ACCELERATING VECTOR CALCULATIONS ON GPU

Karol Grondžák[a)], Penka Martincová[a)], Ondrej Šuch[b)]

[a)] Faculty of Computer Science and Management, Žilina, [b)] Faculty of Natural Sciences, University of Matej Bel, Banská Bystrica

***Abstract**: Multicore computational accelerators such as Graphics Processor Units (GPUs) became common for gaining high-performance computing on a larger scale. Programming GPUs requires detailed knowledge of the underlying architecture in order to get maximum performance. In this paper we present solution of vector distance calculation on NVIDIA's parallel computing architecture CUDA (Common Unified Device Architecture), where we optimize the performance of a parallel algorithm and get significant speedup.*

***Keywords**: Vector Calculations, GPU, CUDA, Parallel Programming.*

***JEL Classification**: C61.*

1. Introduction

Current trends on high performance computing are moving towards the deployment of several cores on the same chip of modern processors in order to achieve substantial execution speedup through the extraction of the potential fine-grain parallelism of applications. At the forefront of this trend we find nowadays the modern Graphics Processor Units (GPUs), which due to their simplistic design are able to encompass hundreds of independent processing units on a single chip in contrast to their respective CPUs, which at the moment include only a few cores on the same chip.

CUDA-enabled [10] GPUs are SIMT (Single Instruction Multiple Threads) architectures and provide stream processing capabilities allowing the programmer to execute the parallel portion of the code on GPU devices. CUDA exposes three special programming abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization.

Programmers use these abstractions by dividing the program into coarse-grain sub-problems that can be executed independently in parallel. These sub-problems are further divided into finer-grain slices, which can also be solved cooperatively in parallel. This arrangement leverages one of the key benefits of threads: enabling them to cooperate with each other while solving individual sub-problems.

In recent years, a large amount of work has explored how to use GPUs for general purpose computing, sometimes known as "GPGPU" (General-Purpose Computation on GPU). Before the advent of general purpose languages for GPGPU, GPU implementations could only be achieved using existing 3D-rendering APIs: OpenGL [4] or DirectX [9]. The syntax, the need to pose problems in the context of polygon rasterization, and the limits imposed by pixel independence all made this approach cumbersome. Independently from GPU vendor efforts, several new languages or APIs were created to provide a general-purpose interface and abstract

away the necessary 3D API calls. Brook [1], Sh [8] and its commercial successor RapidMind, and Microsoft's Accelerator [13] are notable examples.

Recognizing the value of GPUs for general-purpose computing, GPU vendors added driver and hardware support to use the highly parallel hardware of the GPU without the need for computation to proceed through the entire graphics pipeline (transforming vertices, rasterization, etc.) and without the need to use 3D APIs at all. NVIDIA's solution is CUDA language, an extension to C. AMD's solution was the combination of a low-level interface, the Compute Abstraction Layer (CAL) and extensions to Brook.

A wide variety of applications have achieved dramatic speedups with GPGPU implementations. A framework for solving linear algebra problems on graphics processors is presented by Krüger et al. [6]. Harris et al. present a cloud dynamics simulation using partial differential equations [3], and molecular dynamics simulations (e.g. [11]) have also shown impressive speedups. Some important database operations have also been implemented on the GPU by using pixel engines [2], and a variety of other applications, such as sequence alignment [12] have been successfully implemented on GPUs.

The rest of paper is organized as follows: in chapter 2 we describe CUDA architecture and programming model, and principle of vector distance calculation. In chapter 3 we present the possibility of accelerating vector calculations using GPU. Three different approaches are presented and the obtained results are compared. We demonstrate step-by-step optimization of the sequential algorithm applying the knowledge of modern GPU architecture. In conclusion the gained results are summarized.

2. Statement of a problem

2.1 CUDA architecture and programming model

Computing system, which uses CUDA consists of a host that is a traditional CPU and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, there are often program sections that exhibit rich amount of data parallelism, a property where many arithmetic operations can be safely performed on program data structures in a simultaneous manner. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism.

CUDA programming model is shown at Fig.1. Code executed by host (CPU) is written in ANSI C and after compiling runs as standard process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called ***kernels***, and their associated data structures.

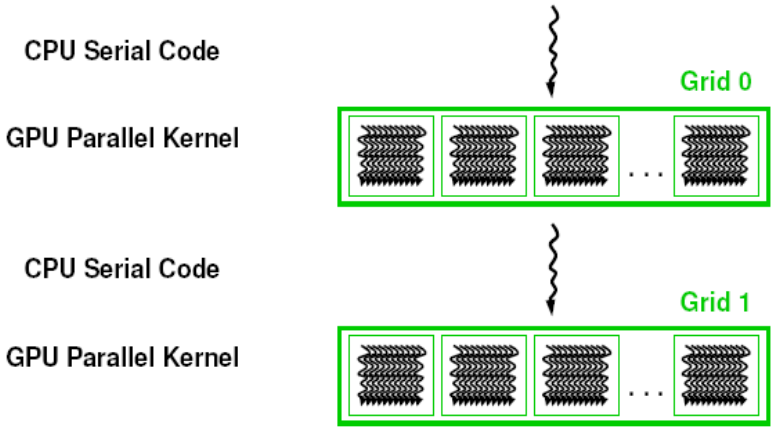

Fig.1: CUDA Programming model

Source: [5]

2.2 Vector Distance Calculation

One of the problems solved in linear algebra packages is the problem of vector distance calculation. Let us consider two matrices $\mathbf{A},\mathbf{B}$ of dimensions $M \times N, K \times N$ respectively, representing a list of N-dimensional vectors. We need to calculate distances of the vectors in the matrix $\mathbf{A}$ from vectors in the matrix $\mathbf{B}$. Resulting matrix $\mathbf{C}$ is of dimension $M \times K$ and its element $c_{i,j}$ represents the distance of vector in row i of matrix $\mathbf{A}$ from vector in row j of matrix $\mathbf{B}$, for $i = 1,2,\ldots,M$ and $j = 1,2,\ldots,K$. To calculate all elements of resulting matrix $\mathbf{C}$, sequential algorithm (Fig. 2) can be applied.

```
for (i = 0; i < M; i++) {
  for (j = 0; j < K; j++) {
    sum := 0
    for (k = 0; k < N; k++)
      sum += (A[i,k] – B[j,k]) * (A[i,k] – B[j,k]);
    C[i,j] = sum;
  }
}
```

Fig. 2: Sequential algorithm for calculation the distances among vectors

Source: (Authors)

It is quite obvious, that particular iterations are independent. Such kind of problem is suitable for parallel processing. To speedup the calculation of matrix $\mathbf{C}$, its elements can be determined in parallel. For matrices which fit into main memory of computer, the shared memory parallel architecture is suitable. If such system contains p processors, we can expect linear speedup of up to p times.

In case when matrices do not fit into main memory, some distributed memory architecture (e.g. MPI, grid or cloud technology) can be considered. In such case the problem would be divided and distributed among p processors, each processing only part of the problem. In the end the partial results would be combined into final result. This approach can suffer from overhead needed to distribute data among processors and collecting results.

3. Problem solving

In this chapter we present the possibility of accelerating vector calculations using Graphical Processing Unit (GPU). Three different approaches will be introduced and the obtained results will be compared. We will demonstrate step-by-step optimization of the sequential algorithm (Fig. 2) applying the knowledge of modern GPU architecture.

Basic paradigm of using GPU for calculation is to divide the solved problem into smaller parts. Each part will then be solved using a thread executed on GPU. Before executing calculation of GPU, necessary data must be transferred from main memory of CPU to main memory of GPU. This introduces some overhead to the calculation.

To obtain some baseline for comparison of results, we have performed the distance calculations on square matrices (i.e. $M = N = K$) of different sizes applying sequential algorithm. Parallel calculation on multi-core CPU using OpenMP was also carried on. Experiments were performed on computer equipped with two quad-core processors Intel Xeon E5530 2.40GHz. Calculation times for different problem size are summarized in Tab. 1, column denoted CPU (sequential algorithm) and column denoted OpenMP (performed on all eight cores). GPU calculations were performed on the same computer, using NVIDIA Tesla C1060 card equipped with 4GB of main memory.

3.1 Naive approach

As was mentioned before, to use the GPU for calculation, one has to divide the solved problem into parts and then assign the parts to the particular threads. In many tutorials on CUDA, authors encourage to create as many threads as possible, and leave the scheduling of them to GPU. Applying this approach to the problem of vector distance calculation leads us to a simple modification of sequential algorithm. We create $M \times K$ threads, each thread calculating exactly one element $c_{i,j}$ of matrix $\mathbf{C}$. The code of each thread will perform only the inner-most loop of the sequential algorithm to calculate distance of two vectors. The modified code is on the Fig. 3. This implementation uses exclusively the main GPU memory.

Results obtained using this algorithm are summarized in Tab. 1, column denoted GPU1. As can be seen, the speedup is of maximum factor 3, which is not satisfactory. This algorithm performance is poor because of the slow access to the main memory of GPU.

```
int i, tmp;
int vysl = 0;

int row = blockIdx.y * blockDim.y +th hreadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;

for (i = 0; i < N; ++i) {
  tmp = (A[row][i] - B[col][i]);
  vysl += tmp * tmp;
}
C[row][col] = vysl;
```

Fig. 3: Code executed by threads on GPU - version 1

Source: (Authors)

3.2 Local memory usage

The detailed knowledge of the GPU architecture allowed us to improve the performance of the algorithm by using faster shared memory. Threads on GPU are organized into blocks. Each block contains memory for private data of threads and local shared memory. This is used to exchange data among threads.

NVIDIA GPUs contain shared memory, which is significantly faster than main memory, but is limited in size. Each block of threads has 8KB of shared memory available.

To use shared memory, the algorithm has to be modified. It consists of two different phases. During first phase data are transferred from main memory of GPU into shared memory of a block. In second phase these local data are used to calculate partial results. Modified algorithm is presented in Fig. 4. The principle is explained on Fig 5. Resulting matrix **C** is divided into square blocks of dimension BLOCK_SIZE. Each block is processed by $BLOCK\_SIZE^2$ threads, executed in parallel. Let us consider block of matrix **C** marked by gray color. To calculate elements in this block, we have to access the stripes of matrices **A**, **B** also marked by gray color. These stripes are divided into blocks of dimension BLOCK_SIZE. In the first step of the algorithm, first blocks of matrices **A**, **B** are transferred into shared memory and used to perform calculations. Then second blocks of both matrices are transferred and used for calculations. This process continues until all blocks are processed and final results are obtained.

Parameter BLOCK_SIZE is essential for optimization of the algorithm performance. Maximum number of threads executed in one block is limited by properties of particular GPU. Typically, for recent NVIDIA GPU families it is usually equal to 512. This determines maximum value of BLOCK_SIZE to be not larger than 22.

```
int m;
int i, tmp, vysl = 0;

int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.y;

__shared__ int sA[BLOCK_SIZE][BLOCK_SIZE];
__shared__ int sB[BLOCK_SIZE][BLOCK_SIZE];

for(m = 0; m < N / BLOCK_SIZE; m++) {
  sA[threadIdx.y][threadIdx.x] = A[row][m * BLOCK_SIZE + threadIdx.x];
  sB[threadIdx.y][threadIdx.x] =B[col][m * BLOCK_SIZE + threadIdx.x];
  __syncthreads();
  for (i = 0; i < BLOCK_SIZE; ++i) {
    tmp = (sA[threadIdx.y][i] - sB[threadIdx.x][i]);
    vysl += tmp * tmp;
  }
  __syncthreads();
}
C[row][col] = vysl;
```

Fig. 4: Code executed by threads on GPU - version 2

Source: (Authors)

Second constraint which limits the possible values of BLOCK_SIZE is warp size. Warp is the set of 32 threads which is executed simultaneously and represents a unit for threads planning. To have all warps full, number of threads in a block must be an integer multiple of warp size.

For our experiments the value of parameter BLOCK_SIZE was set to 16. This gives us 256 threads in block and eight full warps in each block.

In previous algorithm the execution of threads was completely independent. This does not hold in this algorithm. It is obvious that threads in a block depend on each other. Second phase of the algorithm cannot be performed before all the threads have transferred relevant part of data into shared memory. To guarantee this, some kind of synchronization among threads is required. To ensure that all the threads have transferred their respective data into shared memory, __syncthreads() call is used. It represents a barrier. The calculation continues after barrier only when all the threads have reached it.

Results obtained by this algorithm are summarized in the column GPU2 in Tab. 1. It can be seen, that speedup of the factor 30 was achieved, which is better comparing to the previous algorithm.

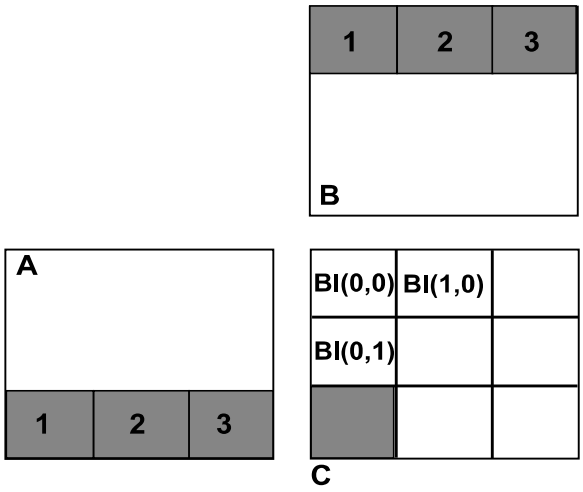

Fig 5 Principle of modified algorithm

Source: (Authors)

3.3 Minimizing bank conflicts

Third step in optimizing the performance of the parallel algorithm is to avoid bank conflicts. Shared memory is organized into banks. If two or more threads executed in the same warp access the same shared memory bank, conflict occurs. This conflict is solved by serializing the access to memory, which decreases the performance [10].

Tesla C1060 organizes shared memory into 16 banks. Brief inspection of the parallel code (Fig. 6) reveals that bank conflicts can occur. To avoid them, we can scatter data of the shared memory matrices among memory banks (Fig. 6).

Applying this modification, we were able to improve the performance of the algorithm. Obtained results are summarized in column GPU3 in Tab. 1. It can be seen, that speedup of a factor 90-100 was achieved.

Tab. 1: Summary of calculation times for different algorithms and problem size

Matrix dimension (M, K, N)	Execution time [ms]				
	CPU	OpenMP	GPU1	GPU2	GPU3
128	7.6339	4.7381	2.5	0.25	0.12
256	48.594	12.1866	29.47	1.27	0.51
512	324.39	54.868	350	9.15	4.3
1024	2551.6	372.86	1245	70	27
2048	21096	2819	6950	563	206
4096	168920	20580	53557	4505	1647

Source: (Authors)

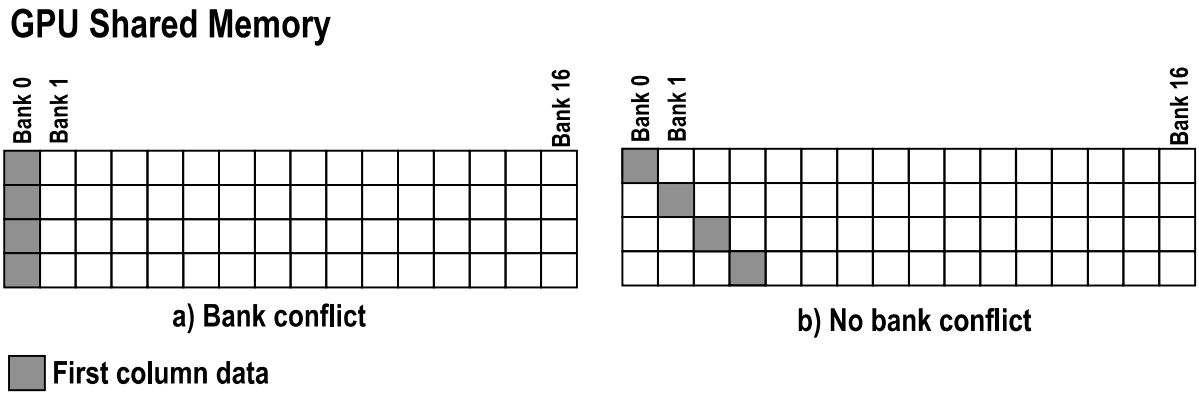

Fig. 6: Memory banks of shared memory

Source: (Authors)

4. Conclusion

In this contribution the possibility of vector calculations acceleration using GPU was presented. Recently the CUDA technology attracts the attention of researchers involved in numerical calculations. Massive parallelism achievable by this technology is used to speedup time-consuming calculations.

One of the areas, where application of massively parallel processor is applicable is the vector and matrix manipulation. This paper deals with simple vector calculation. First the serial algorithm run on CPU was presented. Next, the parallel version suitable to run on GPU was presented. Obtained results indicate, that naive transformation of the serial algorithm into parallel one does not lead to significant performance improvement. To achieve optimal performance, detailed knowledge of the underlying parallel technology is necessary. Vector distance calculation is essential in such areas as signal recognition, data clustering and data mining. Acceleration of the vector distance calculation can significantly improve the performance of these algorithms.

To motivate other scientists we have presented step-by-step procedure of optimization of the parallel algorithm. Final version of the parallel algorithm is approximately 90 – 100 times faster than serial one.

Acknowledgement

This work was supported by the operation research program Research and development for the project "Creating a new diagnostic algorithm for selected cancer diseases", ITMS 26220220022 co-financed from EU sources.

Európska únia
Európsky fond regionálneho rozvoja
Agentúra Ministerstva školstva, vedy, výskumu a športu SR pre štrukturálne fondy EÚ
Operačný program VÝSKUM a VÝVOJ

„We supported the research activities in Slovakia/Project is co-financed from EU sources"

References

[1] BUCK I., FOLEY T., HORN D., J. SUGERMAN, FATAHALIAN K, HOUSTON M. AND HANRAHAN P., *Brook for GPUs: Stream computing on graphics hardware*, ACM Transactions on Graphics **23** (3) (2004), pp. 777–786.

[2] GOVINDARAJU N. K., B. LLOYD, W. WANG, M. LIN, D. MANOCHA, *Fast computation of database operations using graphics processors,* in: Proc. of the ACM SIGMOD International Conference on Management of Data, 2004

[3] HARRIS M. J., BAXTER W.V., SCHEUERMANN T., LASTRA A.*, Simulation of cloud dynamics on graphics hardware*, in: Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2003

[4] KESSENICH J., BALDWIN D., ROST R, *The OpenGL shading language*, http://www.opengl.org/documentation/glsl

[5] KIRK B. D., HWU W.W, *Programming Massively Parallel Processors, A hands Approach*, ISBN: 978-0-12-381472-2 , Morgan Kaufmann Publishers is an imprint of Elsevier, 2010

[6] KRÜGER AND J., WESTERMANN R., *Linear algebra operators for GPU implementation of numerical algorithms*, ACM Transactions on Graphics **22** (3) (2003), pp. 908–916

[7] LINDHOLM E., NICKOLLS J., OBERMAN S. AND MONTRYM J., NVIDIA Tesla: *A unified graphics and computing architecture*, IEEE Micro **28** (2) (2008), pp. 39–55

[8] MCCOOL M. D., *Metaprogramming GPUs with Sh*, AK Peters (2004).

[9] Microsoft, DirectX 10, http://www.gamesforwindows.com/en-US/AboutGFW/Pages/DirectX10.aspx

[10] NVIDIA Corporation, *NVIDIA CUDA programming guide*, November 2007

[11] RODRIGUES C. I., HARDY D. J., STONE J. E., SCHULTEN K., HWU W.-M.W., *GPU acceleration of cutoff pair potentials for molecular modeling applications*, in: Proc. of the 2008 Conference on Computing Frontiers, 2008

[12] SCHATZ M., TRAPNELL C., DELCHER A. AND VARSHNEY A., *High-throughput sequence alignment using Graphics Processing Units*, BMC Bioinformatics **8** (1) (2007), p. 474

[13] TARDITI D., PURI S., OGLESBY J., *Accelerator: using data parallelism to program GPUs for general-purpose uses*, in: Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, 2006

Contact Address

Ing. Karol Grondžák, PhD.
Faculty of Computer Science and Management, University of Žilina
Univerzitna 1, 010 26 Zilina, Slovak Republic
E-mail: Karol.Grondzak@fri.uniza.sk
Phone number: +421-41-5134173

Doc. Ing. Penka Martincová, Ph.D.
Faculty of Computer Science and Management, University of Žilina
Univerzitna 1, 010 26 Zilina, Slovak Republic
E-mail: Penka.Martincova@fri.uniza.sk
Phone number: +421-41-5134150

Ondrej Šuch, Ph.D.
Faculty of Natural Sciences, University of Matej Bel, Banská Bystrica
Tajovského 40, 974 01 Banská Bystrica, Slovak Republic
E-mail: ondrej.such@umb.sk
Phone number: +421-48-446 7128