COMPARISON OF RECTANGULAR MATRIX MULTIPLICATION WITH AND WITHOUT BORDER CONDITIONS

Petre Lameski Igor Mishkovski Sonja Filiposka Dimitar Trajanov Leonid Djinevski
Ss. Cyril and Methodius University in Skopje / FINKI FON University
Skopje, Macedonia Skopje, Macedonia

ABSTRACT

Matrix multiplication algorithms are very common and widely used for computation in almost any field. There are many implementations for matrix multiplication on different platforms and programming models. GPU devices in the recent years have become powerful computational units that have entered the segment of high performance computing. In this paper we are analysing two approaches for the matrix multiplication algorithm with and without border conditions for parallel GPU execution.

I. INTRODUCTION

Matrix multiplication is an operation that is widely used in different algorithms. For that reason the speed up and optimization of this operation may improve their performance. As a mathematical operation, matrix multiplication has been given a significant attention from the computer science community. The regular complexity of the "school" method for sqyare matrix multiplication is $O(n^3)$. Several other methods for matrix multiplication exist that reduce this complexity. These are the Strassen algorithm with complexity of $O(n^{2.807})$, the Coppersmith–Winograd algorithm with complexity of $O(n^{2.376})$ [1] and Williams algorithm $O(n^{2.373})$ [2]. For rectangular matrix multiplication ($n \times m$ and $p \times n$] the complexity is $O(nmp)$ The reduction in algorithm complexity introduce the speedup in matrix multiplication that increases overall performance for algorithms where matrix multiplication is used in significant portion of the operations.

Given the architecture of the computers used, there are existing attempts to speed up matrix multiplication using the specific computer architecture design [3]. The architecture specific characteristics of the computer are used to increase performance such as instruction level parallelism, tiling, avoiding cache conflicts, pre-fetching etc.
The optimizations of algebra operations has been addressed by the community and wide variety of libraries exist that try to optimize mathematical operations. Hardware vendors usually provide libraries for their specific architectures that use the architecture specific characteristics to improve performance on some operations. Such libraries are MKL and ACML. All of these libraries have the standard BLAS (Basic Linear Algebra Sub-programs) libraries [4] which performance greatly depends on the underlying architecture [5].

With the introduction of the parallel programming, especially with the introduction of GPU devices, there have been significant improvements of a wide variety of algorithms and processes that can use the benefits of parallelism. The main problem with the introduction of the parallel execution is that not all algorithms can be efficiently parallelized. The standard library that is introduced by NVIDIA, for all of their GPU architectures is CUBLAS that includes the matrix multiplication. CUBLAS optimizes the performance of matrix multiplication, however it is not very well documented what characteristics of the hardware are used for the introduced speedup. There have been attempts that successfully outperform CUBLAS by means of increasing the efficiency of the algorithm based on the specifics of certain architectures [6].

One of the things that can be seen in literature is that all optimizations of the algorithm for matrix multiplication is based on the underlying hardware architecture since it is always a good idea to use the benefits that the architectures have to offer in order to improve the performance.

In this paper we compare two approaches for the standard Matrix Multiplication Algorithm on custom sized rectangular matrices on GPU devices. The approaches differ in the size of memory transfer between the global memory and the device shared memory, and in the number of operations that are used in the matrix kernel calculation. The main motivation for this work is the comparison of the performance between memory transfer between shared and global memory and the branching inside the kernel function for matrix multiplication.

This paper is organized as follows: Section 2 present a short overview of General-Purpose computing on GPU devices (GPGPU). The matrix multiplication algorithms are defined in Section 3. The testing methodologies used are described in Section 4, followed by the obtained results in Section 5. We conclude this paper in Section 6.

II. GPU DEVICES

Today GPU devices are the most powerful computational processors for the cost at which are being sold [7]. The latest Kepler [8] architecture provides almost 3TFLOPS in the hands of a workstation PC, thus providing tremendous computational resources for the average user. Either the fact that GPUs were traditionally developed for graphics applications, with the help of enthusiasts, a significant research was performed for utilizing the computational resources for GPGPU (General-Purpose applications on GPU devices) [9]. Nvidia as one of the leading vendor for GPU processors, in 2007 released parallel computing platform and programming model for execution general-purpose applications, called CUDA (Compute Unified Device Architecture) [10]. Additionally, OpenCL was released as a

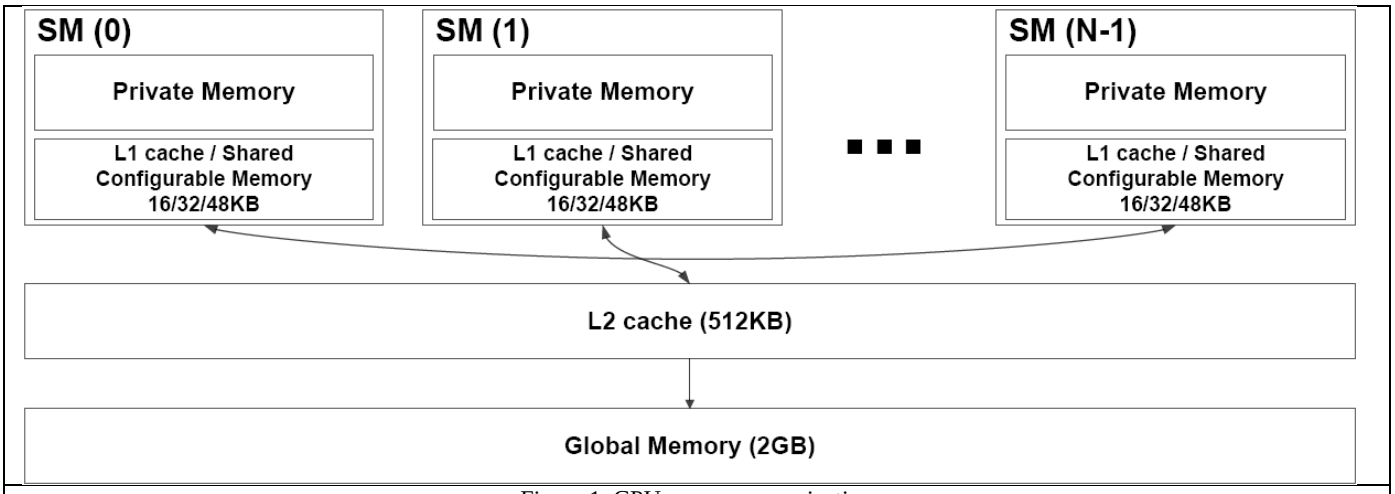

Figure 1. GPU memory organization

standard programming language, formed by the major vendors like Apple, Nvidia, AMD/ATI, Intel, and other companies in the industry [11]. OpenCL is very similar to CUDA, however it agnostic to any platform, and vendor independent, unlike CUDA which is bounded to Nvidia. OpenCL is based on ANSI-C99 extended with additional data types, qualifiers and build-in functions. Beside C/C++, CUDA on the other hand is available to more programming styles like fortran, java, phyton, perl, MATLAB and others, by adopting a wrapper of the native CUDA C/C++ compiler.

As one of the latest GPU devices from Nvidia, which has the latest Kepler architecture at the moment of writing this paper is the GTX 680 model. On Fig. 1 we present the memory organization of the GTX 680, which consists of 3 levels: L1 cache memory (configurable by the developer 16/32/48 KB), L2 cache memory that is of fix size of 512KB and the Global memory of 2GB.

III. MATRIX MATRIX MULTIPLICATION ALGORITHM

The regular matrix multiplication is done by multiplying each row of a matrix A with each column of a matrix B. Each element in the matrix C can be defined as:

$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} * B_{k,j} \qquad (1)$$
$$j \in [0, m], i \in [0, n]$$

In parallel multiplication of matrices on GPU, each result $C_{i,j}$ is calculated by a separate thread. Threads access the global memory, but threads in same block, access the very fast shared memory inside the GPU. The shared memory in the system used in this paper is consisted of blocks with size 32x32. The matrix is divided in these blocks and the calculation is done. The problem arises when the sizes of the matrix are not divisible by the block size. In this case the sum in (1) cannot be calculated in the same way since not all the memory entries in the block have a valid value. Another problem is that the allocated shared memory is also included in the calculation sum.

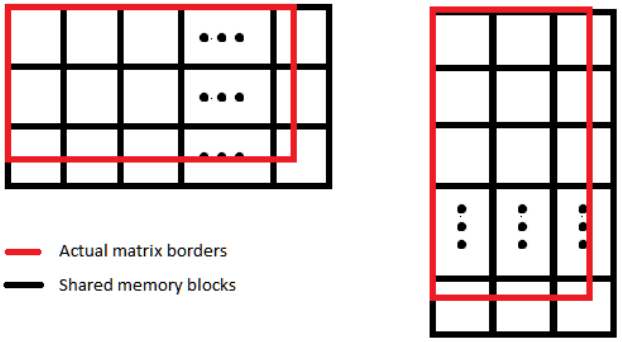

Figure 2. Shared memory blocks and Actual matrix borders

Inside the kernel function each block has two block IDs *bx* and *by* and each thread has also two IDs, *tx* and *ty*. Lets define the block to be the size of B. The data from the global memory is copied in the shared memory for higher performance of the calculation. Each cell of the matrix is copied in the shared memory such that *i=B*bx+tx* and *j=B*by+ty*. The border cases are when the values *B*bx+tx* or *B*by+ty* cross the values of the widths and heights of *A* and *B*. The difference between the allocated shared memory blocks and actual matrix sizes is given in Figure 2.

When the sizes of A and B are not divisible with B, the border blocks and threads do the calculation anyway and if they are not ignored explicitly. Because the memory content is not known the results of this calculation will be wrong. The border cases are a problem when calculating the product of two rectangular matrices with arbitrary dimensions using CUDA.

To solve the problem with the border cases in rectangular matrix multiplication we have two choices. The first choice is to copy only the data within the border of the matrices and deal with the border cases inside the kernel function. In this way we save on the memory transfer bandwidth that is used between the system memory and the device memory. The second choice is to increase the width and height of the matrices so that their dimensions are divisible with the used block size. The fields of the matrices that don't actually belong to them are initialized with zero values so that they don't interfere with the multiplication process. By increasing the dimensions of the matrices, the border conditions no longer exist and the kernel doesn't have the need of logical branching in the code.

The first approach is defined in Listing 1 as follows:

Listing 1:
Allocate Matrices A (n x m) and B (p x n);
If (m mod BLOCK_SIZE) !=0
  Set *new_m* to next multiple of BLOCK_SIZE >m;
If (n mod BLOCK_SIZE) !=0
  Set *new_n* to next multiple of BLOCK_SIZE >n;
If (p mod BLOCK_SIZE) !=0
  Set *new_p* to next multiple of BLOCK_SIZE >p;
Allocate Matrices NewA and NewB to new_n x new_m and new_p x new_n;
Initialize all fields of NewA and NewB to 0;
For i in [0:new_n] and j in [0:new_m]
Set NewA(i,j) = A(i,j) if i<n and j<m;
For i in [0:new_p] and j in [0:new_m]
Set NewB(i,j) = B(i,j) if i<n and j<m;
Do normal kernel multiplication of NewA and NewB on GPU such that NewC=NewA*NewB;
Get subMatrix C with dimensions m x p from NewC starting at (0,0);
Return C;

The first approach moves bigger matrices in kernel memory and does the calculation with them. The result will be the same since all the fields from the bigger matrices *NewA* and *NewB* that are not common with *A* and *B* are set to 0. In this way we get the same result as we would get if using formula (1) to multiply the matrices.

The second algorithm on the other hand uses only the initialized matrices *A* and *B* with their respectful sizes *n x m* and *p x n*. The border cases are resolved inside the kernel. Whenever the kernel multiplication reaches a border block that is not fully covered with values from the matrices, the shared memory for that block is set to 0. In this way we introduce code branching inside the kernel function. However since there are more threads inside the calculating blocks than needed, we also need to do additional check if the resulting matrix *C* is also inside the defined borders. The second algorithm can be defined as:

Algorithm 2:
Allocate Matrices A and B with dimension n x m and p x n;
Do modified kernel multiplication of A and B on GPU such that C=A*B;
Return C;

Modified kernel multiplication:
Set bx=BlockID.x,
by=BlockID.y,
tx=ThreadID.x,
ty=ThreadID.y;
S=0;
Declare SharedA and SharedB as shared memory matrices;
Do for all fields:
  if (bx and by are a border case)
  if(bx*BLOCK_SIZE+tx>weightA && by*BLOCK_SIZE > heightA)
 SharedA[tx,ty]=0;
  Else
    Load corresponding A element to SharedA
  if(bx*BLOCK_SIZE+tx>weightB && by*BLOCK_SIZE > heightB)
SharedB[tx,ty]=0;
  Else
    Set SharedB[tx,ty] to the corresponding B element
   Synchronize threads;
  Do the multiplication for the sub matrix;
 Synchronize threads;
 If(Current thread has legal result)//is within the borders
    Write block sub matrix to device memory (C);
 Return C;

IV. TESTING METHODOLOGIES

All tests were performed on the hardware infrastructure presented in Table 1. The Ubuntu 12.04 LTS was installed as an operating system, while the implementation was compiled with the Nvidia nvcc compiler that is part of the CUDA 5.0 toolkit.

Table 1:

Utilized hardware for performing all of the tests	
CPU	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
RAM	Kingston 32GB @ 1.60GHz
GPU	NVIDIA GeForce GTX 680

Both approaches were compared by measuring the speed performance of the CUDA matrix multiplication kernels for different sizes of the matrices. Since we are dealing with multiplication of two matrices, the width *wA* of the matrix *A* and the height *hB* of the matrix *B* should have the same value, therefore, in our experiments we are varying the height *hA* of matrix A and width *wB* of matrix B. For *wA = hB* we have chosen cases 32 and 64 in order to analyse both approaches.

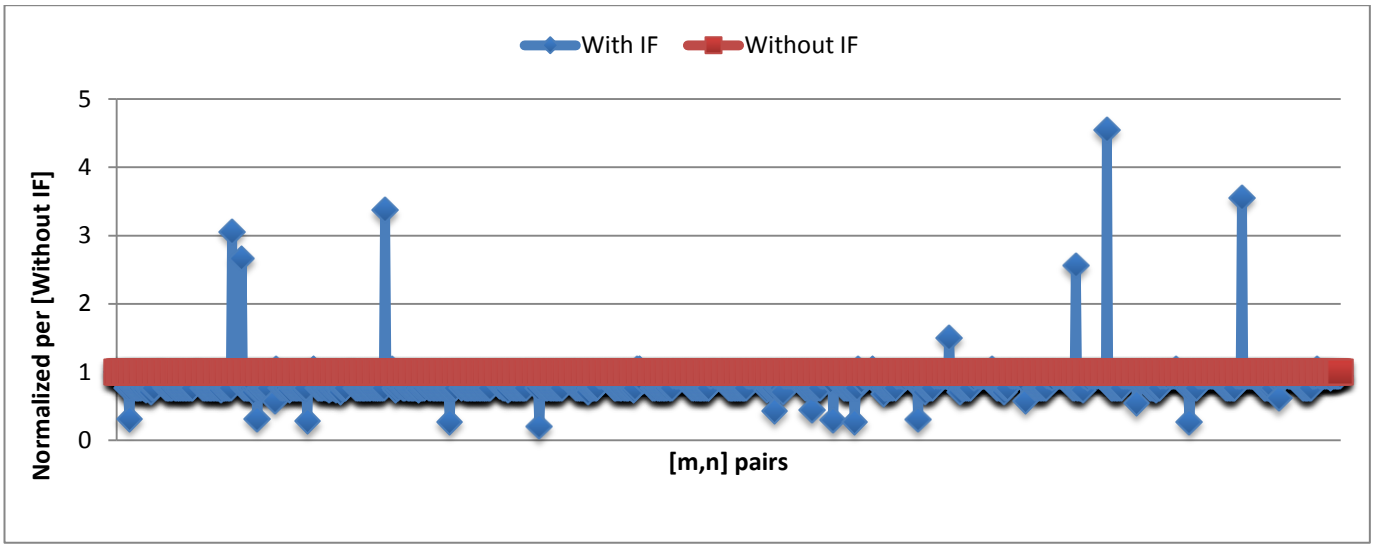

Figure 3 Normalized speed for both approaches for the case *wA = hB = 32*

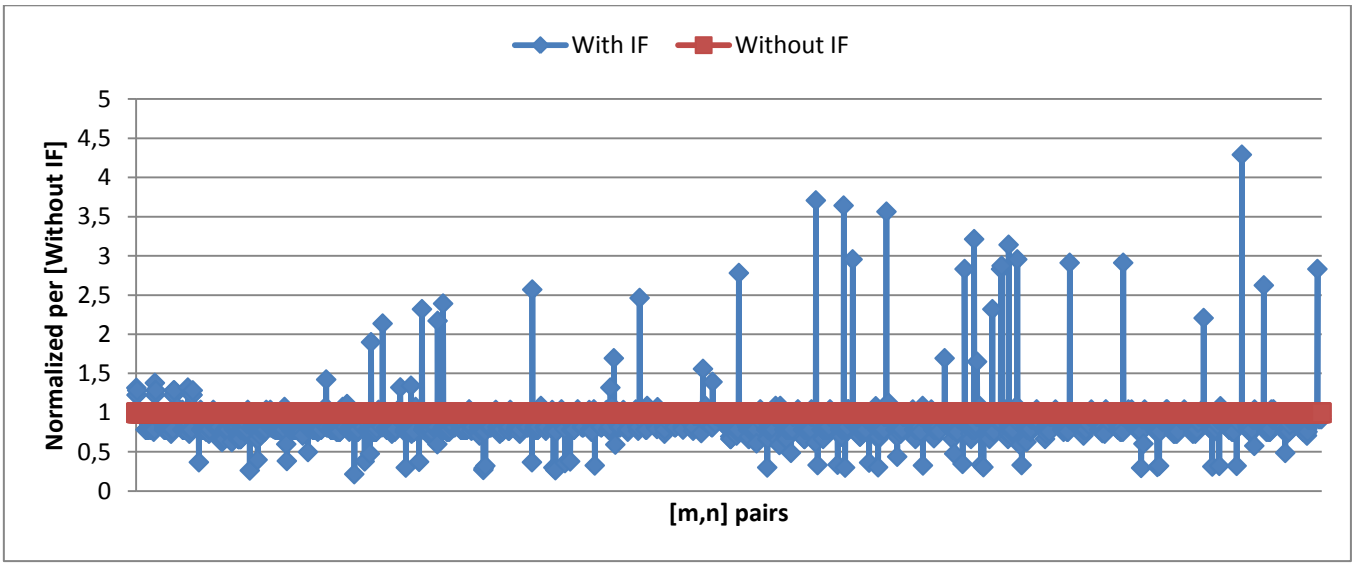

Figure 4 Normalized speed for both approaches for the case *wA = hB = 64*

V. RESULTS

This section presents the results obtained from both of the matrix multiplication approaches for two cases. For the first case of *wA = hB = 32*, we analyze both of the approaches. Fig. 3 presents the speed of the second approach with conditional branching, which is normalized over the speed of the first approach without conditional branching. Additionally, in Fig. 3 we present the speed of first approach which is also normalized, thus is always equal to 1. Higher is better, therefore, every value of the second approach below 1 confirms that introducing few more memory transfers provides increased performance than having conditional branching in the kernel code.

The second experiment that we performed for the second case of *wA = hB = 64* is presented in Fig. 4. Similar as for the first experiment, the results provide an additional confirmation that more memory transfers provides increased performance than having conditional branching in the kernel code.

The average improvement of performance is 10.24% for the case of *wA = hB = 64* and 15.51% improvement in the case of *wA = hB = 32*.

It is interesting to state that for both cases the results present performance increases for some data requirements.

VI. CONCLUSION

In this paper we present an analysis the for parallel matrix multiplication algorithm on GPU devices. Two approaches were implemented and a performance comparison between them was performed. The results show that for both cases the approach with few more memory transfers outperforms the approach with the conditional branching. Additionally, the performance increased noticed for some data requirements is a subject for future work, as well as analysing other matrix multiplication algorithms for GPU devices.

REFERENCES

[1] Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. J. Symbolic Computation, 9(3):251–280, 1990, doi:10.1016/S0747-7171(08)80013-2.

[2] Virginia Vassilevska Williams, Breaking the Coppersmith-Winograd barrier, UC Barkeley and Stanford University unicyb.kiev.ua/~vingar/progr/201112/1semestr/matrixmult.pdf (unpublished manuscript).

[3] Nadav Eiron, Michael Ro deh, Iris SteinwartsMatrix Multiplication: A Case Study of Algorithm Engineering Proceedings WAE98 Saarbruken Germany August 20 22 1997 Ed: Kurt Mehlhorn pp.98-109.

[4] E. Anderson et al. LAPACK: A portable linear algebra library for high-performance computers. Technical Report 20, LAPACK Working Note, May 1990.

[5] K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw, 34:12:1–12:25, May 2008.

[6] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 35:1–35:11, New York, NY, USA, 2011. ACM.

[7] D. Kirk, W. Wen-mei, and W. Hwu, "Programming massively parallel processors: a hands-on approach," USA, 2010.

[8] NVIDIA, "Next generation cuda compute architecture: Kepler gk110," 2012.

[9] Harris, M.J., "General Purpose Computation on GPUs", retrieved February 2013 from http://www.gpgpu.org/.

[10] NVIDIA CUDA, retrieved February 2013 from http://developer.nvidia.com/object/cuda.html/.

[11] The OpenCL Specification, Version 1.1, document Revision 43, 2009, retrieved February 2013 from http://www.khronos.org/opencl/.