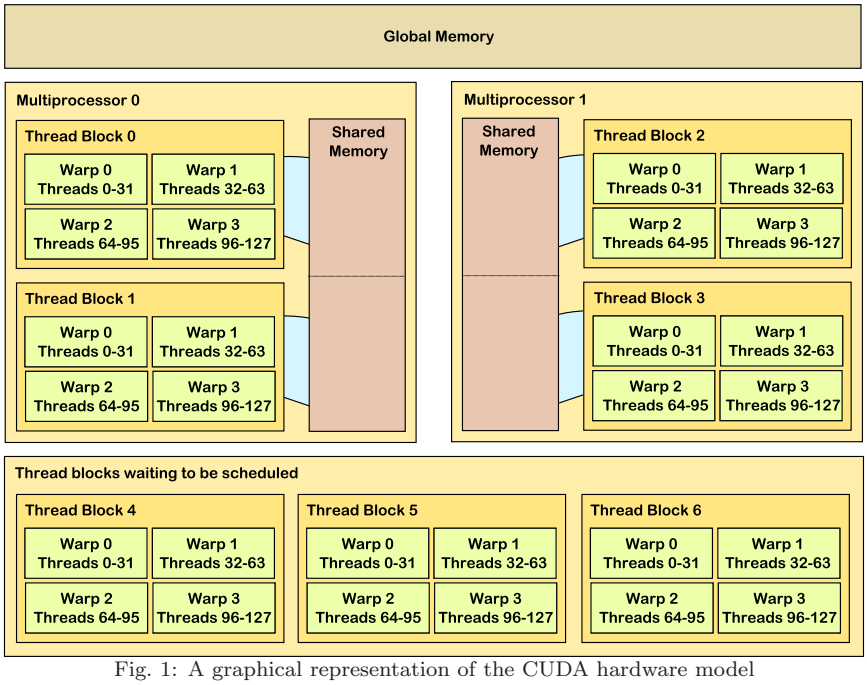


Fig. 1: A graphical representation of the CUDA hardware model

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 barrier-function forces all threads in the same block to synchronize, that is, a thread calling it will not be allowed to continue until all other threads have also called it. They will then continue from the same position in the code. 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.

Newer graphics processors support atomic instructions such as CAS (Compare-And-Swap) and FAA (Fetch-And-Add). See Figure 2 for the specification of these synchronization operations.

Memory Data is stored in a large global memory that supports both gather and scatter operations. There is no caching available automatically when accessing this

```
function CAS(ptr, oldval, newval)
    if *ptr = oldval then
        *ptr ← newval;
        return oldval;
    return *ptr;

function FAA(ptr, value)
    *ptr ← *ptr + value;
    return *ptr − value;
```

Fig. 2: The specification for the Compare-And-Swap and Fetch-And-Add operation. The two operations are performed atomically.

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, the hardware can coalesce several read or write operations into one big read or write operation, which increases 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

As in the original Quicksort algorithm, the method used 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. In our experiments we use a deterministic pivot selection that is described in Section 5, but a randomized method could also be used. 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 graphics

processors 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 is 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 [Hoare 1962] 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. For the experiments we decided to use bitonic sort, but other sorting algorithms could also be used.

In-place On conventional SMP systems it is 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 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 and the primary becomes the auxiliary and vice versa.

3.1.1 *Partitioning.* The principle of two phase partitioning is outlined in Figure 3. A 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 to the right of the pivot. The problem here is how to synchronize this in an efficient way. How do

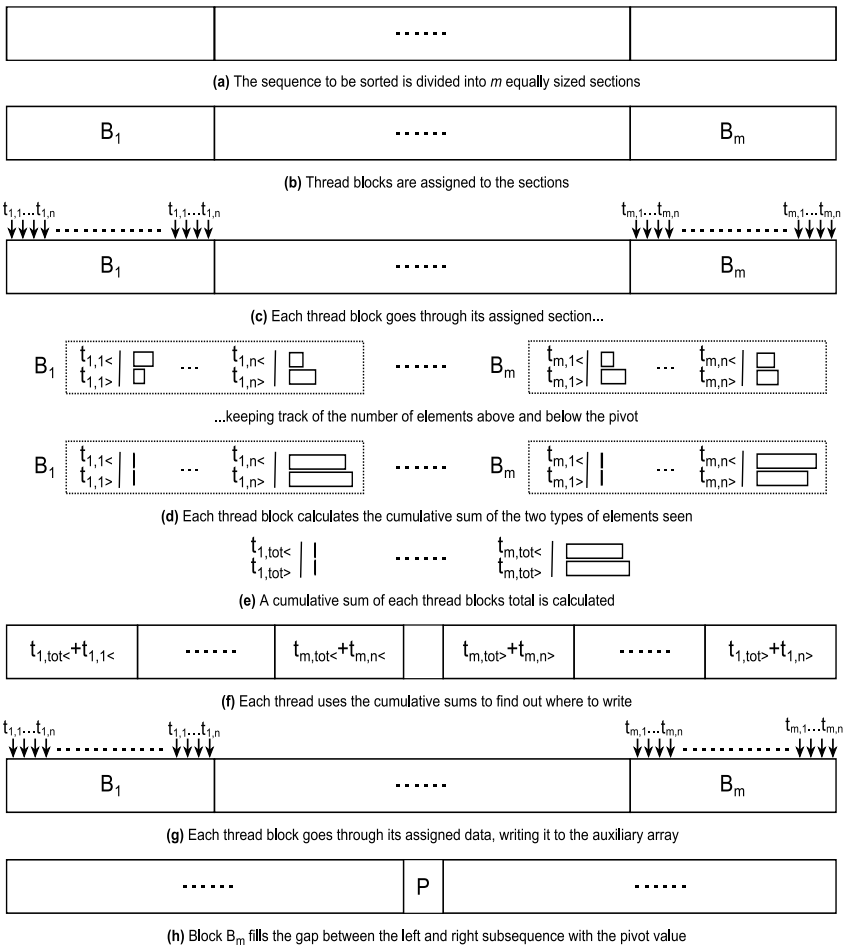


Fig. 3: Partitioning a sequence

we make sure that each thread knows where to write in the auxiliary buffer? It should also be noted that it is 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[1] 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 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

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

Algorithm 1 GPU-Quicksort (CPU Part)

```
procedure GPUQSORT(size, d, d̂)                          ▷ d contains the sequence to be sorted.
startpivot ←median(d_0, d_{size/2}, d_size)              ▷ d_x is the value at index x.
work ← {(d_{0→size}, startpivot)}                        ▷ d_{x→y} is the sequence from index x to y.
done ← ∅
while work ≠ ∅ ∧ |work| + |done| < maxseq do            ▷ Divide into maxseq subsequences.
    blocksize ← Σ_{(seq,pivot)∈work} ||seq|| / maxseq    ▷ ||seq|| is the length of sequence seq.
    blocks ← ∅
    for all (seq_{start→end}, pivot) ∈ work do           ▷ Divide sequences into blocks.
        blockcount ← ⌈ ||seq|| / blocksize ⌉             ▷ Number of blocks to create for this sequence.
        parent ← (seq, seq, blockcount)                  ▷ Shared variables for all blocks of this sequence.

        for i ← 0, i < blockcount − 1, i ← i + 1 do
            bstart ← start + blocksize · i
            blocks ← blocks ∪ {(seq_{bstart→bstart+blocksize}, pivot, parent)}
        blocks ← blocks ∪ {(seq_{start+blocksize·(blockcount−1)→end}, pivot, parent)}

    news ← gqsort≪ |blocks| ≫ (blocks, d, d̂)             ▷ Start |blocks| thread blocks.
    work ← ∅
    for all (seq, pivot) ∈ news do                       ▷ If the new sequences are too long; partition them further.
        if ||seq|| < size/maxseq then
            done ← done ∪ {(seq, pivot)}
        else
            work ← work ∪ {(seq, pivot)}
done ← done ∪ work                                       ▷ Merge the sets done and work.
lqsort≪ |done| ≫ (done, d, d̂)                            ▷ Do final sort.
```

thread just counts the number of elements it has seen that have a 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 does not need to be included in any of the two subsequences.

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 is 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 subsequences that are longer than $minlength = n/maxseq$, where n is the total number of elements to sort, and to stop when we have $maxseq$ number of sequences. We discuss how to select a good value for

Algorithm 2 GPU-Quicksort (First Phase GPU Kernel)

function GQSORT($blocks, d, \hat{d}$)
var global $sstart, send, oldstart, oldend, blockcount$
var block local $lt, gt, pivot, start, end, s$
var thread local $i, lfrom, gfrom, lpivot, gpivot$

$(s_{start \to end}, pivot, parent) \leftarrow blocks_{blockid}$ ▷ Get the sequence block assigned to this thread block.
$lt_{threadid}, gt_{threadid} \leftarrow 0, 0$ ▷ Set thread local counters to zero.

$i \leftarrow start + threadid$ ▷ Align thread accesses for coalesced reads.
for $i < end, i \leftarrow i + threadcount$ **do** ▷ Go through the data...
  if $s_i < pivot$ **then** ▷ counting elements that are smaller...
    $lt_{threadid} \leftarrow lt_{threadid} + 1$
  if $s_i > pivot$ **then** ▷ or larger compared to the pivot.
    $gt_{threadid} \leftarrow gt_{threadid} + 1$

$lt_0, lt_1, lt_2, ..., lt_{sum} \leftarrow 0, lt_0, lt_0 + lt_1, ..., \sum_{i=0}^{threadcount} lt_i$ ▷ Calculate the cumulative sum.
$gt_0, gt_1, gt_2, ..., gt_{sum} \leftarrow 0, gt_0, gt_0 + gt_1, ..., \sum_{i=0}^{threadcount} gt_i$

if $threadid = 0$ **then** ▷ Allocate memory in the sequence this block is a part of.
  $(seq_{sstart \to send}, oseq_{oldstart \to oldend}, blockcount) \leftarrow parent$ ▷ Get shared variables.
  $lbeg \leftarrow FAA(sstart, lt_{sum})$ ▷ Atomic increment allocates memory to write to.
  $gbeg \leftarrow FAA(send, -gt_{sum}) - gt_{sum}$ ▷ Atomic is necessary since multiple blocks access this variable.

$lfrom = lbeg + lt_{threadid}$
$gfrom = gbeg + gt_{threadid}$

$i \leftarrow start + threadid$
for $i < end, i \leftarrow i + threadcount$ **do** ▷ Go through data again writing elements
  if $s_i < pivot$ **then** ▷ to the their correct position.
    $\neg s_{lfrom} \leftarrow s_i$ ▷ If s is a sequence in d, $\neg s$ denotes the corresponding
    $lfrom \leftarrow lfrom + 1$ ▷ sequence in $\hat{d}$ (and vice versa).
  if $s_i > pivot$ **then**
    $\neg s_{gfrom} \leftarrow s_i$
    $gfrom \leftarrow gfrom + 1$

if $threadid = 0$ **then**
  if $FAA(blockcount, -1) = 0$ **then** ▷ Check if this is the last block in the sequence to finish.
    for $i \leftarrow sstart, i < send, i \leftarrow i + 1$ **do** ▷ Fill in pivot value.
      $d_i \leftarrow pivot$
    $lpivot \leftarrow median(\neg seq_{oldstart}, \neg seq_{(oldstart+sstart)/2}, \neg seq_{sstart})$
    $gpivot \leftarrow median(\neg seq_{send}, \neg seq_{(send+oldend)/2}, \neg seq_{oldend})$
    result $\leftarrow$ **result** $\cup\{(\neg seq_{oldstart \to sstart}, lpivot)\}$
    result $\leftarrow$ **result** $\cup\{(\neg seq_{send \to oldend}, gpivot)\}$

maxseq in Section 5.

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 3 (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 (a warp being 32 consecutive threads that are always scheduled together) accessing different memory banks to increase performance.

Algorithm 3 GPU-Quicksort (Second Phase GPU Kernel)

```
procedure LQSORT(seqs, d, d̂)
var block local  lt, gt, pivot, workstack, start, end, s, newseq1, newseq2, longseq, shortseq
var thread local  i, lfrom, gfrom

push seqs_blockid on workstack                          ▷ Get the sequence assigned to this thread block.

while workstack ≠ ∅ do
    pop s_start→end from workstack                      ▷ Get the shortest sequence from set.

    pivot ← median(s_start, s_(end+start)/2), s_end)    ▷ Pick a pivot.
    lt_threadid, gt_threadid ← 0, 0                     ▷ Set thread local counters to zero.

    i ← start + threadid                                ▷ Align thread accesses for coalesced reads.
    for i < end, i ← i + threadcount do                 ▷ Go through the data...
        if s_i < pivot then                             ▷ counting elements that are smaller...
            lt_threadid ← lt_threadid + 1
        if s_i > pivot then                             ▷ or larger compared to the pivot.
            gt_threadid ← gt_threadid + 1

    lt_0, lt_1, lt_2, ..., lt_sum ← 0, lt_0, lt_0 + lt_1, ..., Σ_{i=0}^{threadcount} lt_i     ▷ Calculate the cumulative sum.
    gt_0, gt_1, gt_2, ..., gt_sum ← 0, gt_0, gt_0 + gt_1, ..., Σ_{i=0}^{threadcount} gt_i

    lfrom ← start + lt_threadid                         ▷ Allocate locations for threads.
    gfrom ← end − gt_threadid+1

    i ← start + threadid      ▷ Go through the data again, storing everything at its correct position.
    for i < end, i ← i + threadcount do
        if s_i < pivot then
            ¬s_lfrom ← s_i
            lfrom ← lfrom + 1
        if s_i > pivot then
            ¬s_gfrom ← s_i
            gfrom ← gfrom + 1

    i ← start + lt_sum + threadid                       ▷ Store the pivot value between the new sequences.
    for i < end − gt_sum, i ← i + threadcount do
        d_i ← pivot

    newseq1 ← ¬s_start→start+lt_sum
    newseq2 ← ¬s_end−gt_sum→end

    longseq, shortseq ← max(newseq1, newseq2), min(newseq1, newseq2)
    if ||longseq|| <MINSIZE then                        ▷ If the sequence is shorter than MINSIZE
        altsort(longseq, d)                             ▷ sort it using an alternative sort and place result in d.
    else
        push longseq on workstack
    if ||shortseq|| <MINSIZE then
        altsort(shortseq, d)
    else
        push shortseq on workstack
```

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 3 (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 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 do not already have more than *maxseq* sequences. If we do, the next phase begins. Otherwise we go through another iteration. (See Algorithm 1).

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.

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 has shown [Hoare 1962] 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 [Sedgewick 1978], 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, does not 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 is calculated so often that every performance increase counts, so we decided to use the parallel cumulative sum implementation described in [Harris et al. 2007] which is based on [Blelloch 1993]. 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* $O(\frac{n}{p}\log(n))$ *on a CRCW PRAM.*

PROOF. For the analysis we combine phase one and two since there is no difference between them from a complexity perspective. We assume a p-process arbitrary CRCW PRAM [Jaja 1992]. 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 $O(\frac{n}{p})$ steps, where n is the length of the sequence to sort and p is the number of processors.

The accumulate function has a time complexity of $O(\log(T))$ [Blelloch 1993], where T is the number of threads per thread block. Since T does not vary with n or p, it is a constant cost. The alternative sort only needs to sort sequences that are smaller than q, where q is dependent on the amount of available shared memory on the graphics processor. This means that the worst case complexity of the alternative sort is not dependent on n or p and is thus constant.

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

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

Assuming that $q \ll n$ we can set $q = 1$ which gives us the standard Quicksort recurrence relation, which is proved to be $O(\frac{n}{p}\log(n))$. □

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

PROOF. Phase one is bounded in that it only needs to keep track of a maximum of $maxseq$ subsequences. Phase two always recurses on the smaller sequence, giving a bound of $\log_2(keysize)$ subsequences that needs to be stored. $maxseq$ and $\log_2(keysize)$ do not depend on the size of the sequence and can thus be seen as constants. The memory complexity then becomes $2n + c$, the size of the sequence to be sorted plus an equally large auxiliary buffer and a constant needed for the bookkeeping of $maxseq$ and $B\log_2(keysize)$ sequences, where B is the amount of thread blocks used. □

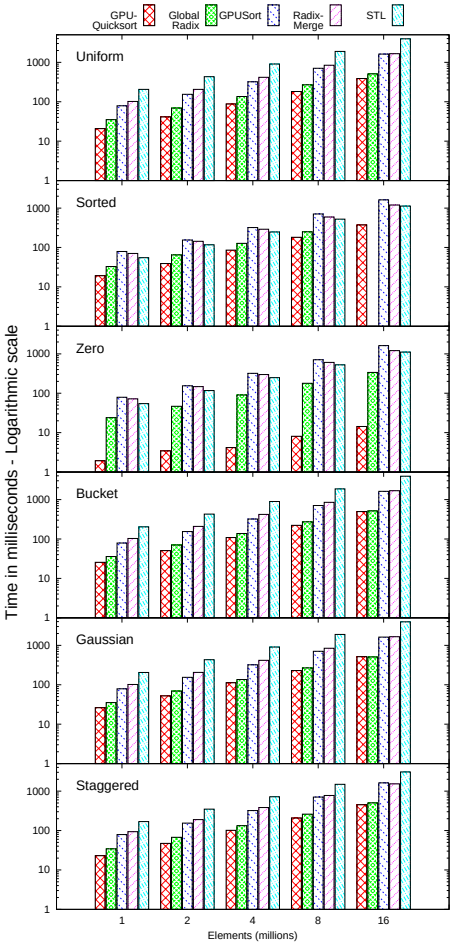


Fig. 4: Results on the 8800GTX

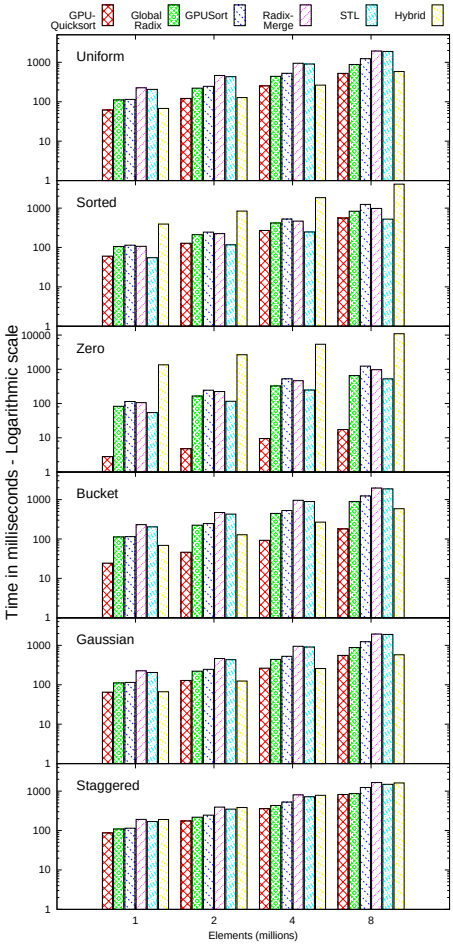


Fig. 5: Results on the 8600GTS

5. EXPERIMENTAL EVALUATION

5.1 Hardware

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

5.2 Algorithms used

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

GPUSort Uses bitonic merge sort [Govindaraju et al. 2005].

Radix-Merge Uses radix sort to sort blocks that are then merged [Harris et al. 2007].

Global Radix Uses radix sort on the entire sequence [Sengupta et al. 2007].

Hybridsort Splits the data with a bucket sort and uses merge sort on the resulting blocks [Sintorn and Assarsson 2007].

STL-Introsort This is the Introsort implementation found in the C++ Standard Library [Musser 1997]. Introsort is based on Quicksort, but switches to heapsort 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.

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 [Sengupta et al. 2007].

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

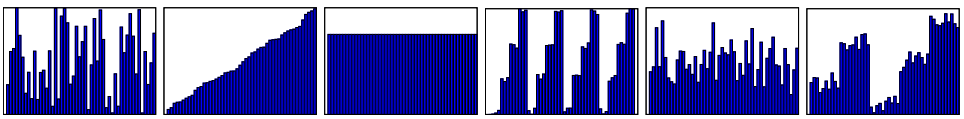
We used different pivot selection schemes for the two phases. In the first phase we took the average of the minimum and maximum element in the sequence and in the second we picked the median of the first, middle and last element as the pivot, a method suggested by Singleton[Singleton 1969].

We used the more computationally expensive method in the first phase to try to have a more even size of the subsequences that are assigned to each thread block in the next phase. This method works perfectly well with numbers, but for generic key types it has to be replaced with e.g. picking a random element or using the same method as in phase two.

The source code of GPU-Quicksort is available for non-commercial use [Cederman and Tsigas 2007].

5.3 Input Distributions

For benchmarking we used the following distributions which are commonly used yardsticks in the literature to compare the performance of different sorting algorithms [Helman et al. 1998]. The source of the random uniform values is the Mersenne Twister [Matsumoto and Nishimura 1998].



(a) Uniform (b) Sorted (c) Zero (d) Bucket (e) Gaussian (f) Staggered

Fig. 6: Visualization of sequences generated with different distributions.

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

Phase	Registers	Shared Memory (32-bit words)
Phase one	14	$4T + 14$
Phase two	14	$\max(2T, S) + 98$

Table I: Registers and shared memory per block required by each phase. T is the number of threads per block and S is the minimum sequence sorted by Quicksort.

Sorted Sorted uniformly distributed values.

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 $(2i+2)\frac{2^{31}}{p} - 1$. For blocks where $i > \lfloor \frac{p}{2} \rfloor$, the values all lie between $(2i-p)\frac{2^{31}}{p}$ and $(2i-p+1)\frac{2^{31}}{p} - 1$. We decided to use a p value of 128.

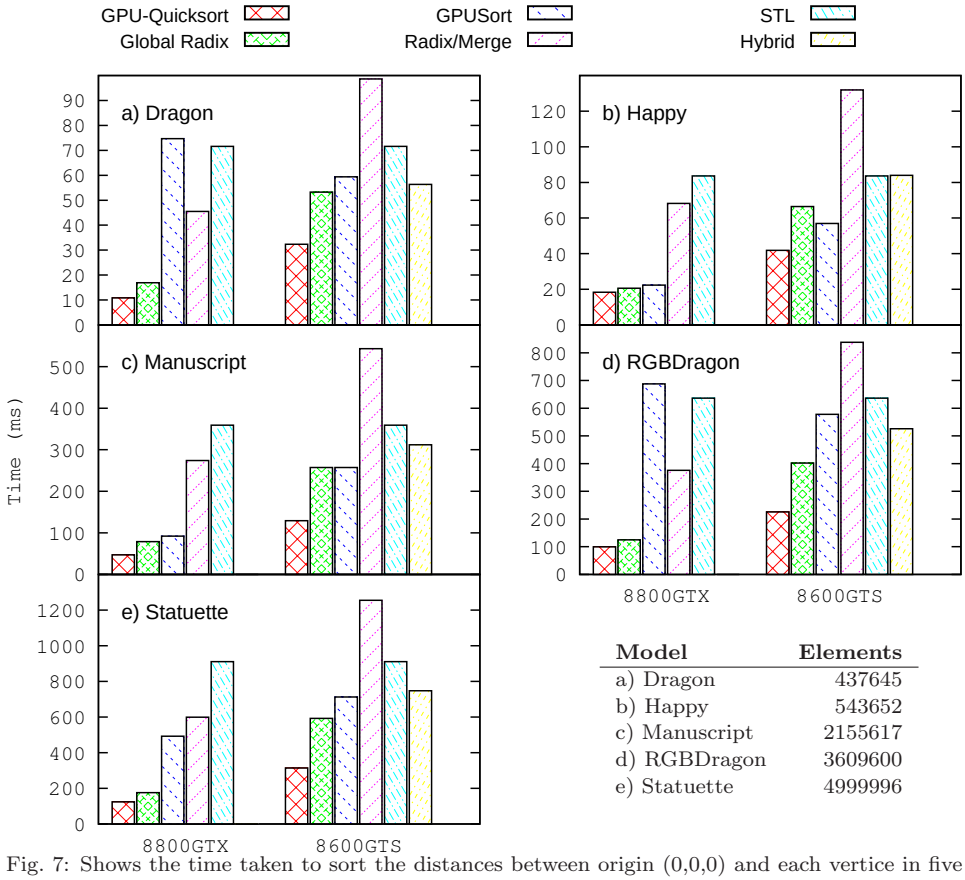
The results presented in Figure 4 and 5 are based on experiments sorting sequences of integers. We have done experiments using floats instead, but found no difference in performance.

To get an understanding of how GPU-Quicksort performs when faced with real-world data we have evaluated it on the task of visibility ordering, i.e. sorting a set of vertices in a model according to their distance from a point in 3D-space. The 3D-models have been taken from *The Stanford 3D Scanning Repository* which is a resource commonly used in the literature [Stanford 2008]. By calculating the distance from the viewer (in the experiments we placed the viewer at origin (0,0,0)), we get a set of floats that needs to be sorted to know in which order the vertices and accompanying faces should be drawn. Figure 7 shows the result from these experiments on the two different graphics processors. The size of the models ranges from 5×10^6 elements to 5×10^7 elements.

GPUSort can only sort sequences that have a length which is a power of two, due to the use of a bitonic sorting network. In Figure 8a we have visualized the amount of data that is sorted on the GPU versus the CPU. In Figure 8b we also show the relative amount of time spent on the GPU versus the CPU.

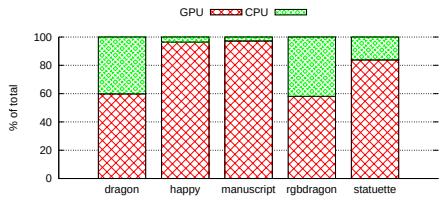
Table I shows the number of registers and amount of shared memory needed by each phase. The higher the number of registers used the fewer the threads that can be used per block and the higher the amount of shared memory used the fewer number of blocks can be run concurrently.

The GPU-Quicksort algorithms can be seen as having three parameters, the maximum number of subsequences created in the first phase, *maxseq*, the number of threads per block, *threads*, and the minimum size of sequence to sort with Quicksort before switching to the alternative sort, *sbsize*. In Figure 9 we have tried several different combinations of these parameters on the task of sorting a uniformly distributed sequence with 8 million elements. The figure shows the five best and the five worst results divided into how much time each phase takes.

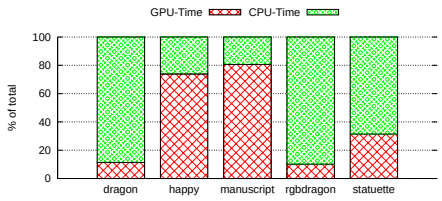


Model	Elements
a) Dragon	437645
b) Happy	543652
c) Manuscript	2155617
d) RGBDragon	3609600
e) Statuette	4999996

Fig. 7: Shows the time taken to sort the distances between origin (0,0,0) and each vertice in five different 3D-models. a) Dragon, b) Happy, c) Manuscript, d) RGBDragon and e) Statuette. Also shown is a table with the number of vertices in each figure, i.e. the size of the sequence to sort.



(a) Percentage of data sorted on the GPU vs CPU when using GPUSort.

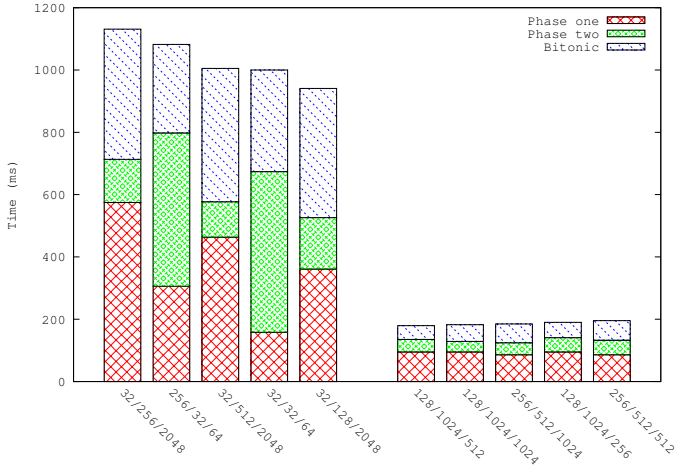


(b) Percentage of time spent sorting on the GPU vs CPU when using GPUSort.

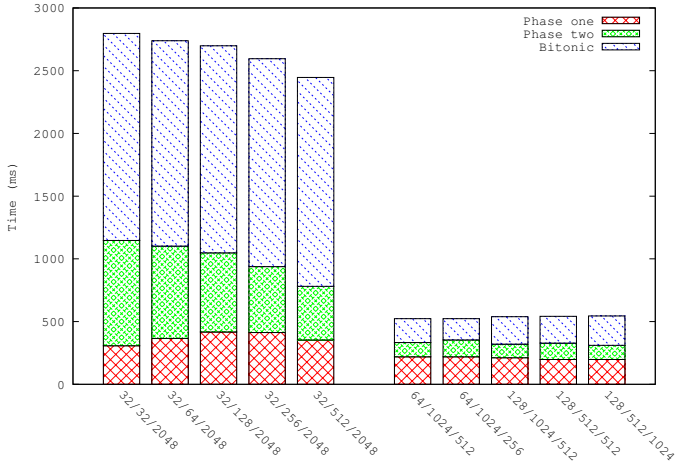
Fig. 8: GPUSort on 8600GTS.

Figure 10 shows how the performance changes when we vary one parameter and then pick the best, the worst and the average result among all other combinations of the two other parameters.

The optimal selection of these parameters varies with the size of the sequence. Figure 11 shows how the values that gives the best result changes when we run larger sequences. All variables seems to increase with the size of the sequence.



(a) 8800GTX.



(b) 8600GTS.

Fig. 9: The five best and worst combinations of *threads per block/maximum number of subsequences created in phase one/minimum size of sequence to sort using Quicksort.*

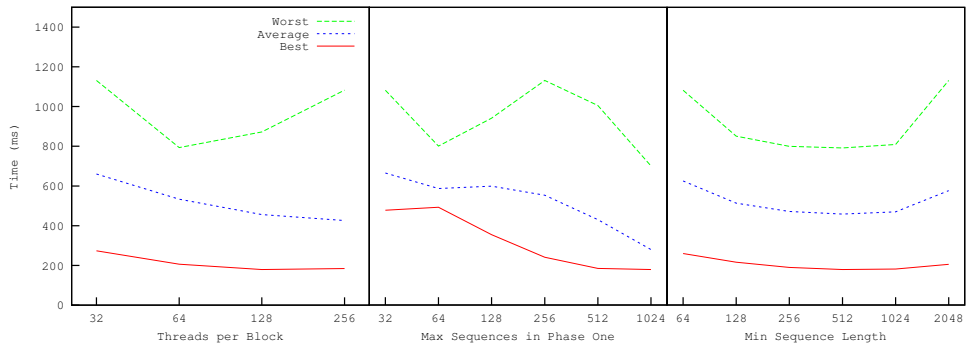
GPU	Threads per Block	Max Blocks in Phase One	Min Sequence to Sort
8800GTX	$optp(s, 0.00001172, 53)$	$optp(s, 0.00003748, 476)$	$optp(s, 0.00004685, 211)$
8600GTS	$optp(s, 0, 64)$	$optp(s, 0.00009516, 203)$	$optp(s, 0.00003216, 203)$

Table II: Shows the constants used to select suitable parameters for a given sequence length s.

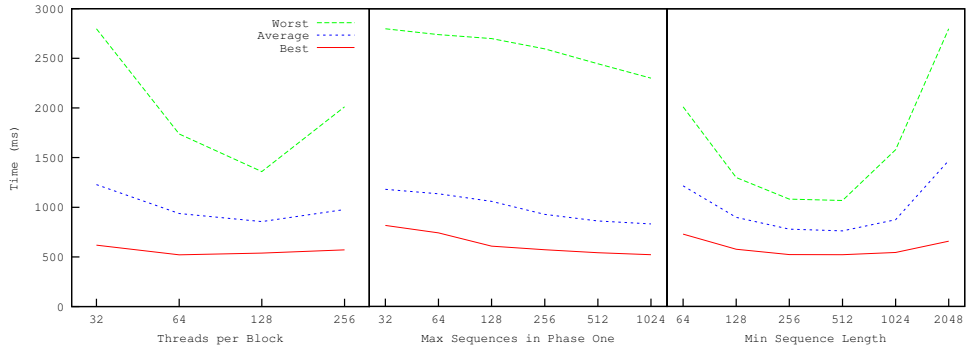
To get the best parameters for any given sequence length we use a linear function for each parameter to calculate its value.

$$optp(s, k, m) := 2^{\lfloor \log_2(sk+m)+0.5 \rfloor}$$

The parameters for this function are presented in Table II. They were calculated by doing a linear regression using the measured values presented in Figure 11.



(a) 8800GTX.



(b) 8600GTS.

Fig. 10: Varying one parameter, picking the best, worst and average result from all possible combination of the others parameters.

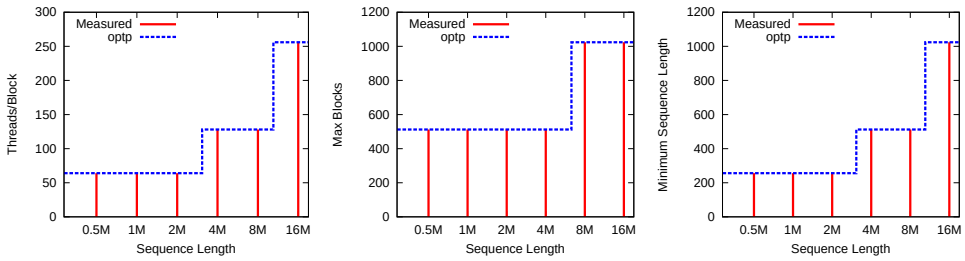
5.4 Discussion

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

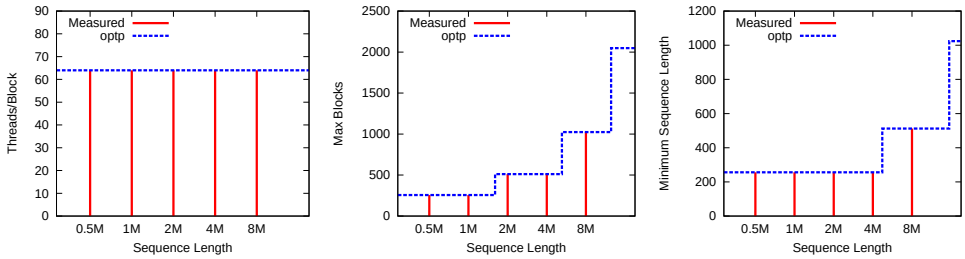
Since sorting on GPUs has received a lot of attention it might be reasonable to start with the following question; **is there really a point in sorting on the GPU?** If we take a look at the radix-merge sort in Figure 5 we see that it performs comparable to the CPU reference implementation. Considering 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 or more compared to Introsort, the CPU reference. On the higher-end GPU in Figure 4, the difference in speed can be up to 10 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



(a) 8800GTX.



(b) 8600GTS.

Fig. 11: The parameters vary with the length of the sequence to sort.

among the best. There was no distribution that caused problems to the performance of GPU-Quicksort. As can be seen when comparing the performance on the two GPUs, GPU-Quicksort shows a speedup by around 3 times on the higher-end GPU. The higher-end 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 the paper by Sengupta et al. [Sengupta et al. 2007].

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 does not 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 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 performs very well on the uniform, bucket and gaussian distribution, but it loses speed on the staggered distributions and becomes immensely slow on the zero and sorted 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 [Sintorn and Assarsson 2007]. This however lowers the performance and wouldn't affect the result in the zero distribution.

How are the algorithms affected by the higher-end GPU? GPUSort does not increase as much in performance as the other algorithms when run 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 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 takes the same amount of time for all of the distribution since it is a sorting network, which means it always performs the same number of operations regardless of the distribution.

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. GPUSort performs the same number of operations regardless of the distribution, so there was no change there. The hybrid sort placed all elements in the same bucket which caused it to show much worse performance than the others. 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 has to do two passes through the data and can sort the zero distribution in $O(n)$ time.

On the sorted distribution all algorithms gain in speed except GPUSort and the hybrid. The CPU reference becomes faster than GPUSort and radix-merge on the high-end graphics processor and is actually the fastest when compared to the algorithms run on the low-end graphics processor.

On the real-world data experiments, Figure 7, we can see that GPU-Quicksort performs well on all models. It seems that the relative differences between the algorithms are much the same as they were with the artificial distributions. One interesting thing though is the inconsistency in GPUSort. It is much faster on the larger manuscript and statuette models than it is on the smaller dragon models. This has nothing to do with the distribution since bitonic sort is not affected by it, so this is purely due to the fact that GPUSort needs to sort part of the data on the CPU since sequences needs to have a length that is a power of two to be sorted with bitonic sort. In Figure 8a we can see that for the two dragon-models 40% is spent sorting on the CPU instead of on the GPU. Looking at Figure 8b we see that this translates to 90% of the actual sorting time. This explains the strange variations in the experiments.

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 traverses 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 [Sengupta et al. 2007]. According to the authors this was due to it being more dependent on the processor speed than on the bandwidth.

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.

ACKNOWLEDGMENTS

We would like to thank Georgios Georgiadis and Marina Papatriantafilou for their valuable comments during the writing of this paper. 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. Last but not least, we would like to thank the anonymous reviewers for their comments that helped us improve the presentation of the algorithm significantly.

REFERENCES

Bilardi, G. and Nicolau, A. 1989. Adaptive Bitonic Sorting. An Optimal Parallel Algorithm for Shared Memory Machines. *SIAM Journal on Computing 18,* 2, 216–228.

Blelloch, G. E. 1993. Prefix Sums and Their Applications. In *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. Morgan Kaufmann.

Cederman, D. and Tsigas, P. December 2007. GPU Quicksort Library. `www.cs.chalmers.se/~dcs/gpuqsortdcs.html`.

Dowd, M., Perl, Y., Rudolph, L., and Saks, M. 1989. The Periodic Balanced Sorting Network. *Journal of the ACM 36,* 4, 738–757.

Evans, D. J. and Dunbar, R. C. 1982. The Parallel Quicksort Algorithm Part 1 - Run Time Analysis. *International Journal of Computer Mathematics 12*, 19–55.

Govindaraju, N., Raghuvanshi, N., Henson, M., and Manocha, D. 2005. A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors. Tech. rep., University of North Carolina-Chapel Hill.

Govindaraju, N. K., Gray, J., Kumar, R., and Manocha, D. 2006. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. 325–336.

Govindaraju, N. K., Raghuvanshi, N., and Manocha, D. 2005. 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*. 611–622.

Gress, A. and Zachmann, G. 2006. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*.

Harris, M., Sengupta, S., and Owens, J. D. 2007. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Chapter 39.

Heidelberger, P., Norton, A., and Robinson, J. T. 1990. Parallel Quicksort Using Fetch-And-Add. *IEEE Transactions on Computers 39,* 1, 133–138.

Helman, D. R., Bader, D. A., and JáJá, J. 1998. A Randomized Parallel Sorting Algorithm with an Experimental Study. *Journal of Parallel and Distributed Computing 52,* 1, 1–23.

Hoare, C. A. R. 1961. Algorithm 64: Quicksort. *Communications of the ACM 4,* 7, 321.

Hoare, C. A. R. 1962. Quicksort. *Computer Journal 5,* 4, 10–15.

Jaja, J. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley.

Kapasi, U. J., Dally, W. J., Rixner, S., Mattson, P. R., Owens, J. D., and Khailany, B. 2000. Efficient Conditional Operations for Data-parallel Architectures. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*. 159–170.

Khronos Group. 2008. OpenCL (Open Computing Language). `http://www.khronos.org/opencl/`.

Kipfer, P., Segal, M., and Westermann, R. 2004. UberFlow: A GPU-based Particle Engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. 115–122.

Kipfer, P. and Westermann, R. 2005. Improved GPU Sorting. In *GPUGems 2*, M. Pharr, Ed. Addison-Wesley, Chapter 46, 733–746.

Matsumoto, M. and Nishimura, T. 1998. Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *Transactions on Modeling and Computer Simulation 8,* 1, 3–30.

Musser, D. R. 1997. Introspective Sorting and Selection Algorithms. *Software - Practice and Experience 27,* 8, 983–993.

Purcell, T. J., Donner, C., Cammarano, M., Jensen, H. W., and Hanrahan, P. 2003. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*. 41–50.

Sedgewick, R. 1978. Implementing Quicksort Programs. *Communications of the ACM 21,* 10, 847–857.

Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. 2007. Scan Primitives for GPU Computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. 97–106.

Singleton, R. C. 1969. Algorithm 347: an Efficient Algorithm for Sorting with Minimal Storage. *Communications of the ACM 12,* 3, 185–186.

Sintorn, E. and Assarsson, U. 2007. Fast Parallel GPU-Sorting Using a Hybrid Algorithm. In *Workshop on General Purpose Processing on Graphics Processing Units*.

Stanford. 2008. The Stanford 3D Scanning Repository. `graphics.stanford.edu/data/3Dscanrep`.

Tsigas, P. and Zhang, Y. 2003. 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*. 372–381.

Received Month Year; revised Month Year; accepted Month Year