A Practical Quicksort Algorithm for Graphics Processors
Kait Kasak

Abstract—This paper is a shorter, less technical summary of the one [Cederman, 2008] published by Daniel Cederman and Philippas Tsigas which describes implementing a fast Quicksort algorithm on a GPU (Graphics Processor Unit).

I. INTRODUCTION

While analyzing the performance of different sorting algorithms on GPUs has become more commonplace during recent years, quicksort has generally been discarded as a possible candidate. It is widely considered an inefficient algorithm to implement on Graphics Processors. Mostly due to their limited cache and expensive thread synchronization, which leads to poor performance with in-place substitution [Cederman, 2008, p 249].

In this paper a different implementation of quicksort is proposed, that is more suitable for the architecture of GPUs.

A. Why sort on GPUs?

Since most sorting algorithms are memory bandwidth bound [Cederman, 2008], interest in sorting on the high-bandwidth GPUs is easily explainable. The fastest currently available GPU (Geforce 580 GTX) has a memory bandwidth of 192.4 GB/sec [Nvidia, 2010] compared to the best consumer CPU in this regard, Intel Core i7 which has triple channel memory, has only 25.6 GB/sec [Intel, 2010]. A difference of an order of a magnitude.

B. Parallelisation of quicksort

There are two naive ways to parallelise quicksort, both of them are quite inefficient on the GPU:

• The trivial approach is to assign new recursive calls of the function to different processors. This however means that there is very little parallelisation in the beginning when sequences can be very long and few.

• The second approach is to divide each sequence into sorted blocks that can be dynamically assigned to available processors. however this requires a lot of atomic synchronization primitive FAA(fetch and add) which either isn’t supported directly on some GPUs or is very slow[Cederman, 2008].

Because of these inter-thread synchronization problems quicksort has usually been considered too slow to be efficiently implement on GPUs [Naga K. Govindaraju, 2005, Cederman, 2008]. In section III an alternative approach is covered, which can be considered a hybrid of these two approaches with a number of additional changes to mitigate the FAA usage.

II. SYSTEM MODEL

The GPU mainly used for the article was a NVIDIA Geforce 8800 GTX and the algorithm was implemented in CUDA. CUDA is a parallel computing architecture and programming model developed by NVIDIA [GPGPU.org, 2011]. It consists of a compiler for a C-based language which can be used to create kernels that can be executed on the GPU [Cederman, 2008].

The graphics card has 16 multiprocessors each consisting of 8 SIMD (Single Input Multiple Data) processors that can execute the same instruction on different data. Having in total 768 threads [Nvidia, 2008] and 16KiB of fast local memory [Cederman, 2008]. Threads are grouped into thread blocks that are assigned to a specific multiprocessor and they are usually executed in parallel if enough resources are available at a given time.

Threads within the same thread block can use the multiprocessor shared memory and also synchronize through thread barrier function. Data is stored in a large global memory. There is no implicit cache but fast shared local memory can usually be used by thread-blocks.

III. THE ALGORITHM

Quicksort algorithm recursively partition a sequence of elements based on randomly chosen pivots. To avoid the problem of limited parallelism during the early stages of execution (too few sequences available) mentioned in I-B, the algorithm is divided into two phases.

In the first phase there are too few (sub)sequences of quick sort for assigning each thread block its own, therefore multiple thread blocks are working on same sequences. This phase requires CPU - GPU communication for synchronizing between thread block results, which can only be done if all of the required thread blocks in a given sequence are finished. In the second phase each thread block is assigned its own sub-sequence of input data, eliminating the need for synchronization between different thread blocks. The second phase can be entirely done on a GPU.

The principle of two phase partitioning is outlined on figure 1. In step A and B the sequence is split into m equal parts (where m is the number of thread blocks available) and each block is assigned one of those sequences. In part C the sequence is partitioned in a typical quick sort way, meaning a random pivot is selected and elements smaller than the pivot are moved to the left of it and larger to the right.

As multiple threads are doing this process simultaneously we need to synchronize this behavior. This is done by each thread reading one element and telling the other threads (using
Figure 1. Partitioning a sequence \( (m \text{ thread blocks with } n \text{ threads each}) \)

a barrier function) where it wants to write it, left or right of the pivot. Because of this, each thread knows the number of other threads (with lower ID number), that want to write to either side of the pivot. For instance if our active thread has a value lower than the pivot and \( x \) other threads also want to write to the left, we know to write the element to position \( x + 1 \) in the write buffer. This approach is called cumulative sum. These actions are done in steps D, E, F.

As calculating cumulative sum per element can be expensive, this is optimized, by doing two passes over the data. First, each thread just counts the number of larger elements than the pivot, it encounters in its search space and shares the total number. In the second pass these sums are used, instead of single elements, for calculating where to write by the other threads (step G).

IV. RESULTS

A. Experiment evaluation

GPU-Quicksort was compared to a number of other well known GPU sorting types and one CPU based sorting (STL-Introsort). The other sorting algorithms used are:

- **GPUSort** - Using bitonic merge sort [Naga K. Govindaraju, 2005]
- **Radix-Merge** - Uses radix sort to blocks that are merged.
- **Global Radix** - Uses radix sort on the entire sequence.
- **HybridSort** - A bucket sort followed by a merge sort
- **STL-Introsort** - CPU side implementation of Introsort in C++ for comparison

Random distributions used were: Uniform (value \( s \) between \( 0 - 2^{31} \)), Zero (random fixed value) Bucket, Gaussian and Staggered, which are commonly used yardsticks for sorting. The sorting was tested on Integers. The results are shown on figure 2.

The implemented quicksort shows the best performance of the used algorithms in almost all cases. In all distributions of data it performed close to the lower bound of computational complexity of \( O(n \cdot \log(n)) \). Higher end GPU (8800GTX) is roughly 3 times faster than the mid-range equivalent (8600 GTS).

V. CONCLUSIONS

The referred paper [Cederman, 2008] presented an efficient implementation of the quicksort algorithm on the GPU which was previously though unlikely to exist. Due to the minimization of synchronization and the usage of high bandwidth and massive parallelisation the proved to be one of the fastest to exist thus far and can therefore be considered as a practical alternative.

REFERENCES


