Fast Polynomial Approximation Acceleration on the GPU

Lumír Janošek
Department of Computer Science
VSB-Technical University of Ostrava
Ostrava, Czech Republic
Email: lumir.janosek.st@vsb.cz

Martin Němec
Department of Computer Science
VSB-Technical University of Ostrava
Ostrava, Czech Republic
Email: martin.nemec@vsb.cz

***Abstract*—This article presents the possibility of parallelization of calculating polynomial approximations with large data inputs on GPU using NVIDIA CUDA architecture. Parallel implementation on the GPU is compared to the single thread CPU implementation. Despite the enormous computing power of today's graphics cards there is still a problem with the speed of data transfer to GPU. The article is mainly focused on the implementation of some ways of transferring data from memory into GPU memory. The aim is to show what method is suitable for a large amount of data being processed and what for the lesser amount of data. Afterwards performance characteristics of the implementation of the CPU and GPU are matched.**

Keywords-GPU; CUDA; Direct Memory Access; Parallel Reduction; Approximation.

I. INTRODUCTION

This article is focused on the application of a parallel approach to the implementation of the polynomial approximation of the k-th degree and its comparison with conventional single thread approach. Polynomial approximation model is widely used in practice. The statistics commonly use the basic model of approximation of 1th degree - a linear approximation, in other statistics called the linear regression.

Nowadays it is possible to create a massively parallelized applications using modern GPUs (Graphics Processing Unit) that enable the distribution of calculations among tens of multiprocessors of graphic cards. The problem still remains the need to transfer data between the CPU (Central Processing Unit) and GPU. This can become a limiting factor in performance when the time needed to transfer data between memory and GPU memory, the host system plus the time the GPU processes data exceeds the time after, which the same data can be handled by the CPU. But there are ways to at least partially eliminate this lack of trying.

In this article will be shown how to implement polynomial approximation using the GPU parallel computing architecture of NVIDIA CUDA (Compute Unified Device Architecture), which provides a significant increase of computing power [1]. The parallel implementation is compared with single threaded CPU implementation. Performance results of both implementations are compared with each other and show the differences between the parallel implementation approach and common single threaded approach for certain volume of data. By comparison of these two approaches it can be seen for how much data is suitable for the parallel approach and for how much it is already inappropriate. A substantial part of the implementation is a comparison of the chosen methods of copying data from RAM (Random-Access Memory) to graphics card memory, and especially the methods of allocating this memory. Three methods are compared: the allocation of pageable memory, the allocation of page-locked memory (also known as Pinned memory), and the allocation of memory mapped into the address space of the CUDA [2].

A common approach is the method of allocation and data transfer, when the input data are placed in pageable memory and from this memory are then transmitted by conventional copying approach into graphics card memory. The allocation of page-locked memory when copying data allows the GPU to use DMA (Direct Memory Access). Mapping memory allocation into the address space of the CUDA is a special case that allows to read data stored in RAM directly from the GPU.

This paper is structured as follows. First, some mathematical background related to polynomial approximations is presented. Next, a description of the implemented memory approaches and a description of the implementation of a parallel reduction are presented. Lastly, results and conclusion are presented.

II. MATHEMATICAL MODEL OF APPROXIMATION

Consider a set of points with coordinates x_i z $\mathbb{R}^d$, where $i \in \{1, \cdots, n\}$. The aim of the approximation data problem is to find the function $f(x)$ in the general case, which best approximates the scalar value f_i at point x_i. The result, using the least squares method, is a function $f(x)$ such that the distance between scalar data values f_i and functional values $f(x_i)$ is as small as possible [3]. Least squares method based linear approximation in its simplest application, which approximates an input data by linear function in the form of:

$$f : b_0 + x \cdot b_1, \tag{1}$$

 ISBN: 978-1-61208-176-2

where the sum of squares has the form:

$$\psi(b_0, b_1) := \sum_{i=1}^{n} [f(x_i) - f_i]^2 \quad (2)$$

Minimum of sum of squares then we found as:

$$\frac{\partial \psi}{\partial b_0} = 0 \qquad \frac{\partial \psi}{\partial b_1} = 0$$

By adjusting the obtained:

$$\begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = \begin{pmatrix} n & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \end{pmatrix}^{-1} \begin{pmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \end{pmatrix} \quad (3)$$

Members of the vector of the right side b_0 and b_1 are coefficients of the polynomial approximation (1). Input data of the algorithm are represented by a set of vectors (pairs) of $\mathbb{R}^2$. For input data it is sufficient to calculate the four sums (vector of sums):

$$V_\Sigma = \left(\sum_{i=1}^{n} x_i, \sum_{i=1}^{n} y_i, \sum_{i=1}^{n} x_i y_i, \sum_{i=1}^{n} x_i^2 \right) \quad (4)$$

The results of these sums are then just put back into the system of equations (3). By solving it, we get the sought coefficients of b_0 and b_1 approximation polynomial (1).

A. Polynomial approximation

A special case of linear model approximation is polynomial approximation. It is an approximation by polynomial of k-th degree. Using the procedure for calculating the linear approximation it is possible to express polynomial approximations formula as a set of [4]:

$$b = A^{-1} Y \quad (5)$$

where

$$A = \begin{pmatrix} n & \sum_{i=1}^{n} x_i & \cdots & \sum_{i=1}^{n} x_i^k \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \cdots & \sum_{i=1}^{n} x_i^{k+1} \\ \cdots & \cdots & \cdots & \cdots \\ \sum_{i=1}^{n} x_i^k & \sum_{i=1}^{n} x_i^{k+1} & \cdots & \sum_{i=1}^{n} x_i^{2k} \end{pmatrix}$$

$$b = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_k \end{pmatrix}, Y = \begin{pmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \\ \cdots \\ \sum_{i=1}^{n} x_i^k y_i \end{pmatrix}$$

The solution of this system of equations is a vector b, the individual members of which $b_0 \cdots b_k$ represent the coefficients of the approximation polynomial. Using the system of equations (5) it is possible to derive a vector of sum for any polynomial k-th degree as in the case of linear approximation.

III. Implementation of memory approaches

The algorithm for calculating polynomial approximation, which was described in the previous section, was implemented using the CUDA architecture of NVIDIA. The implementation was designed for processing large amounts of data represented as a set of vectors of $\mathbb{R}^2$. Parallel implementation of polynomial approximations on GPU is compared with single threaded implementation on CPU. During implementation, the goal was due to the large amount of input data, to optimize data flow between RAM and GPU memory.

Given the size of the input data set, several approaches to copy data from RAM to graphics card memory (global memory), or to access data from the GPU were compared. Three approaches were compared: normal approach with pageable memory allocation, the allocation of page-locked memory (also known as Pinned memory), and the allocation of page-lock memory mapped into address space of the CUDA.

The actual calculation of the polynomial approximation was implemented in part on the GPU and in part on the CPU. For comparison approximations of 1th degree, 2th degree and 3th degree were implemented. A parallel approach was used for calculating the vector of sums (4) by using a the parallel reduction algorithm. Calculation of the resulting coefficient b_0 and b_1 of the approximation polynomial is then completed on the CPU.

A. Pageable system buffer and page-locked memory

A common approach to transfer data from RAM to the global memory was compared with direct access of the GPU to RAM when copying data, otherwise the DMA (Direct Memory Access). The disadvantage of the common approach is double copying of transferred data. Data are transmitted in the first step from pageable memory (pageable system buffer) to the page-locked memory, and then from this page-locked memory to the GPU memory. By direct allocation of a data buffer in the page-locked memory extra data copying can be avoided. By allocation page-locked memory the operating system guarantees that this memory is not paged to disk, thus ensuring its place in physical memory [5]. By knowing the physical address of the buffer in the memory, GPU can copy data to the global memory direct memory access - DMA.

B. Memory mapping into address space of the CUDA

Another approach to access data in the RAM from the GPU is using direct mapping of page-lock memory into address space of the CUDA. The data are, as in the previous case, stored in memory allocated as page-locked memory, with the only difference being that this data can be accessed directly from the GPU. This eliminates the need for allocating memory block in global memory and the need to copy data into this block of memory.

 ISBN: 978-1-61208-176-2

IV. PARALLEL REDUCTION

With access to parallel hardware the entire process of the sum calculation can be parallelized. If we have hundreds of threads, then each thread can contribute to the gradual calculation of the resulting sum. This approach is called parallel reduction [6]. The main idea of the parallel reduction is that each thread performs the sum of two values in the memory and then saved back. The algorithm therefore starts at the beginning with half the number of threads than the number of inputs. In each step, one thread adds the two values. In the next step the process is repeated, but with half the number of threads. The process continues until the final sum is achieved by gradual reduction. The parallel reduction algorithm is especially efficient for large data inputs.

Reduction of the vector (4) is divided between $C \cdot N_{threads}^{-1}$ blocks, where C is the count of input data (vectors of $\mathbb{R}^2$) and $N_{threads} = 256$ is the number of threads per block. The data are this way evenly divided between the individual blocks, when each block handles one subset of the input data.

Implementation of the parallel reduction of the vector of sums can be divided into three steps: 1) The first step is to copy data from global memory to the shared memory. In the shared memory the reduction of the vector of sums is subsequently made. Copying data from global memory to shared memory is implemented in the CUDA kernel by using all threads of the block, thus each thread copies always the four values that belong to one of the four sums of a vector (4). Simultaneously with copying the data into shared memory, is made the first reduction step - first add during load [6]. This leads to the reduction of the required number of blocks by half. The total number of blocks needed to run the CUDA kernel is

$$\frac{1}{2} \cdot \frac{C}{N_{threads}}$$

2) After copying the data into shared memory all the threads of block are synchronized, which ensures that no thread starts reading the shared memory until all threads finish copying the data. Then begins the process of reduction. In each iteration, one thread performs the sum of the vector of sums, which leads to a gradual reduction of input data. Before entering the next iteration, the number of threads is reduced by half. Reduction cycle ends when the number of threads reaches zero. The data inside the loop are processed in the shared memory (on-chip memory), accordingly there is no unnecessary transfer of data between global memory and GPU multiprocessor. 3) The result of each block is transferred back to global memory after the reduction. This copy process takes place before the end of kernel one of the threads. The results of the individual blocks are copied from global memory back to the RAM on the CPU. Completion of reduction, thus the sum of all results of individual blocks, is completed on the CPU. The result is a vector (4). It is then possible, without difficulty, to apply the described algorithm of a parallel reduction, with minor modifications, to the calculation of vectors of sums of approximations of higher degrees.

V. RESULTS

The presented method for parallel calculation of linear approximation to the GPU has been implemented and tested on a graphics card GeForce 9600 GT, GeForce 9800 GTX and GeForce GTS 450 NVIDIA. The implementation was tested for various sizes of the input file in order to determine what amount of data is preferable to compute on the CPU and for how much data it is more efficient to use a parallel implementation on GPUs. Performance characteristics of both implementations were compared, the result is shown in the Fig. 1. From a comparison of the characteristics of computations on the CPU and GPU it is obvious that for a smaller amount of data it is preferable to keep the calculation of polynomial approximation on the CPU. GPU in this case is more appropriate for larger data amounts. The size of test data ranged from 12 KB to 50 MB (1365 - 6400000 input data).

As written in the previous section, three approaches of transferring data between RAM and the global memory were compared. A common approach to copy data between RAM and global memory of GPU is compared with the approach of direct access of the GPU to RAM (DMA) when copying data. This method of implementation has brought strong effect especially in an expanding volume of copied data, as seen from the Figure 2 because there is no need to copy data from pageable memory into the page-locked memory, before transferring data to the GPU global memory.

The last of the studied approaches of transferring data from memory into the GPU global memory was the use of mapping page-lock memory into address space of the CUDA. Mapping page-lock memory is especially suitable for integrated graphics processors that are built into the system chipset and usually share their memory with the CPU. In this case, using the mapping page-lock memory removes unnecessary data transfers. For discrete graphics processors, the mapping page-lock memory is only suitable just in some cases [7]. For this reason, this method also did not bring any optimization of implementation. On the other contrary, when using the mapping page-lock memory into address space of the CUDA, there was a significant downgrade in performance, see Figure 3. Below are listed the size of data transfers that have occurred during the calculation between the CPU and GPU memory.

A total number of N bytes of data was transmitted into the global memory from RAM. After completion of the calculation on the GPU back to RAM was transmitted a total of

$$\|V_\Sigma\|_2 \cdot \left(\frac{NumBlocks}{2} \cdot 4_{bytes} \right)$$

 ISBN: 978-1-61208-176-2

The total number of bytes transferred from global memory back into the RAM is equal to:

$$\|V_\Sigma\|_2 \cdot \left(\frac{1}{2} \cdot \frac{N}{4_{bytes}} \cdot \frac{1}{NumThreads} \cdot 4_{bytes} \right)$$

$$\frac{1}{2} \cdot \|V_\Sigma\|_2 \cdot \frac{N}{NumThreads}$$

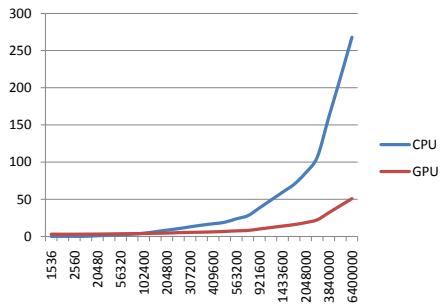

Figure 1. The speed of calculating a linear approximation for the input data (vectors) in milliseconds. Comparison of speed of calculation on the CPU and GPU.

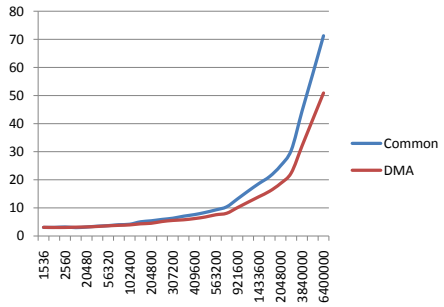

Figure 2. The speed of calculating a linear approximation for the input data (vectors) in milliseconds. Comparison of the effectiveness of implementation of calculation on the GPU using DMA access and common access to copy data to the global memory.

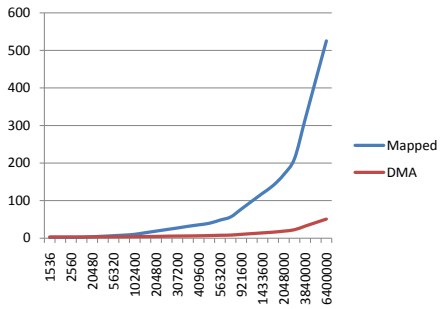

Figure 3. The speed of calculating a linear approximation for the input data (vectors) in milliseconds. Comparison of implementations using the mapping page-lock memory into the address space of the CUDA and approach using the DMA to copy data to the global memory.

VI. Conclusion

This article presented a parallel implementation of polynomial approximations on the GPU, which will significantly optimize the performance during calculation the large amounts of data. The implementation was compared with single thread CPU implementation, which is more suited for smaller data amount. It was shown that copying data from RAM, allocated as a page-lock memory, using the direct memory access (DMA), significantly accelerated the application performance in the result. In contrast, the use of mapping page-locked memory into address space of the CUDA in the implementation provided no improvement in application performance. This method is suitable for integrated GPU, which almost always produces a positive result due to the shared memory of CPU and GPU.

Acknowledgment

The thanks belongs to Professor Václav Skala for his substantive comments.

References

[1] NVIDIA Corporation, *CUDA ZONE*, http://developer.nvidia.com/, retrieved: December, 2011.

[2] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, Version 4.0, 2011.

[3] G. Coombe, *An Introduction to Scattered Data Approximation*, October 31, 2006.

[4] K. Rektorys, *Survey of Applicable Mathematics II*, 7th ed. Praha, 2003.

[5] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*, 1th ed. United States of America, 2011.

[6] M. Harris, *Optimizing CUDA*, SC, 2007.

[7] R. Farber, *CUDA, Supercomputing for the Masses*, May 14, 2009, http://drdobbs.com/high-performance-computing/217500110, retrieved: December, 2011.

 ISBN: 978-1-61208-176-2