Comparing CPU and GPU Implementations of a Simple Matrix Multiplication Algorithm

Tomaž Dobravec*, Patricio Bulić
University of Ljubljana, Faculty of computer and information science, Slovenia.

* Corresponding author. Tel.: +386 1 47 98 256; email: tomaz.dobravec@fri.uni-lj.si
Manuscript submitted May 25, 2017; accepted July 25, 2017.
doi:10.17706/ijcee.2017.9.2.430-438

Abstract: In recent years, graphics processing units (GPU) have become a standard part of high-performance computing systems used for solving large scale computation problems. To relieve the main processor more and more time consumptive tasks are moved from CPU to GPU where algorithms run in parallel on a high number of GPU's processors. In this paper we present both sequential and parallel implementations of a simple matrix multiplication algorithm and we compare the overall execution time. To further speed up the execution we introduce the GPU's fast shared memory and the implementation of the matrix multiplication algorithm that exploits this memory. The results presented in this paper show that the GPU implementation with the use of shared memory is two times faster than the implementation that uses only device's global memory and up to 7.5 times faster than the CPU implementation.

Key words: GPU v.s. CPU, CUDA, shared memory, algorithm optimization.

1. Introduction

A many-core GPU processor is an integrated circuit in which a higher number of processors has been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks or threads [1]. A Graphical Processing Unit has up to M streaming processors (SP), organized as up to N streaming multiprocessors (SM). Each SM has 8 or 16 SPs. SM is an independent processing unit. SM consists of a Fetch/Issue unit and an execution unit. Each SM processes batches of blocks of threads. A block of threads is processed by only one SM. Each block is split into groups of 32 threads called warps. A warp is executed physically in parallel.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. It provides a unified hardware and software solution for data-intensive computing. CUDA is also an extension of the C programming language and was created by NVIDIA. Using CUDA the programmer can take advantage of the massive parallel computing power of an nVidia graphics card in order to do general-purpose computation [2]-[4]. From the programmer's point of view, CUDA is a device, which can be used through a parallel programming model. The code is split into the so-called kernels, which are executed in up to several thousand parallel and/or asynchronous threads. Parallel threads can cooperate using synchronization mechanisms and shared memory, while asynchronous threads can only use a common global memory.

In CUDA device and programming environment a GPU is viewed as a compute device operating as a coprocessor to the main CPU (host). Data-parallel, compute intensive functions are to the device and

executed many times, but independently on different data. A function compiled for the device is called a kernel. The kernel is executed on the device as many different threads.

The execution model is as follows. All threads make a computational grid. The computational grid consists of thread blocks where each thread executes the kernel. The size of the block depends on GPU memory and kernel complexity but it is usually limited to 512 threads per block. The threads within the same block can be synchronized and can communicate via shared memory, which is implemented on the die within one SM. The threads from different blocks can communicate only via global memory, which is implemented in DDR DRAM modules on the GPU card [1], [5]-[7].

CUDA exposes different types of memory on the GPU: registers per thread, local memory per thread, shared memory per block (used for sharing data within a block), global memory per grid, constant memory per grid and texture memory per grid.

The rest of this paper is organized as follows. In Section 2 we introduce three implementations of matrix multiplication algorithms: matrixMulH (a CPU implementation), matrixMulDG (GPU implementation with the use of global memory) and matrixMulDS (GPU implementation with the use of shared memory). In Section 3 we present the results of performance tests - we examined the speed of the tree presented algorithms and the impact of the number of used GPU threads to the overall execution time. We conclude this paper in Section 4.

2. Matrix Multiplication

Matrix multiplication is a very frequent and at the same time very slow mathematical operation, since its time complexity (even if we use some improvements, such as for example Strassen's) is close to cubic [8]-[11]. This mathematical operation was taken as a test example for several reasons: a) it involves a simple operation for which we can predict the behavior at various sizes of input data, b) the basic implementation of the operation is well known and very simple, c) because of the nice properties of all sub-operations (commutativity and distributivity of addition and multiplication), matrix multiplication may be carried out in several ways, also by using block sub-matrix multiplication; this variant of the algorithm is suitable for the testing of the use of the device's shared memory that is much faster than the global one.

All matrices are presented in the tests as one-dimensional tables with additional information about the matrix dimension. If a matrix of dimension $h \times w$ is represented by table M, the index of the element (i, j) of this matrix is calculated according to formula $i * w + j$. The task of calculating indices was thus transferred from the compiler to the programmer and at the same time, by using this presentation, we simplified the transfer to the device and working with data on it.

The implementation speed of matrix multiplication was measured in three different ways: a) the implementation of ordinary multiplication on a single-processor host, b) the implementation of ordinary multiplication on CUDA device with the use of global memory, and c) the implementation of modified block multiplication on CUDA device with the use of shared memory. The result of implementing the three mentioned multiplications (product matrix) was the same every time; the difference between individual implementations was only in the time of implementation. In the continuation, we will present individual implementations and the results of the testing. Matrices will be marked A, B, and C, whereas their dimensions will be marked hX, wX, whereby X indicates one of the mentioned matrices, h indicates its height, and w indicates its width.

The C program code that is described in this paper was introduced in the *NVIDIA CUDA C SDK Code Samples* library [6]. We used this code without modifications.

2.1. Ordinary Multiplication on the Host

For ordinary matrix multiplication on the host, matrixMulH function was used (see Listing 1) in which we

applied a known formula for the calculation of (i, j)-element of product matrix.

$$c_{ij} = \sum_{k=0}^{wA} a_{ik}\, b_{kj}$$

```
void matrixMulH(float* C, const float* A,
    const float* B,  int hA, int wA, int wB)
{
    for (int i = 0; i < hA; ++i)
        for (int j = 0; j < wB; ++j) {
            double sum = 0;
            for (int k = 0; k < wA; ++k) {
                double a = A[i * wA + k];
                double b = B[k * wB + j];
                sum += a * b;
            }
            C[i * wB + j] = (float)sum;
        }
}
```

Listing 1. Matrix multiplication on host.

Three nested for loops directly imply the cubic time complexity of this implementation.

2.2. Ordinary Multiplication on the Device

Matrices may be multiplied on the device, so that one kernel is charged for the calculation of each product matrix element. This will perform its task by multiplying the corresponding row of the first and column of the second matrix. Ordinary (hence non-blocked) matrix multiplication on the device was carried out with matrixMulDG kernel (see Listing 2). The data on both matrices (matrix A and B) were read directly from the global memory; whereto the result (matrix C) was also recorded.

```
__global__ void matrixMulDG(float* C, float* A,
    float* B, int wA, int wB)
{
  float Cvalue = 0;
  int r = blockIdx.y * blockDim.y + threadIdx.y;
  int c = blockIdx.x * blockDim.x + threadIdx.x;
  for (int e = 0; e < wA; ++e)
    Cvalue += A[r * wA + e] * B[e * wB + c];
  C[r * wB + c] = Cvalue;
}
```

Listing 2. Matrix multiplication on device.

The calculation of one product matrix element on the device (similar to that on the host) is performed with the help of one for loop. Since more kernels are running simultaneously on the device, we can expect that the time needed for the calculation of the product here will be shorter for a constant factor than the time needed at the host.

Kernels were organized in blocks of size BLOCK_SIZE × BLOCK_SIZE, and the number of blocks was modified according to the size of matrices. To avoid troubles with calculating border elements at arbitrary matrix dimensions we limited our tests only to the matrices of ``correct'' dimensions (a multiple of BLOCK_SIZE parameter). The code in Listings 3 shows the call of matrixMulDG kernel with the mentioned number of threads and blocks.

```
    dim3 threads(BSIZE, BSIZE);
    dim3 grid(wC / threads.x, hC / threads.y);

    matrixMulDG<<< grid, threads >>>
        (C, A, B, wA, wB);
```

Listing 3. Invoking the matrixMulDG kernel.

2.3. Modified block Multiplication on the Device

The largest deficiency of ordinary matrix multiplication on the device, which was presented in the previous section, is the use of slow global memory. Shared memory on the device (memory that is common to all threads of the block) is much faster, but unfortunately there is not enough of it to load all the data (both matrices) in it. The condition needed for successful use of shared memory, namely that the threads of the same block use the same data, is not met in classic matrix multiplication. Namely, here the (i, j)-thread requires the i-row of matrix A and the j-row of matrix B; no other thread requires these two pieces of information. The multiplication procedure is to be modified, so that the threads within the same block will require a smaller quantity of the same data. This is done by using block multiplication of sub matrices of dimension $b \times b$ (note: parameter b, which in theory may be of any size, in practice determines, in addition to the dimension of submatrices, also the number of threads that will carry out the calculation; because of the limitation of devices that up to 512 threads may run in one block, its value must be somewhere between 2 and 16).

```
__global__ void matrixMulDS( float* C,
    float* A, float* B, int wA, int wB) {

  // indices of thread and block
  int bx = blockIdx.x,  by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;

  int aBegin = wA * BSIZE * by;
  int aEnd   = aBegin + wA - 1;
  int aStep  = BSIZE;
  int bBegin = BSIZE * bx;
  int bStep  = BSIZE * wB;

  float Csub = 0;

  for (int a = aBegin, b = bBegin;
            a <= aEnd;
            a += aStep, b += bStep) {

      __shared__ float As[BSIZE][BSIZE];
      __shared__ float Bs[BSIZE][BSIZE];

      As[ty][tx] = A[a + wA * ty + tx];
      Bs[ty][tx] = B[b + wB * ty + tx];
      __syncthreads();
      for (int k = 0; k < BSIZE; ++k)
          Csub += As[ty][k] * Bs[k][tx];
      __syncthreads();
  }
  int c = wB * BSIZE * by + BSIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
```

Listing 4. Modified multiplication kernel with the use of shared memory.

To calculate product $A_{ij}B_{jk}$ of dimension $b \times b$ we need b^2 elements of matrix A and just as many elements of matrix B. This multiplication sub-operation is carried out in b^2 threads, namely so that each thread first loads "its own" elements of matrices A and B in the shared memory and then by using local data calculates its own piece of the result.

Synchronization is mandatory between the first and the second part of the procedure – after loading the

piece of information in the shared memory, the thread has to wait for the other threads to finish their work – this ensures that all data is in its place. In the code below, the mentioned block matrix multiplication with matrixMulDS kernel is implemented (see Listing 4). The outer for loop takes care of the "walk" along the block row and column, whereas the inner one for the calculation of one element of product $A_{ij}B_{jk}$.

Kernel matrixMulDS is executed the same way as kernel matrixMulDG (see above). Parameter BLOCK_SIZE simultaneously determines the size of submatrices b =BLOCK_SIZE) and the number of threads.

3. Evaluating the Algorithms

The algorithms that were presented in the previous section were checked in several ways. We were primarily interested in the correctness of operation and the speed of implementation. Regarding the correctness of operation, we found out that all algorithms give back almost the same result. Namely, in the multiplication of arbitrary matrices of different sizes, it sometimes (rarely, but nevertheless) occurs that individual elements of product matrices, obtained with different algorithms, differ on the fourth and fifth decimal. This problem was ascribed to the rounding error in working with real numbers (for the presentation of real numbers, the float type was used).

The implementation speed was measured in two ways: first we compared different implementations, and then the best implementation was used to assess the impact of the selection of the number of threads on the speed of implementation.

3.1. Implementation Environment

To test the program designed for CUDA architecture we used GeForce 9400M graphic card, which contains 2 multiprocessors with eight kernels each, 256MB of global memory and a clock with the speed of 1.1GHz. For reference tests we used a computer with a dual-core 2.66GHz Intel processor and with 4 GB of memory and MacOS 10.9 operating system.

3.2. The Speed Tests

To measure the speed of different implementations two tests were carried out, the first by using 64 threads (block of 8×8 size) and the second by using 256 threads (block of 16×16 size). Both tests gave similar results and in continuation, the data about the first (64 threads) are presented. To avoid problems that may arise in calculating edge elements of the result in matrices of "irregular" dimensions we carried out the measurements on matrices of dimension $8n \times 8n$ at $n = 1, 2, \dots, 100$. The implementation speed of multiplications (time in milliseconds) for some matrix dimensions is shown in Table 1, and graphically in picture Fig. 1. MulH was used to mark the implementation speed of an ordinary algorithm on the host, and MulDG was used to mark the implementation speed of ordinary (non-block) multiplication on the device. Despite the use of slow memory, we see that multiplication on the device is much faster, and the difference between multiplying on the host and multiplying on the device increases with the increase in the size of input data. Even better results were obtained with an algorithm that exploits the shared memory (MulDS) with the help of block multiplication – Table 1 shows that it involves almost a twofold speedup compared to ordinary multiplication on the device and a very large speedup (speedup factor between 5 and 8; with larger dimensions of matrices, the speedup factor increases) compared to the multiplication on the host. The implementation time for each algorithm is shown in Fig. 1, namely MulH in red, MulDG in black and MulDS in blue. We can see that the use of CUDA architecture for calculating the matrix product is justified, especially with large dimensions of matrices and with the selection of the algorithm that uses the shared memory correctly. This finding is not dampened even by the fact that data (elements of matrices) have to be transferred to the device before the implementation of multiplication and

that at the end the result has to be transferred back to the host. Namely, our measurements showed that less than 10ms are needed for these operations at the largest tested matrix, which is compared to the speedup (275ms instead of 2169ms) negligible.

At this point we should also mention the side result of the measurements, which we cannot explain satisfactorily. When measuring the implementation speed of ordinary multiplication on the host, we obtained the following results: 5522, 5792, 27893, 6277, 6882 with matrices of sizes $16n \times 16n$, whereby $n = 62, 63, 64, 65, 66$, and whereby $n = 94, 95, 96, 97, 98$, the results: $20401, 20707, 37020, 22175, 23383$ were obtained. We can see that it is an ordinary cubic increase of the implementation time as a function of the size of the input, but two exceptions do occur (with $n = 64$ and $n = 96$), which show a significantly larger time value than expected. It is interesting that these anomalies occur with the "nice" values of parameter n ($4 * 16$ and $6 * 16$). We try to explain the noticed fact with the paging of the main memory (probably a part of the matrix lies on one, and the other part on the other page; the switch between the pages is slow); however, it is difficult to understand why the situation is back to normal already in the values of parameter n that follow.

Table 1. The Implementation Speed (in ms) of the Tree Implementations of Matrix Multiplication. The Input Data Size was $8n \; x \; 8n$. MulH = Multiplication on the Host, MulDS = Multiplication on the Device with the Use of Shared Memory, MulDG = Multiplication on the Device with the Use of Global Memory

n	MulH	MulDG	MulDS
10	1	0	0
20	10	5	2
30	34	13	8
40	85	37	18
50	156	65	35
60	290	127	60
70	576	182	95
80	895	302	142
90	1450	398	201
100	2169	598	275

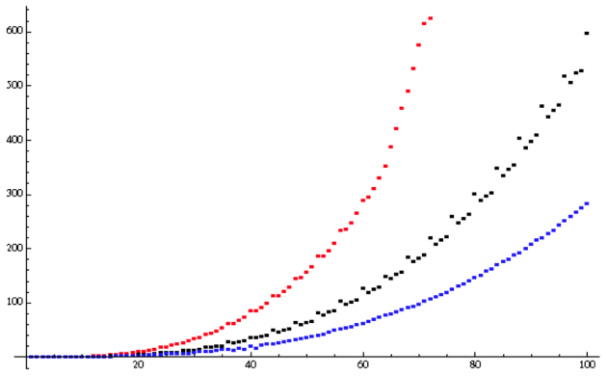

Fig. 1. The implementation speed of different algorithms for matrix multiplication. The graph shows the time spent in relation to the size of input matrices, namely for (a) ordinary multiplication on the host (red color), (b) multiplication on the device with the use of global memory (black color), (c) multiplication on the device with the use of shared memory (blue color).

3.3. The Optimal Number of Threads Test

We have also measured the implementation speed of matrix multiplication by using the best of the above mentioned algorithms (application of matrixMulDS kernel) in relation to the number of used threads per block. For this purpose, the kernel code needed to be partly changed. Namely, CUDA foresees two ways of

assigning shared memory: (a) if the size of the memory is known at the time of compiling, and (b) if the size of the memory is known during implementation. The code presented above foresees that the number of threads (and thus the size of the required memory) is known at the time of compiling (BLOCK_SIZE is a constant). We have modified the code, so that tables As and Bs were declared together with the reserved word extern as one-dimensional dynamic tables. By doing this, we undertook the task of calculating two-dimensional indices into one-dimensional ones, which slightly slowed down the entire implementation. With the kernel processed in this manner, we multiplied matrices of size 512×512 in blocks with 1, 4, 16, 64, and 256 threads (see Table 2).

Table 2. The Implementation Speed of the Algorithm for the Multiplication of Matrices of Dimension 512×512 on the Device with the Use of Shared Memory as a Function of the Number of Threads per Block. Reference MulH=357

Number of threads	MulDS
1	11149
4	1531
16	278
64	110
256	83

As expected, it turned out that the implementation time decreases with the increase of the number of threads. The implementation time was compared to the time that was required for the product of the same matrices on the host (MulH=357). The implementation of the algorithm on the device with the use of a single thread and with the use of four threads is much slower than the implementation on the host. However, the use of 16 threads tips the scales in favor of the algorithm that is being implemented on the device. With the use of 256 threads, the speedup is more than fourfold.

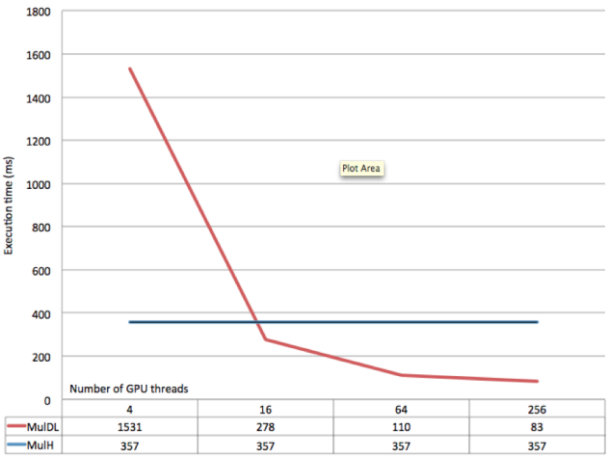

Fig. 2. Execution time of GPU MulDS algorithm with different number of threads comparing to CPU MulH algorithm.

4. Discussion

We compared three implementations of the matrix multiplication algorithm, a classical cubic one for execution on one-threaded CPU and two designed for execution on multi-threaded GPU. The main difference between the former implementations is in the usage of device memory: the first one uses slow global while the second one uses fast shared memory. The goal of this paper was to show how the architecture selection (i.e. CPU or GPU) and the memory usage impacts on the execution time of an

algorithm.

To show the detailed differences in coding for different platforms we presented the three source codes of implemented algorithms. While the first two are very simple and straight-foreword the third is quite complicated. To exploit the shared memory of a GPU device this implementation has to rearrange the calculation order so that the threads that use the same input data run in parallel. A great speedup can be achieved only if a data loaded from global to shared memory is used several times.

By comparing the execution time of simple CPU and GPU implementations on a set of input matrices we observe that the usage of GPU significantly improves the speed of execution. The improvement factor increases with the size of input. For matrices of size 800×800, the GPU implementation is more than 3.5 times faster than the CPU one.

Furthermore, the usage of the shared memory in the GPU implementation further speeds up the execution dramatically. On our input data this fast GPU implementation was up to 7.5 times faster than the CPU implementation.

We also measured how the number of threads impacts on the execution time and we showed that the speed of algorithm increases if GPU uses more threads.

5. Conclusion

In this paper we focused on a matrix multiplication problem and we presented three implementations of algorithms solving this problem – a classical CPU implementation and two GPU implementations, one using global and the other using shared memory. We run these implementations in a real environment and measured the execution time. The results show that GPU computation has a lot of advantages, especially in speeding up the execution time of our programs while on the same time the computer's main processor is released and thus able to perform other tasks. When designing the GPU implementation, a special attention should be paid to the usage of device's shared memory. Careful use of this valuable resource has a significant impact to the overall computation time.

References

[1] Andre, R. B., Trond, R. H., & Martin, L. S. (2013). Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing, 73(1)*, 4–13, Metaheuristics on GPUs.

[2] Ghorpade, J., Parande, J., Kulkarni, M., & Bawaskar, A. (2012). GPGPU processing in CUDA architecture. *Advanced Computing: An International Journal, 3(1)*, 105–120.

[3] Gupta, S., & Babu, M. R. (2011). Performance analysis of GPU compared to single-core and multi-core CPU for natural language application. *International Journal of Advanced Com puter Science and Applications, 2(5)*, 50–53.

[4] Ghorpade, J., *et al.* (January 2012). GPGPU processing in CUDA architecture. *Advanced Computing: An International Journal (ACIJ), 3(1).*

[5] Che, S., *et al.* (October 2008). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing, 68(10).*

[6] Nvidia. (2011). Nvidia developer zone. Retrieved from http://goo.gl/AlmtlP

[7] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE: Vol. 96, No. 5* (pp. 879–899).

[8] Coppersmith, D., & Winograd, S. (1981). On the asymptotic complexity of matrix multiplication. *SIAM J. Comput, 11(3)*, pp. 472–492.

[9] Coppersmith, D. (1982). Rapid multiplication of rectangular matrices. *SIAM Journal on Computing,*

11(3), pp. 467–471.

[10] Le Gall, F. (2012). Faster algorithms for rectangular matrix multiplication. *Proceddings of the 53rd Annual Symposium on Foundations of Computer Science, IEEE.*

[11] Eunice, E. S. (June 2003). Parallel complexity of matrix multiplication. *J. Supercomput., 25(2)*, pp. 155–175.

Tomaž Dobravec received his Dipl.Ing. degree in 1996 in mathematical science and his Ph.D. in 2004 in computer science, both from the University of Ljubljana. He is an assistant professor at the Faculty of Computer and Information Science, University of Ljubljana. His main research interests are in algorithm design, analyses and evaluation, in theory of programming languages and in networks.

Patricio Bulić received the B.S. degree in electrical engineering from the University of Ljubljana in 1998 and the M.S. and Ph.D. degrees in computer science from the University of Ljubljana in 2001 and 2004, respectively. Since 2012 he has been an associate professor at the Faculty of Computer and Information Science, University of Ljubljana. His research interests include computer architecture, computer arithmetic, digital design, embedded systems and parallel processing.