Bulletin of the *Transilvania* University of Braşov
Series I: Engineering Sciences • Vol. 5 (54) No. 2 - 2012

GPU ENHANCED STREAM-BASED MATRIX MULTIPLICATION

L.M. ITU[1] **C. SUCIU**[1,2]
F. MOLDOVEANU[1] **A. POSTELNICU**[3]

Abstract: *The paper introduces an algorithm which improves the value of the real giga floating point operations per second (GFLOPS) for matrix multiplication algorithm on Graphical Process Unit-GPU by overlapping the data transfers between (CPU) and the device (GPU) with the kernel execution. The input matrices are divided into n sections and the output matrix into* n^2 *sections. Streams are used to perform simultaneous data transfers and kernel executions in order to hide the memory copy operations. The results show that improved execution times and GFLOP values are obtained. The optimum value of n depends mainly on the matrix dimension and on the GPU type.*

Key words: *matrix multiplication, GPU, stream, CUDA.*

1. Introduction

An alternative to CPU based solutions has been raised from Graphics Processing Unit (GPU) based implementations. The GPUs were introduced initially as graphical accelerators with focus, mainly, to image processing applications. A GPU is a stream processor, specifically designed to perform a very large number of floating point operations (Flops) in parallel by using simultaneously multiple computational units. Currently their performance has gone beyond 1 teraflop which makes the GPU several times faster than any multi-core CPU similar implementation. The relative recent introduction of non-graphics application programming interfaces (API) for GPUs brought a new perspective on these, transforming in general purpose units (GPGPU) [8].

The GPU's highly parallel device structure is the main reason for its good results [2]. When a GPU is programmed through CUDA, the GPU is viewed as a compute device which is able to run a large number of threads in parallel. A kernel is a function, written in C language, which is executed on the GPU by several threads [9]. The GPU contains several streaming multiprocessors, each of them containing eight cores.

Practically, the vast majority of GPU implementations are done in a CPU-GPU tandem manner. The CPU (called host) is responsible for launching the main application and initializing the used data (e.g. acquired from a database or a process). This data has to be transferred to

[1] Dept. of Automatics, *Transilvania* University of Braşov.
[2] Siemens Corporate Technology, Romania.
[3] Dept. of Mechanical Engineering, *Transilvania* University of Braşov.

the GPU for processing. The GPU (usually also called device) contains a certain amount of global memory to/from which the CPU or host thread can write/read being accessible to all multiprocessors.

The most important parameter, indicating whether it is worth to move the computational intensive part of a program to a GPU, is the execution time. In order to have a complete picture of the comparison, when the execution time for the GPU is determined, the execution times required by the data manipulation between host and device have to be taken into consideration. When comparing the GFLOPS obtained for a GPU and a CPU, one should calculate the *real GFLOPS* for the GPU (sometimes also called honest GFLOPS), with the following formula [6]:

$$f_r = \frac{n_{GFLOP}}{t_{H \to D} + t_{KE} + t_{D \to H}}, \quad (1)$$

f_r - real GFLOPS; n_{GFLOP} - number of giga floating point operations; $t_{H \to D}$ - time needed to copy the data from the host to the device; t_{KE} - time needed to execute the kernel; $t_{D \to H}$ - time needed to copy the data from the device to the host.

Several scientific computation algorithms are based on matrix manipulations over large sets of data [1]. Therefore, any performance improvement for a basic matrix operation might imply reduced running time for the overall application based on the fact that the respective operation is called for a significant number of times. The goal of this article is to introduce an algorithm which improves the value of the real GFLOPS for a matrix multiplication problem by overlapping the data transfers between the host and the device with the kernel execution.

Section 2 will provide a short overview of the matrix multiplication problem. Section 3 will introduce the idea of asynchronous transfers and overlapping of transfers and computation. Afterwards we will present the generalized form of the algorithm and indicate the changes which have to be implemented for the matrix multiplication kernel in order to use the algorithm. Section 4 contains detailed results obtained by applying the algorithm on a GTX260 device. Finally, we will draw some conclusions on our work in section 5.

2. Overview of the Matrix Multiplication

Matrix multiplication is a very popular problem in parallel computing and a kernel routine of many numerical algorithms [5], [4].

The hardware vendors provide two main libraries to be used by the programmers: CUBLAS library (for basic linear algebraic computation) and CUFFT library (for Fast Fourier Transform [3]).

Solutions for matrix multiplication problems are provided in the CUBLAS library (SGEMM) [11] and, as a result, this is the library to which we will relate to. The early versions of CUBLAS (like 1.0) have shown poor performance, which were then strongly improved in the newer versions (like 2.3).

Besides the global memory of the GPU, each multiprocessor also contains shared memory and registers which are split between the thread blocks and the threads, which run on that multiprocessor. A thread block is a batch of threads which can efficiently cooperate through shared memory. The kernel is executed for a batch of blocks which form a two dimensional grid [7]. Each block will be allocated to a multiprocessor, which can contain up to eight blocks. The threads of a block will be grouped into groups of 32, called warps, which are executed in a SIMD (single instruction multiple data) fashion.

The main steps, which have to be followed when computational work is performed on a GPU, are [6]: allocate host and device memory for the input data as well as for the results, copy the data from the host to the global memory of the device, execute the kernel according to the execution configuration, copy back the results from the device to the host, and free the memory resources. Memory instructions are very important in the case of matrix multiplication. Local and global memory copies are not cached for devices of compute capability 1.x and as a result there are 400 to 600 clock cycles of memory latency [10], [12].

Shared memory is on-chip memory and, therefore, it is faster than global or local memory. In [5] the authors assume that the multiprocessor accesses shared memory with zero latency if there are no memory bank conflicts. Shared memory is divided into memory banks, which are equally sized memory modules that can be accessed simultaneously. As a result, the load or store operations which use *n* addresses situated in *n* different memory banks can be performed simultaneously [10]. If the two addresses of the same memory request fall in the same memory bank, a bank conflict appears and the access has to be serialized.

Figure 1 illustrates the main idea behind the matrix multiplication kernels. The input matrices as well as the result matrix are divided into blocks (not necessarily square blocks; a very interesting pattern is indicated in [3]). A block of threads computes a block of elements in the results matrix (every thread will usually compute several elements) by calculating the dot product of the corresponding rows and columns of the matrices A and B respectively. In order to limit the wasted global memory bandwidth, data is first brought by the threads into shared memory Because the shared memory is a limited resource, data is read in tiles from the global memory and the dot-product is computed in several steps.

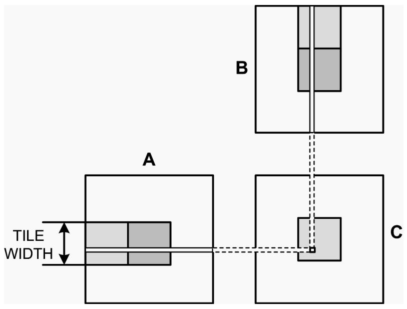


Fig. 1. *Matrix multiplication with shared memory*

Several papers have indicated that the main limiting factor in GPU applications is the bandwidth of the host ↔ device memory transfers (around 8 GB/s over the PCI bus in full duplex mode). This is the case for applications where the number of floating point operations is rather small, but the matrix multiplication is a compute intensive problem. However, time can still be saved by hiding some or most of the host ↔ device memory transfers. The Big-O notation is a popular way to describe how the size of the input elements affects the consumption of a computational resource for a certain algorithm. In case of the matrix multiplication problem, for matrices of NxN elements, there are $3N^2$ transfers and N^3 operations (multiply-add). As a result the ratio is O(N), meaning that the larger the matrix the greater the performance benefit.

3. The Stream-Based Matrix-Matrix Multiplication Algorithm

3.1. Introduction

Data transfers between the host and the device, which are performed through *cudaMemcpy()*, are blocking, i.e. the control is returned back to the host thread only when the transfer is complete. On the other side, *cudaMemcpyAsync()* is non-blocking but requires pinned host memory and an additional argument, namely a stream ID. Pinned memory, also called page locked memory, attains the highest bandwidth between host and device but excessive use can reduce the overall performance because it is a scarce resource [10].

A stream is a sequence of operations, which are performed in order on the device. Operations, which are executed in different streams can be interleaved or overlapped.

The asynchronous transfers lead to overlap of memory copies and computation in two situations:

• host computation can be overlapped with both memory copies and kernel execution; for example, when a kernel is launched and if the CUDA device is idle, the kernel immediately starts running based on the execution configuration and according to the function arguments; meanwhile, the host continues to the next line of code after the kernel launch;

• kernel execution can be overlapped with memory copies.; this requires some additional conditions to be fulfilled: both operations have to be executed in different non-default streams (with non-zero stream IDs) and the device must be capable of “concurrent copy and execute”; the most important fact is to ensure that simultaneous operations in different streams do not operate on the same data segments when using streams.

3.2. The algorithm

The input matrices A and B are split into n slices or sections, then these slices are copied from the CPU memory into the global memory of the GPU and the elements of the result matrix C are processed as these slices become available for the GPU. The elements of matrix C are computed by n^2 kernels. Further, matrix C is also split into slices and when all elements of a slice have been processed, the whole slice is copied back to the CPU memory, while the processing of the next slice takes place (Figure 2).

To be able to use this technique, the only additional requirement is that matrix A has to be stored in the CPU memory in row-major order and matrix B in column-major order. This requirement is necessary in order to be able to efficiently copy the slices of the two matrices into the GPU global memory.

The maximum value of n depends on the size of the matrices. If n is too high, the execution configuration of the kernels may contain too little blocks and as a result the kernel may not take full advantage of the capabilities of the GPU and the total execution time may increase. As a result, the optimum value of n depends on the matrix size and one should experiment in order to find the value corresponding to the shortest execution time. Thus only $1/n$ of the memory operations can not be hidden through the described algorithm.

Although the matrices are divided into n slices, only two streams are needed to handle the overlapping scenarios. Figure 3 displays the operations for the generalized form of the algorithm. The first stream is used to perform the memory copies for the first section of each input matrix and for all kernel executions.

When the kernels corresponding to a section of the result matrix have been processed, an event is recorded and as a result, a total of n events are recorded. The second stream first performs all memory copies left for matrix B, because the kernels corresponding to the first section of matrix C need these data for the computation. Then all memory copies, which correspond to matrix A, are accomplished. Finally the sections of matrix C are copied back to the host. These copies are synchronized by the events recorded in the first stream, as they can only take place after all kernels of a section have finished their processing.

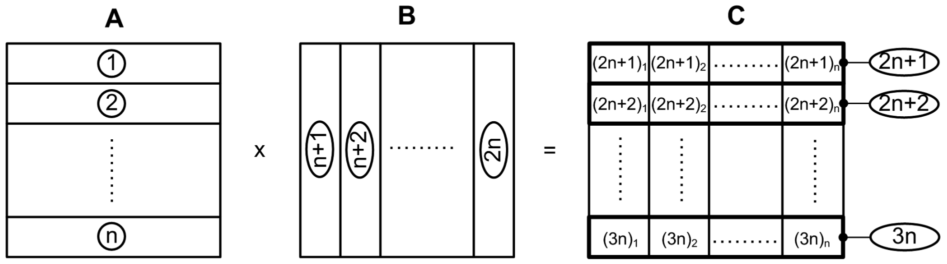
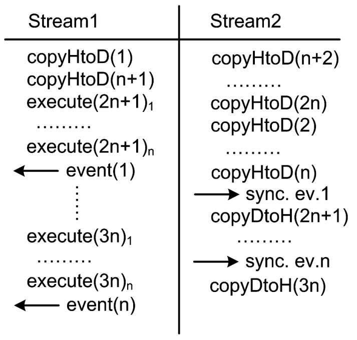


Fig. 2. *Generalized slicing principle for the matrix multiplication algorithm*



Stream1	Stream2
copyHtoD(1)	copyHtoD(n+2)
copyHtoD(n+1)	………
execute(2n+1)$_1$	copyHtoD(2n)
………	copyHtoD(2)
execute(2n+1)$_n$	………
⟵ event(1)	copyHtoD(n)
⋮	⟶ sync. ev.1
⋮	copyDtoH(2n+1)
execute(3n)$_1$	………
………	⟶ sync. ev.n
execute(3n)$_n$	copyDtoH(3n)
⟵ event(n)	

Fig. 3. *Mapping of the operations to the two streams of the generalized algorithm*

As discussed earlier in the paper, all the CUDA functions used for the algorithm are asynchronous, which means control is immediately returned back to the host thread.

3.3. Changes to the multiplication kernel

In order to be able to apply the algorithm described above, obviously the kernel will have to undergo some changes. Since two additional parameters are needed for the kernel (row and column index for each kernel), the total amount of shared memory used by each block increases by 8 bytes.

The main changes inside the kernel are determined by the fact that two additional parameters have to be considered when determining the row and the column of the elements in the result matrix, which are computed by the threads. Further, because of the fact that matrix B is stored in column-major order, we have chosen to transpose the rows of the tiles read from B when they are copied into the shared memory. This way the reads performed by a half warp are coalesced and there is no wasted bandwidth when accessing the global memory. A problem though appears when storing the values read from global memory into shared memory. For simplicity, we will address this problem for tiles of 16x16, but it is similar for greater tiles, as those used in the optimized matrix multiplication algorithms described in chapter two. Devices of compute capability 1.x have 16 memory banks and if the matrix is stored in row-major order all locations of the same column are situated in the same memory bank. Consequently, when the threads of a half warp store the values read by them through a single global memory transaction into the shared memory, there will be 16 bank conflicts, all write operations will be serialized and plenty of time will be lost. A very simple solution to this problem is to allocate a further column of memory in the shared memory, which will not be used. This way only 16 x 4 = 64 extra bytes of shared memory per block will be used, the problem is solved in a very elegant way and there should be no issues regarding the occupancy of the multiprocessors.

If we choose to not transpose the tile read from global memory, then the problem described above still appears. The only difference is that it is postponed until the values are read from shared memory (when the dot-product is calculated). As a result, a column still has to be added to the matrix allocated in the shared memory. If we choose to transpose the matrix, the code which calculates the dot-product remains the same and needs no changes.

4. Results

The algorithm has been tested on a NVIDIA GPU device (GTX260) of compute capability 1.3, containing 27 streaming multiprocessor and a total of 216 cores. Every SM of the GTX260 can contain up to 1024 threads and like every other NVIDIA GPU a total of 8 blocks. As the kernel discussed in section 3.3 uses blocks of 256 threads, a maximum of four blocks can reside inside a SM at any moment in time. This means, that in order to take full advantage of the capabilities of this GPU, the kernel should be launched with a grid of at least 4x27=108 blocks so that all SMs are fully occupied at start time. Nevertheless, this doesn't mean that when a grid of fewer blocks than 108 is used (and thus a higher value for n), the execution time can not become shorter compared to a grid with more blocks. Table 1 displays the sizes of the matrices computed by one kernel and the corresponding number of blocks in the execution configuration, for different values of n (for tiles of 16x16).

Table 1
Execution configurations for different values of n

n	**Size of matrix computed by one kernel**	**Number of blocks in the grid**
1	2048x2048	16384
2	1024x1024	4096
4	512x512	1024
8	256x256	256
16	128x128	64

As one can see, when the value of *n* is doubled, the number of blocks in the grid is divided by four. For *n* = 8 the number of blocks is still high enough to keep all SMs busy at the initial moment, for *n* = 16 though the number of blocks is less than 108 and may lead to an increase of the total execution time.

Next we will present the results we have obtained for matrices of 2048 x 2048 elements and for different values of *n*. First we have run the regular matrix multiplication algorithm, which does not use any streams (which actually is the case of *n* = 1), or more precisely which uses the default zero stream (Table 2 displays the execution duration for this case).

Table 2
Case n = 1 (regular matrix multiplication algorithm)

Stream0	**End time [ms]**	**Duration [ms]**
copyHtoD(1)	3.55	3.55
copyHtoD(2)	7.04	3.49
Execute(3)	88.01	80.97
copyDtoH(3)	**94.32**	6.31

Afterwards we have tested the generalized form of the algorithm with *n* = 2 (Table 3) and *n* = 4 (Table 4). In each table the total execution duration is displayed in bold. The time measurements were performed by using CUDA events, as described in the introduction, and the numbers represent average values obtained after several iterations of the same code.

The execution time has been reduced by 1.7 ms for *n* = 2 and by 3.04 ms for *n* = 4 through the generalized algorithm. By reducing the execution time, the value of the *real GFLOPS* (see Eq. (1)), or *"honest" GFLOPS* have increased correspondingly.

We have stated in the previous chapter that with the basic form of the algorithm half of the memory copies can be hidden.

The total time needed for the memory copies, for n = 1, is 13.35 ms. This means that for n = 2, the execution time should be shorter by 6.675 ms. The difference though is of only 1.7 ms because the kernel execution for the second case takes longer compared to the first one (88.03 ms as opposed to 80.97 and a total increase of 7.06 ms). On the other side in the second case the memory copies are performed much faster, presumably because of the smaller portions of data that are copied (8.97 ms as opposed to 13.35 ms). From the 8.97 ms needed for memory copies, 4.38 ms are hidden and as a result the total time for memory copies is reduced by 8.76 ms. As a result 7.06 ms have been lost on the kernel execution side but 8.76 ms have been gained on the memory copy side.

Now let us compare the cases n = 2 and n = 4. For n = 4, the total execution time should be shorter by a quarter of the total time spent on memory copies for n = 2, i.e. 2.24 ms. The difference though is of only 1.34 ms. Now, for n = 4 the total time spent for memory copies is of 8.96, i.e. approximately equal to the case of n = 2. Of these 8.96 ms, 6.63 ms are hidden. Regarding the kernel execution, it has increased from 88.03 ms to 88.95 ms. As opposed to the previous comparison, this time the kernel code is exactly the same, consequently the time difference can only be explained through the overhead needed to launch a kernel (this time 16 kernels are executed compared to only 4 in the previous case). As a result, compared to the previous case 2.26 ms have been gained on the memory copy side and 0.92 ms have been lost on the kernel execution side.

5. Conclusions

In this paper we have introduced an algorithm used to hide the memory copies between the host and the device for a matrix multiplication problem (AxB=C) in order to improve the total execution time.

Case n = 2 (basic form of the stream-based matrix multiplication algorithm) Table 3

Stream 1	End time [ms]	Duration [ms]	Stream 2	End time [ms]	Duration [ms]
copyHtoD(1)	1.53	1.53	copyHtoD(4)	4.51	1.41
copyHtoD(3)	3.10	1.57	copyHtoD(2)	5.97	1.46
Execute $(5)_1$-$(5)_2$	47.13	44.03	copyDtoH(5)	48.64	1.51
Execute $(6)_1$-$(6)_2$	91.13	44	copyDtoH(6)	**92.62**	1.49

Table 4
Case n = 4 (generalized form of the stream-based matrix multiplication algorithm)

Stream 1	End time [ms]	Duration [ms]	Stream 2	End time [ms]	Duration [ms]
copyHtoD(1)	0.763	0.763	copyHtoD(6)	2.29	0.72
copyHtoD(5)	1.57	0.807	copyHtoD(7)	3.01	0.72
execute $(9)_1$-$(9)_4$	23.84	22.27	copyHtoD(8)	3.74	0.73
Execute $(10)_1$-$(10)_4$	46.08	22.24	copyHtoD(2)	4.49	0.75
Execute $(11)_1$-$(11)_4$	68.33	22.25	copyHtoD(3)	5.20	0.71
Execute $(12)_1$-$(12)_4$	90.52	22.19	copyHtoD(4)	5.96	0.76
			copyDtoH(9)	24.59	0.75
			copyDtoH(10)	46.82	0.74
			copyDtoH(11)	69.08	0.75
			copyDtoH(12)	**91.28**	0.76

The novelty of the approach is to divide the input matrices A and B into *n* horizontal, respectively vertical sections and to compute the elements of the result matrix C through n^2 kernels. This way only the memory copies corresponding to the first section of matrix A and B and to the last section of matrix C can not be hidden by the kernel execution. As a result we have managed to hide $(n-1)/n$ of the memory operations.

The kernel had to be adapted (two additional input parameters and extra shared memory in order to avoid memory bank conflicts) but the performance penalties determined by the changes have been outweighed by the hidden memory copy time. When the value of n increases, the time of the unhidden memory transfers is reduced, but there is a slight increase of the total kernel execution time caused by the overhead needed to launch the kernels. Another aspect that has to be considered is that for values which are too high, the grid may contain too few blocks to keep the GPU busy. This is why it is important to test the code with various values of n in order to find value corresponding to the shortest execution time. It can be concluded that the optimum value of n depends on many factors, like: matrix dimensions, GPU type, and CUDA toolkit version.

Acknowledgment

This paper is supported by the Sectoral Operational Programme Human Resources Development (SOP HRD), financed from the European Social Fund and by the Romanian Government under the contract number POSDRU/88/1.5/S/76945.

References

1. Blazewicz, M., et al.: *Problems Related to Parallelization of CFD Algorithms on GPU, Multi-GPU and Hybrid Architectures*. In: AIP Conference Proceedings, Rhodes, 2009, p. 1301-1304.
2. Chen, G., et al.: *High Performance Computing Via a GPU*. In: International Conference on Information Science and Engineering, Shanghai, 2009, p. 238-241.
3. Cui, X., Chen, Y., Mei, H.: *Improving Performance of Matrix Multiplication and FFT on GPU*. In: 15th Intern. Conf. on Parallel and Distributed Systems, Shenzen, 2009, p. 42-48.
4. Dziekonski, A., Mrozowski, M.: *Tuning Matrix-Vector Multiplication on GPU*. In: 2nd Inter. Conf. on Information Technology, Gdansk, 2010, p. 167-170.
5. Fujimoto, N.: *Faster Matrix-Vector Multiplication on GeForce 8800GTX*. In: IEEE Inter. Symp. on Parallel and Distributed Processing, Osaka, 2008, p. 1-8.
6. Kirk, D., Hwu, W.M.: *Programming Massively Parallel Processors*. London. Elsevier, 2010.
7. Kothapalli, K., et al.: *A Performance Prediction Model for the CUDA GPGPU Platform*. In: Intern. Conf. on High Performance Computing, Kochi, 2010, p. 463-472.
8. Owens, J.D., et al.: *GPU Computing*. In: Proc. of the IEEE **96** (2008), p. 879-884.
9. Zou, C., Xia, C., Zhao, G.: *Numerical Parallel Processing Based on GPU with CUDA Architecture*. In: Inter. Conf. on Wireless Networks and Information Systems, Wuhan, 2009, p. 93-96.
10. NVIDIA Corporation. CUDA, Compute Unified Device Architecture Best practices guide v4.0, 2012.
11. NVIDIA Corporation. CUDA, Compute Unified Device Architecture CUBLAS Library v4.0, 2012.
12. NVIDIA Corporation. CUDA, Compute Unified Device Architecture Programming guide v4.0, 2012.