Annals of DAAAM for 2011 & Proceedings of the 22nd International DAAAM Symposium, Volume 22, No. 1, ISSN 1726-9679
ISBN 978-3-901509-83-4, Editor B. Katalinic, Published by DAAAM International, Vienna, Austria, EU, 2011
Make Harmony between Technology and Nature, and Your Mind will Fly Free as a Bird
Annals & Proceedings of DAAAM International 2011

OPTIMIZATION TECHNIQUES FOR DATA SORTING ALGORITHMS

PIRJAN, A[lexandru]

Abstract: *This paper describes the designing of high performance parallel radix sort routines for many cores graphics processing units, benefitting from the high computational power offered by the Compute Unified Device Architecture (CUDA). I address issues of great importance when optimizing a classical data-sorting algorithm regarding fine-grain parallelism and memory management techniques. One particular interest was to research how well the optimization techniques, applied to data sorting algorithms written in CUDA, scale to the latest generation of general-purpose graphic processors units (GPGPU), like the Fermi architecture implemented in the GTX580 and the previous architecture implemented in GTX285.*

Key words: *parallelism, radix sort, CUDA, algorithms, shared memory*

1. INTRODUCTION

Sorting is one of the most studied algorithmic problems. Its importance has led to the designing of efficient sorting algorithms for a variety of parallel architectures (Cormen et al., 2001). A wide class of efficient sorting algorithms is based on optimized computing routines. Sorting operations are extensively used in database systems and are essential in computer graphics and geographic information systems, representing the basis for building spatial data structures. Efficient sorting routines are of paramount importance in implementing algorithms such as parallel programming models based on MapReduce (Dean & Ghemawat, 2008). Therefore, it is very important to implement efficient sorting routines on any programming platform taking into account the continuous evolution of computer hardware architectures that provide an increased computational processing power.

The continuous increase in the level of implemented parallelism has been one of the dominant trends in microprocessor architectures in recent years. Many-cores central processing units are becoming more and more widespread and thus there is an evident tendency of increasing the level of parallelism by using multiple processing units. Based on multiple processing cores, the graphics processing units represent viable solutions for increasing the level of parallelism. For example, the current NVIDIA Fermi graphics processing units contain up to 512 processing cores per chip and in contrast to previous generations of GPUs, they can be programmed directly in C using CUDA (Dean & Ghemawat, 2008). In this paper, I describe the design of efficient sorting algorithms for GPUs that implement the Compute Unified Device Architecture.

CUDA offers a flexible programming model, thus the current generation of GPUs allows us to take into account a wider range of algorithms than it was previously possible on GPU architectures. In particular, I focus on optimizing a radix sort algorithm (with binary represented keys) for the GTX580 architecture. The graphic processing unit GPU is a multithread processor and therefore, in order to design efficient algorithms on such architectures one has to involve large amounts of fine-grained parallelism.

The described radix sort algorithm implements multiple parallel scan operations through efficient fine grain parallelism (Nickolls et al., 2008); (Dusseau et al., 1996).

2. THE CUDA PARALLEL PROGRAMMING MODEL

In the following, I briefly present the most important aspects regarding the architecture of current GPUs and NVIDIA's CUDA parallel programming model.

For a long time, graphics processing units (GPUs) have been used solely for graphics rendering purposes on computers. In recent years, the GPU has evolved from an initial specialized architecture to a complex one, able to solve a wider range of complex problems and not only the video rendering. The introduction of the NVIDIA Compute Unified Device Architecture made possible to accelerate and improve a large suite of applications. A briefly description of this architecture and its main features are depicted in Figure 1.

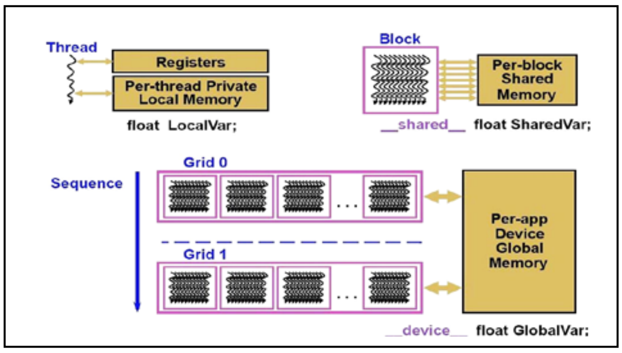
Thread
Registers
Per-thread Private Local Memory
float LocalVar;
Block
Per-block Shared Memory
__shared__ float SharedVar;
Sequence
Grid 0
Grid 1
Per-app Device Global Memory
__device__ float GlobalVar;

Fig. 1. NVIDIA CUDA (Dean & Ghemawat, 2008)

NVIDIA graphics processor is enabled by the CUDA hardware and software architecture to execute programs written in different languages including C, C++ or FORTRAN. A CUDA program is designed to invoke a set of parallel program kernels, that are processed in parallel by multiple threads, grouped into thread blocks and grids of thread blocks.

3. THE RADIX SORT ALGORITHM

In the category of sorting algorithms, the radix sort is among the oldest, probably the best known and the most efficient in sorting small keys on sequential machines. One of the most important advantages of the radix sort algorithm is the fact that it is easy to parallelize, based on the reduction of the counting sort used at each pass to a parallel prefix or sum operation (Satish et al., 2009). Another major advantage is that scans can be efficiently implemented on a GPU or on other many-cores processors, being a fundamental operation in managing parallel data (Nickolls et al., 2008). Therefore, radix sort may be considered one of the easiest parallel sort algorithms from the implementation point of view and one of the most efficient ones, comparable with some more sophisticated algorithms (Satish et al., 2009). In order to optimize the radix sort, I have divided the sequence into tiles that have been assigned to a number of t thread blocks. I have used memory optimization techniques and global synchronization between each separate parallel kernel invocation.

For every block, its tile is loaded and sorted in the on-chip memory, and then each block writes its histogram and the stored data tile to global memory. Then, a prefix sum is performed over the histogram table; it is stored in column-major order, for computing global digit offsets. The prefix sum results are used by each block for copying the elements to their correct output position (Satish et al., 2009); (Dusseau et al., 1996).

4. EXPERIMENTAL RESULTS

In the following, I describe several experimental results regarding the implemented sorting algorithm. The performance tests sorted sequences of 32-bit words key-value pairs, this length being suitable for building irregular data structures and sorting points. The keys were produced using a uniform random number generator.

I recorded the execution time that also contains the necessary amount of time for transferring input data from the host memory to the GPU's on-board memory across the peripheral component interconnect express bus. The algorithm described in this article is designed so that it scales across a significant range of parallel hardware architectures that implement the Compute Unified Device Architecture. This scaling is illustrated through the performance measured on the GeForce GTX285 and the GeForce GTX580 NVIDIA GPUs, whose technical specifications are described in Table 1.

One important aspect is to choose the right data partitioning in order to ensure enough workload for the complete utilization of the available hardware parallel resources. I have computed the number of key-values pairs sorted per second by dividing the input size by total running time. I have also adopted the technique developed in (Satish et al., 2009) and evaluated how the sorting performance was influenced by using different key distributions. The chosen sizes for the number of random bits generated for obtaining the keys were 8, 16, 24 or 32 bits.

In Figure 2 is represented the radix sorting time obtained for different sequence sizes on GTX285 and GTX580.

NVIDIA GPU	GTX285	GTX580
Release Year	2009	2010
CUDA Cores	240	512
Graphics Clock (MHz)	648 MHz	772 MHz
Processor Clock (MHz)	1476 MHz	1544 MHz
Texture Fill Rate (billion/sec)	51.8	49.4
Memory Clock (MHz)	1242	2004
Standard Memory Config	1 GB GDDR3	1536 MB GDDR5
Memory Interface Width	512-bit	384-bit
Memory Bandwidth (GB/sec)	159.0	192.4
Fabrication Process	55 nm	40 nm
Number of Transistors	1.4 billion	3.0 billion
Streaming Multiprocessors (SM)	30	16
Streaming Processors (per SM)	8	32

Tab. 1. Technical specifications of GTX285 and GTX580

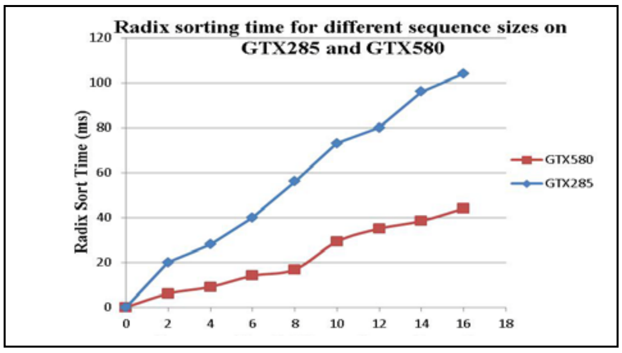

Fig. 2. Radix sorting time for different sequence sizes on GTX285 and GTX580

5. CONCLUSIONS

One particular interest in this article was to research how well the optimization techniques available in the literature for data sorting algorithms written in CUDA, scale to the latest generation of general-purpose graphics processors units (GPGPU), like the Fermi architecture implemented in the GTX580 and the previous architecture implemented in GTX285. Lately, there has been a lot of interest in the literature for optimizing data sorting algorithms using CUDA. None of those works so far (to my best knowledge) tried to validate if the optimization techniques can apply to the GTX580 (the high-end GPU of the Fermi architecture) and how well does the Fermi architecture scale to these data sorting algorithms performance improving techniques.

The obtained numerical results have confirmed and revealed some important aspects regarding the studied problem. The best way to obtain algorithmic efficiency on the Fermi architecture is to use the fast on-chip memory of the GPU and to employ fine-grained parallelism in order to benefit from the computing power of thousands of parallel threads offered by this architecture. Fast memory spaces have a significant influence on the overall performance and must be properly managed. These techniques, along with the increased level of parallelism provided by the 512 CUDA processing cores of the GTX580, have determined significant improved execution times. Future work involves a more thorough optimization using a larger selection of CUDA applications and an exhaustive benchmarking process. I intend to analyze how the new CUDA architecture can optimize and parallelize other types of sorting algorithms.

6. REFERENCES

Cormen T. H.; Leiserson C. E.; Rivest R. L. & Stein C. (2001). *Introduction to Algorithms*, MIT Press, ISBN-10: 0-262-03293-7, New York

Dean J.; Ghemawat S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, Vol. 51, No. 1, (ianuary 2008) pp. 107–113, ISSN 0001-0782

Dusseau A. C.; Culler D. E.; Schauser K. E.; Martin R. P. (1996). Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Trans. Parallel Distrib. Syst.*, Vol. 7, No. 8, (august 1996) pp. 791–805, ISSN 1045-9219

Nickolls J.; Buck I.; Garland M.; Skadron K. (2008). Scalable parallel programming with CUDA, *ACM Queue*, Vol. 6, No. 2, (march/april 2008) pp. 40–53, ISSN 1542-7730

Satish N.; Harris M.; Garland M. (2009). Designing efficient sorting algorithms for manycore GPUs, *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 23-29 may 2009, Rome, ISSN: 1530-2075, ISBN: 978-1-4244-3751-1, Satish, N. (Ed.), pp. 1-10, IEEE Publisher, Washington