PERFORMANCE IMPROVEMENT OF PARALLEL ALGORITHMS EXECUTED ON A GPU

Irena Skrceska, Fakulty of Informatics, European University – Republic of Macedonia

Abstract

A large number of algorithms that encompass intensive calculations are basically parallel algorithms or such that can be parallelized. GPU devices (Graphical Processing Units) which were originally designed for graphics applications, are massively parallel structures that have an enormous potential in massive parallel processing. With a leader in technology such as Nvidia in the frontline, the approach for using General Purpose computation on Graphics Processing Units (GPGPU) which combines the need to process intensive computations with high performance (High Performance Computing, HPC) with the characteristics of GPU devices, has rapidly gained ground compared to other methods for parallel programming. The present paper provides experimental results of the impact of certain optimization techniques on the working performance of the algorithm for matrix-matrix multiplication. Experiments were done on a GPU device GeForce GTX480. A number of optimization techniques were applied on the initial parallel algorithm, and a speed up of 3706x on the kernel function was achieved.

1. Introduction

The fact that GPU devices offer great potential for a fairly low price makes investigating the ways to exploit it attractive for research. The execution of data parallel algorithms with simple execution flow and a variety of arithmetic operations can be moved from CPU to GPU, which would result in significant performance improvement for sufficiently large data set.

With the introduction of the Compute Unified Device Architecture (CUDA) by Nvidia in 2006 [01, 02], it became possible for GPU devices to be used for general purpose computation, General Purpose computation on Graphics Processing Units (GPGPU) [01, 03, 04].

The CUDA architecture was developed based on the following objectives related to its design:

- To provide a small extension set over standard programming languages, such as the C programming language, which will contribute to a simple implementation of parallel algorithms. By using CUDA and CUDA C, developers will be focused on parallelizing algorithms, rather than on implementation.
- To support heterogeneous processing, so that applications will use both CPU and GPU. The serial part of the application would be executed on CPU, while the parallel part in the form of kernel functions would be performed on the GPU. CPU and GPU are two different devices which have their own memory space. This configuration allows for both the CPU and GPU device to perform calculations simultaneously, without memory resources collision.

The initial CUDA parallel implementation for algorithms suitable for parallelization is relatively simple, but making the most of GPU’s capacity is not a simple task. Utilization of the GPU capabilities is a complex task that depends on the various characteristics of the GPU device, the dimensions of the multidimensional structure of threads defined by the configuration parameters by calling the kernel function and the implementation of the algorithm, which should implement adequate optimization techniques suitable with the GPU execution model.

This research study evaluates the efficiency of several well-known optimization techniques such as the tiling technique, the data pre-fetching technique, and the loop unrolling technique that are often elaborated in the CUDA literature [05, 06, 07], as well as two new optimization techniques suitable for CUDA parallel algorithms, a technique that changes block granularity and a technique that changes thread granularity.

2. Scalable programming model

By calling the kernel function, the CUDA execution system genetartes appropriate grid of threads. Each thread will execute the code of the kernel function individually. Grid dimensions are defined by the execution configuration parameters specified when the kernel function is called. The threads within the grid are organized in a two levels hierarchy. In the top (first) level, the grid is composed of one or more blocks, while in the lower (second) level each block consists of several threads.

As all threads of the grid perform the same code, the CUDA programming is an example of the well known

SPMD (Single-Program, Multiple-Data) model of parallel programming [08]. Although in SPMD parallel processing units perform the same program on different data, at specific point of time not all processing units have to perform the same instruction, which is typical for the SIMD (Single-Instruction, Multiple-Data) model [09, 10].

The multi-dimensional hierarchical structure of threads will be mapped in the hierarchy of processors within the GPU device, which performs the execution of threads. Thus, the GPU device executes one or more grids, SM (Streaming Multiprocessor) executes one or more blocks, and CUDA cores execute the threads. SM executes groups of 32 threads (warp).

Since SM performs one instruction for all the threads within a warp at the same time, the SM employs a SIMT (Single-Instruction, Multiple-Thread) architecture [09]. The price for fetching and instruction processing will be amortized by the large number of threads that are performing the instruction.

3. Performance improvement of CUDA applications

The capacities of the GTX480 GPU device will be analyzed on a linear algebra matrix-matrix multiplication algorithm. This algorithm has simple execution flow and large number of arithmetic operations on loaded data. When the matrix dimensions are large enough, the parallel implementation of the algorithm which will be executed on GPU will provide high scale of data parallelism. This research study uses 1024x1024 matrices. Apart from the initial naive parallel CUDA implementation, a number of optimization techniques relevant to both CUDA architecture and parallel algorithms are applied to the initial implementation. The optimization techniques examined in this study are: a technique that uses shared memory and the tiling technique, a technique that changes the block granularity, a technique that changes the thread granularity, the data pre-fetching technique and the loop unrolling technique.

The efficiency of the individual optimization technique will be measured by the achieved speed up of the kernel function and the number of executed floating point operations per second during kernel execution [GFLOPS]. In order to calculate the speed up of the kernel function, the total execution time of the parallel kernel function should include: data transfer from the host to the device memory, the actual kernel function execution time, and data transfer from the device memory to the host, while calculating the number of executed floating point operations per second during kernel execution is based on the actual kernel function execution time.

The efficiency of the initial naive CUDA parallel algorithm for matrix-matrix multiplication is given in Table 1. Although the initial naive parallel algorithm provides significant speed up of 1241x compared with the serial version of the algorithm, the achieved performance is far from the peak performance of the mentioned GPU device. The main reasons for such a poor performance are limitation of the memory bandwidth and the large global memory latency. This could be overcome by introducing localization in data access by using shared memory, which will reduce the number of global memory accesses. Another limitation is the capacity of the shared memory, which would be overcome by using the tiling technique.

Algorithm	Execution time of kernel function [ms]	Achieved performance [GFLOPS]	Kernel function speed up [x]
MM00	30,37146	70,70729	1241,20377

Table 1. Efficiency of the naive parallel algorithm

Thus, the naive parallel algorithm is a starting algorithm to which subsequently a number of optimization techniques will be applied.

3.1 Tiling technique

Due to the limitation of the shared memory capacity, the initial naive parallel algorithm has to be adapted using the tiling technique which will incorporate partial calculations in several phases performed on a part of the data set i.e. a tile, that can be stored in the shared memory. This technique will reduce the number of global memory accesses, and will take the advantage of the shared memory regarding the fast data access.

In the matrix-matrix multiplication algorithm, each thread performs dot product calculation in an N/WTILE phases, N being the matrix dimension, and WTILE the tile dimension. In each phase, each thread within a block will copy one element from the matrix A located in the global memory to a tile As located in the shared memory, and likewise, one element from the matrix B located in the global memory to a tile Bs located in the shared memory. When the tile dimension corresponds to the block dimension and is equal to 16x16, the number of global memory accesses will be reduced by a factor of 16.

Algorithm	Execution time of kernel function [ms]	Achieved performance [GFLOPS]	Kernel function speed up [x]
MM00	30,37146	70,70729	1241,20377
MM01	10,03508	213,99748	2559,39281

Table 2. Efficiency of the tiling algorithm

Tiling technique leads to a function speed up of 2559x compared with the sequential function execution time. The efficiency of the tiling technique is given in Table 2.

The basic requirement for introducing the tiling technique is the possibility for independent execution of the kernel function on data set segments. It should be noted that not all data sets in individual kernel functions can be tiled, so the tiling techniques not always can be applied.

3.2 Technique that changes the block granularity

The simplest example of tiling technique is when the tile and block dimensions are the same, used in the previous section. The analysis of the global memory access reveals a certain redundancy in the activity of the threads which belong to the neighboring blocks; it appears that they copy the same data from the global to the shared memory. The redundancy can be traced in the activities of the threads within the same block too; they read the same data from the shared memory.

Since the neighboring blocks which have the same value for by use the same tiles As from the A matrix, their redundant reading by two, four or eight blocks can be eliminated given that the threads from one block calculate elements from two, four or eight blocks with the same value for by. The blocks that have the same value for bx use the same Bs tiles from the B matrix, so if the threads from one block compute the elements from two, four or eight blocks that have the same value bx, the redundant reading by two, four or eight blocks can also be eliminated.

The changes in the initial naive parallel algorithm will address the block granularity, or the workload that the block has to perform. By increasing the workload that the block has to perform, data reusability in shared memory is also increased; however, the number of blocks within the grid or the number of threads that execute the kernel functions is decreased.

If the threads from one block perform the usual calculations from eight blocks, or if the block reads one As tile from the A matrix and eight Bs tiles from the B matrix (which have the same value by) and perform the calculation for eight tiles of the resulting matrix (MM02a), or if the block reads one Bs tile from the B matrix and eight As tiles from the A matrix (which have the same value bx) and performs the calculation for eight tiles of the resulting matrix (MM02b), then the global memory access is reduced by 7/16.

The efficiency of the algorithm that changes the block granularity so that one block performs the workload of eight blocks with block dimensions of 16x16 in a matrix with dimension 1024x1024 is given in Table 3.

Algorithm	Execution time of kernel function [ms]	Achieved performance [GFLOPS]	Kernel function speed up [x]
MM00	30,37146	70,70729	1241,20377
MM01	10,03508	213,99748	2559,39281
MM02a	6,01385	357,08930	3221,13731
MM02b	5,51768	389,20009	3350,28418

Table 3. Efficiency of the algorithm that changes the block granularity

3.3 Technique that changes thread granularity

Redundancy can also be considered within the block. Threads which belong to one block and have the same value for tx read the same data from the Bs tile located in the shared memory. If one thread computes the elements from WTILE threads which have the same value for tx, the redundant reading by WTILE threads will be eliminated. The threads which belong to one block and have the same value for ty read the same data from the As tile also located in the shared memory. If one thread computes the elements from WTILE threads that have the same value for ty, the redundant reading by WTILE threads will be eliminated.

In this case, the changes in the initial tiling algorithm will address the granularity of the thread within the block, or the workload that the thread has to perform. By increasing the workloads of the thread, data reusability in shared memory as well as data reusability in registers is also increased.

The efficiency of the algorithm that changes the granularity of the thread so that one thread performs the workload of WTILE = 16 threads with the same value tx, block dimensions 16x8, BLOCK_SIZE_X = 16, BLOCK_SIZE_Y = 8 and matrix dimension 1024x1024 is given in Table 4.

Algorithm	Execution time of kernel function [ms]	Achieved performance [GFLOPS]	Kernel function speed up [x]
MM00	30,37146	70,70729	1241,20377
MM01	10,03508	213,99748	2559,39281
MM02a	6,01385	357,08930	3221,13731
MM02b	5,51768	389,20009	3350,28418
MM03	4,52679	474,39410	3429,90761

Table 4. Efficiency of the algorithm that changes the thread granularity

The execution time of the kernel function in situations when one threads performs the workload of WTILE = 16 threads which have the same value of ty for various block dimensions, brings to significantly worse performance compared with the initial tiling algorithm MM01, mainly because of the cost of an uncoalesced global memory access when reading data from matrix A directly onto an automatic variable.

Generally, the changes in block/thread granularity will lead to decreasing the number of global memory accesses.

The limitation of the algorithm that changes block/thread granularity is that the new kernel function now uses more shared memory and/or registers, which can lead to a reduction in the number of active blocks that the SM can execute. Additionally, due to the reduction in the number of blocks within the grid by half, significant reduction of the parallelization level can occur, especially in the case of matrices with smaller dimensions.

3.4 Data pre-fetching technique and loop unrolling technique

The tolerance of the large global memory **latency** in CUDA programming model is a result of the execution system's ability to select warps which are ready for execution instead of warps which must wait for the result of a previously initiated long-latency operation such as global memory access. Unfortunately, there is a possibility for all warps to be waiting for a long-latency operation. In such a case a performance improvement can be achieved by using the instructions to prepare the following data while threads are using current data (already read in the previous step), a technique known as a data pre-fetching technique. The pre-fetching technique will increase the number of independent instructions between memory access instructions and the consumer instructions of the accessed data.

As far as the above mentioned tiling technique is concerned, each thread executes the loop cycle given in Code1. Within this loop cycle, apart from the floating point instructions, there is an extra instruction for updating loop counter (k), an instruction for conditional branching at the end of each iteration, and an extra address arithmetic instruction which uses the loop counter (k) to index Ads and Bds.

```
for (int k=0; k<WTILE; k++)
sum+=a_ds[ty][k]*b_ds[k][tx];
```

Code 1. Simple loop cycle

It can be considered that only one third of the executed instructions are floating point calculation instructions, which limits the achievable performance. This instruction mix which consumes part of the instruction processing bandwidth can be significantly changed by unrolling the loop cycle. The loop unrolling will reduce (or entirely eliminate, depending of the unrolling level) the number of instructions for updating the loop counter (k), the instructions for conditional branching and address arithmetic instructions, since the indices are constant.

The integration of these two techniques, the data pre-fetching technique and loop unrolling technique (one level of unrolling and pre-fetching two elements from the B matrix), into the algorithm that changes the thread granularity (MM03) provides significant performance improvement (MM04). This is shown in Table 5 which displays the efficiency of the algorithm.

Algorithm	Execution time of kernel function [ms]	Achieved performance [GFLOPS]	Kernel function speed up [x]
MM00	30,37146	70,70729	1241,20377
MM01	10,03508	213,99748	2559,39281
MM02a	6,01385	357,08930	3221,13731
MM02b	5,51768	389,20009	3350,28418
MM03	4,52679	474,39410	3429,90761
MM04	4,09799	524,03272	3706,34096

Table 5. Efficiency of the algorithm that integrates the technique that changes thread granularity, the data pre-fetching technique, and the loop unrolling technique.

4. Conclusion

The presented results in this study show that GPGPU provides an opportunity for a significant performance improvement or a decrease of the execution time of parallelizable algorithms.

The initial transformation from serial but parallelizable algorithm into a parallel version of the algorithm which will be executed on a heterogeneous CPU/GPU system is relatively straightforward, but the achieved speed up usually is not satisfactory. The initial transformation usually does not exploit the resources and processing capabilities of the GPU device. For an efficient parallel CUDA algorithm, the GPU characteristics and its memory hierarchy need to be considered in order to apply a set of appropriate optimization techniques.

The initial naive parallel algorithm for the problem of matrix-matrix multiplication when the matrices dimensions are 1024x1024 provides a function speed up of 1241x. The improved algorithm that uses shared memory and tiling technique gives a function speed up of 2559x. By applying the technique that changes the block granularity, a function speed up of 3350x is achieved, and by applying the technique that changes the thread granularity, the result is a function speed up of 3429x. Further improvement of the algorithm with the data pre-fetching and loop unrolling leads to a function speed up of 3706x.

The programmer should properly choose the memory location where the data will be stored based on the algorithm that needs to be parallelized, the characteristics of the GPU device, and the CUDA memory hierarchy.

References

[01] David B. Kirk, Wen-mei W. Hwu; **Programming Massively Parallel Processors;** Elsevier; 2010; ISBN: 978-0-12-381472-2

[02] NVIDIA Corporation; **NVIDIA CUDA C Programming Guide** Version 4.0; March 2011

[03] NVIDIA Corporation; **NVIDIA CUDA C Programming Guide** Version 4.0; March 2011

[04] NVIDIA Corporation; **NVIDIA Best Practice Guide** Version 4.0; March 2011

[05] Chang Xu, Steven R. Kirk, Samantha Jenkins; **Tiling for Performance Tuning on Different Models of GPUs**; Pages: 500 – 504; IEEE Explore, December, 2009; ISBN: 978-1-4244-6325-1

[06] Tushar Athawale, Xie Xu; **Optimization Techniques for CUDA application**; Department of Computer and Information Science and Engineering, University of Florida; 2012

[07] Yuri Torres, Arturo Gonzalez-Escribano, Diego R. Llanos; **Understanding the Impact of CUDA Tuning Techiques for Fermi**; in Int.Conf. on High Performance Computing and Simulation, HPCS 2011; 2011; pages:631-639

[08] Laurent Badduel, Francoise Baude, Denis Caromel, “**Object-Oriented SPMD**”, Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid, Volume 2, Pages: 824-831, 2005

[09] John Nickolls, Ian Buck, Michael Garland, Kevin Skadron; **Scalable Parallel Programming with CUDA**; ACM Queue-GPU Computing; Volume 6 Issue 2, Pages: 40-53; March/April 2008

[10] Mahmoud Hassaballah, Saleh Omran, Youssef B. Mahdy; **A review of SIMD Multimedia Eztensions and their Usage in Scientific and Engineering Applications**; The Computer Journal, Volume 51, Issue 6, Pages:630-649; November 2008