

# GPU Acceleration of Dense Matrix and Block Operations for Lanczos Method for Systems over Large Prime Finite Field

Nikolai Zamarashkin<sup>(✉)</sup> and Dmitry Zheltkov

INM RAS, Gubkina 8, Moscow, Russia

[nikolai.zamarashkin@gmail.com](mailto:nikolai.zamarashkin@gmail.com), [dmitry.zheltkov@gmail.com](mailto:dmitry.zheltkov@gmail.com)

<http://www.inm.ras.ru>

**Abstract.** GPU based acceleration of computations with dense matrices and blocks over large prime finite field are studied. Particular attention is paid to the following algorithms:

- multiplication of rectangular  $N \times K$  blocks with  $N \gg K$ ;
- multiplication of  $N \times K$  blocks by square  $K \times K$  matrices;
- LU-decomposition of matrices.

Several approaches for optimal use of GPU resources are proposed.

Efficiency analysis of implemented algorithms is provided for prime finite field with number of elements about  $2^{512}$ ,  $2^{768}$ ,  $2^{1024}$  and GPUs of different computational performance and architecture generations. Numerical experiments prove efficiency of proposed solutions.

From numerical results it follows that GPU usage allows to accelerate block operations and to expand area of almost linear parallel scalability of Lanczos method implementation by INM RAS. Moreover, a sparse system of size about 2 millions, with 82 average nonzero elements per row, over field with about  $2^{512}$  elements, on 128 nodes of Lomonosov supercomputer will be solved 2 times faster in case of GPUs used.

**Keywords:** GPGPU · RSA · Large prime finite field · Block Lanczos method

## 1 Introduction

This research is the result of analysis made for implementation of the improved block Lanczos method for the linear systems over large prime finite field (see, [6]).

Until recently, large data exchanges were considered as the main reason for poor scalability of block Lanczos method implementations on powerful computing systems [3–5, 7]. Moreover, in case of low speed communication network the acceleration noticeably deviated from linear for the number of nodes about 100.

The basic idea of [6] was the efficient way of data storage. Thanks to this, the data exchange is significantly reduced, and is *perfectly scalable for block size  $K$* . As a result, in the improved implementation the time for data exchange turned out to be less than the time for operations with dense matrices and blocks, which

does not depend on the block size  $K$ . As a matter of fact, the larger the block size  $K$  the greater the difference.

Thus, it is the computations with dense matrices and blocks that determine the limit of linear scalability for the improved implementation of block Lanczos method. Namely, while the time for symmetrized sparse matrix by the block multiplication significantly exceeds the time for operations with dense matrices and blocks, the parallel properties of the method are almost perfect. In practice, this is valid up to a few hundreds or a thousand nodes.

In order to spread the ideal scalability even further (for example, up to  $2^{13}$  nodes), we have to speed up the computation with blocks.

As modern computing nodes are multi-core systems, they are very advantageous for operations with dense matrices and blocks. The use of multicore accelerates these operations proportionally to the number of cores. However, the number of cores per node is usually limited (typical values about 8 – 16). But for really hard problems this is not enough. The systems with much larger number of cores are needed. The most common example of such system are the graphic accelerators (GPUs).

This paper explores the possibility of the using GPUs for computations with dense matrices and blocks with elements from large prime fields. The examples are considered for the fields with the number of elements of order  $2^{512}$ ,  $2^{768}$ , and  $2^{1024}$ .

The choice of algorithms for efficient implementation is constricted due to restrictions on access to GPU resources and dependence of time on presence of branches in the program. For these reasons, we prefer simple algorithms with a regular structure. However, if possible we use the Winograd's idea to reduce the number of multiplications twice. This is important, since in large prime fields the complexity of multiplying greatly exceeds the complexity of addition and subtraction.

Our main purpose is to clarify the possibility of *significant acceleration (more than 10 times)* of calculation dense matrix by block product.

The applicability of this research is not just limited to calculations in the block Lanczos method. The same improvements can be useful for Thome type algorithm implementations [1,2].

## 2 GPU Acceleration of Operations with Dense Matrices and Blocks

### 2.1 Algorithms

The algorithms with simple structure are preferred for GPU accelerators since the limited resource management, and dependence of algorithm running time on the presence of branches in the program.

Let us assume that elements in the large prime field can be specified using 512, 768, or 1024 bits. Further we show that the field size (the number of elements in the field) can significantly affect the implementation efficiency.

We are interested in two particular cases:

1. implementation of block-by-block multiplication  $X^T Y$  for  $N \times K$  blocks  $X, Y$ ;
2. implementation of block-by-matrix multiplication  $XU$  for  $N \times K$  block  $X$ , and  $K \times K$  matrix  $U$ ;

We decided on “naive algorithm”, as well as on an algorithm by Winograd, which reduces the number of multiplications twice.

In addition, we use Winograd idea for efficient  $LU$  decomposition, which is used in block Lanczos method for  $K \times K$  matrix inversion. However, the total time for operations with  $K \times K$  matrices is still significantly smaller than the time for operations with  $N \times K$  blocks [6, 7]. So the improvement of  $LU$  decomposition via Winograd method is considered only as a theoretical result.

The Winograd method is based on the elementary equality for the elements of matrix  $C = AB$ . Assuming the number of columns and rows of  $A$  to be  $2m$ , we write

$$\begin{aligned}
 c_{ij} &= \sum_{k=1}^{2m} a_{ik} b_{kj} \\
 &= \sum_{k=1}^m (a_{i,2k-1} + b_{2k,j}) (a_{i,2k} + b_{2k-1,j}) \\
 &\quad - \sum_{k=1}^m a_{i,2k-1} a_{i,2k} - \sum_{k=1}^m b_{2k-1,j} b_{2k,j}.
 \end{aligned} \tag{1}$$

The last two sums have low complexity and can be pre-calculated in advance. The main calculation corresponds to the sum (1). It is easy to see that the number of multiplications in this sum is 2 times less than the one in “naive algorithm”.

While Winograd method for matrix product multiplication is well known, the similar technique for Gaussian elimination is not. Since a complete description of an algorithm would be unnecessarily cumbersome, we only give its main idea.

Consider a *strictly regular* matrix  $A$  of order  $N$  in the block form

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \tag{2}$$

with  $2 \times 2$  block  $A_{11}$ . The first two steps of elimination can be written as

$$A \rightarrow A - \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} A_{11}^{-1} \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & A_{22}^{(1)} \end{bmatrix}, \tag{3}$$

where the submatrix  $A_{22}^{(1)}$  of  $A$  is used as a starting point for the next steps of Gaussian eliminations. Thus calculation of  $A_{22}^{(1)}$  determines the complexity of the whole algorithm.

Indeed, let us transform (3) by removing  $A_{11}^{-1}$  from it. For this, we represent  $A_{11}$  using the strict regularity of  $A$  and  $\mathcal{O}(1)$  multiplications as

$$A_{11} = L_{11} U_{11}, \tag{4}$$

with lower triangular  $2 \times 2$  matrix  $L_{11}$  and upper-triangular  $2 \times 2$  matrix  $U_{11}$ . Then

$$A \rightarrow A - \begin{bmatrix} L_{11} \\ A_{21}U_{11}^{-1} \end{bmatrix} [U_{11} \ L_{11}^{-1}A_{12}] = A - \begin{bmatrix} L_{11} \\ \hat{A}_{21} \end{bmatrix} [U_{11} \ \hat{A}_{12}], \quad (5)$$

Note that  $\mathcal{O}(N)$  multiplications are enough to find  $\hat{A}_{21}$  and  $\hat{A}_{12}$ .

Now we will show that it is possible to calculate  $A_{22}^{(1)} = A_{22} - \hat{A}_{21}\hat{A}_{12}$  with just  $(N-2)^2 + \mathcal{O}(N-2)$  multiplications. In order to do this consider a matrix product

$$C = [A_1 \ A_2] \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}, \quad (6)$$

where  $A_1, A_2$  are columns, and  $B_1, B_2$  are rows of order  $N-2$  (take into account that the sizes of  $\hat{A}_{21}$  and  $\hat{A}_{12}$  are equal to  $(n-2) \times 2$  and  $2 \times (N-2)$ , respectively). Then using Winograd technique we can write

$$C_j^i = (a_{i1} + b_{j2})(a_{i2} + b_{j1}) - a_{i1}a_{i2} - b_{j1}b_{j2}, \quad (7)$$

where  $a_{i1}, a_{i2}, b_{j1}$ , and  $b_{j2}$  are components of the vectors  $A_1, A_2, B_1$  and  $B_2$ , respectively. The statement about the number of multiplications for two steps of Gaussian eliminations directly follows from (7), and the general result follows from the induction on the matrix size.

## 2.2 Algorithm Mapping on GPU Architecture

**“Naive Algorithm” for Matrix Multiplication.** The organization of calculations is similar to the one proposed in [9]. Consider multiplication of  $N \times M$  matrix  $A$  and  $M \times K$  matrix  $B$  with the elements in a large prime field.

Suppose that the memory size of GPU is sufficient to store the matrices  $A, B$  and the resulting matrix  $C = AB$ . Let the matrix  $A$  be represented in the following row-block form

$$A = \begin{bmatrix} A^1 \\ A^2 \\ \dots \\ A^{\frac{M}{t}} \end{bmatrix}, \quad (8)$$

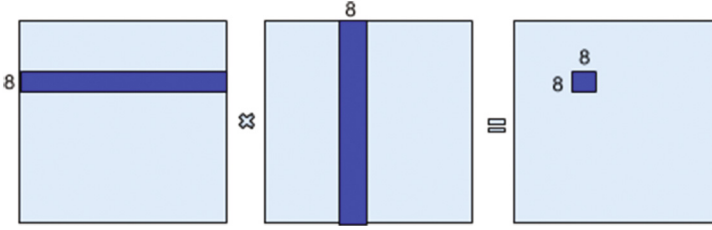
and  $B$  in column-block form

$$B = \begin{bmatrix} B_1 & B_2 & \dots & B_{\frac{K}{t}} \end{bmatrix}, \quad (9)$$

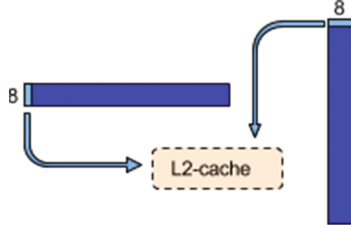
with parameter  $t$  denoting the block size (number of rows/columns).

Each executable block relates to calculation of  $C_j^i = A^i B_j$ . Since the submatrices  $C_j^i$  do not intersect, the operating results for different blocks are independent. The total number of executable blocks in the algorithm is  $N_b = \frac{MK}{t^2}$ .

Now let's turn to the threads inside the executable blocks. Each thread calculates a product of a row of  $A^i$  by a column of  $B_j$ , which corresponds to one



**Fig. 1.** Calculation in one executable block



**Fig. 2.** “Naive” matrix multiplication algorithm: data loading for one executable block.

element in the submatrix  $C_j^i$ . Thus, the number of threads is equal to  $t^2$  (Figs. 1 and 2).

As follows from the above, the number of executable blocks and the number of threads in blocks depend on the value  $t$ , namely with increasing  $t$  the number of blocks decreases, while the number of threads increases. We propose the following heuristic principle to obtain the optimal value of  $t$ :

*the more blocks, the better.*

Without going into details, we note that a large number of blocks has the following advantages:

1. More multiprocessors on GPU are filled (and more uniformly);
2. The scheduler can more effectively “hide” the time for data and instructions swapping.

But this rule is applicable only for reasonable value of  $t$ , as there are several GPU architecture limitations on number of threads per block and number of blocks for multiprocessor [10]. Formally, the maximum number of blocks is obtained with  $t = 1$ . But there are four objections to this choice.

First, the number of blocks is limited by the computational grid size of the particular GPU. This restriction, however, is not too strong. For example, it can be avoided by considering multiplication of smaller submatrices, such as parts of the rows of  $A$ , and parts of the columns of  $B$ .

Second limitation is the linear dependence of the number of downloads from global memory from  $t$ . Actually, due to high complexity of the calculations with long numbers the loading time does not have a decisive influence.

Third, maximal number of blocks per multiprocessor is limited, so with small number of threads per block total number of threads per multiprocessors would be less than maximal available (occupancy will be low). In this case multiprocessor would worse hide data and instruction fetch.

Fourth reason is provided by the condition that the number of threads in the block must be a multiple of 32 (the number of threads in a warp). Choosing  $t = 1$ , we use only one of the 32 threads that are allocated anyway. This is absolutely ineffective. Therefore, we must chose  $t$  such that  $t^2$  is a multiple of 32 and the optimal  $t$  is  $t = 8$  (the minimal  $t$  such that  $t^2$  is a multiple of 32).

Note, that with  $t = 8$  occupancy of multiprocessor is not limited by maximal number of blocks for the most modern GPU: each block uses 64 threads, maximal number of blocks is equal to 32. So, in this case block number limitation allows to use 2048 threads per multiprocessor and that is exactly limitation of thread number per multiprocessor. For older architectures such  $t$  limits occupancy by maximal number of blocks per multiprocessors, but for this architectures real limiter for occupancy would be number of used registers.

Consider executable block algorithm.

#### **Algorithm 1. Multiplication of $N \times 8$ blocks. “Naive approach”**

1. *Two  $1 \times 8$  vectors are loaded from the device’s memory in the shared memory (could be considered as equivalent of shared L2 cache of multicore CPU) of streaming multiprocessor (SM): one vector corresponds to the column of the row-block, and another is the row of the column-block;*
2. *Each of the 64 threads loads two numbers (elements of a large prime field) into the registers (Cache L1) of its stream processor (SP);*
3. *Each thread executes a product of its own numbers and sums it with the current value of the result;*
4. *Montgomery conversion is performed once at the end of all calculations; the necessary constants are loaded from the constant memory.*

Despite the simplicity of the Algorithm 1, the very possibility of its execution on a GPU is nontrivial. Let us consider the necessary resources for its execution. Each thread is associated with:

1.  $2W + 1$  32-bit registers for storing the result, where  $W$  is the number of 32-bit words necessary for storing elements of a prime field. For example, for a field with 512 bits per element,  $W = 16$ ; for 768-bit field  $W = 24$ , and for 1024-bit field  $W = 32$ .
2.  $2W$  registers for storing the inputs (i.e. elements of the corresponding row and column).

Note that without loss of performance, we can store only one of the input numbers on registers, and load the second one word by word as needed. Therefore, only  $W + 1$  registers are needed to store the inputs.

Thus, even by the most primitive calculations  $3W + 2$  registers per an executable block are needed, that is:

**512-bit field:** not less than 50 registers;  
**768-bit field:** not less than 74 registers;  
**1024-bit field:** not less than 98 registers.

These elementary estimates show that GPUs with 63 registers per thread have a very limited applicability resource. The latter, of course, does not mean that it is impossible to organize calculations on such GPUs. One can certainly get an implementation for any large field by arranging calculations involving additional work with memory. But the aim of our research is to obtain *the maximum acceleration*. Therefore, we are primarily interested in situations without unnecessary obstacles to the most rapid implementation.

**Remark 1.** *We are interested in two types of block operations for the block Lanczos method: block-by-block multiplications in form  $X^TY$ , and block-by-matrix multiplications in form  $XU$  (with  $K \times K$  matrix  $U$ , and  $N \times K$  blocks  $X$  and  $Y$ ). Note that in applications the parameter  $N$ , is usually very large, but the block size  $K$  can be insignificant (for example, about 8). In this case, there are certain difficulties in choosing  $t$ . This is especially characteristic for  $X^TY$  calculation. Indeed, by the above scheme, for  $t = 8$  we get only one executable block. And reducing  $t$  would result in inefficient use of threads.*

*We can partially solve the problem by considering  $X$  and  $Y$  in a form*

$$X = \begin{bmatrix} X_1 \\ X_2 \\ \dots \\ X_l \end{bmatrix}, \quad Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \dots \\ Y_l \end{bmatrix}, \quad (10)$$

*with the same number of rows in each block  $X_i, Y_j$ . Since in this case*

$$X^TY = \sum_{j=1}^l X_j^TY_j, \quad (11)$$

*one can consider computations of the form  $X_j^TY_j$  as executable blocks.*

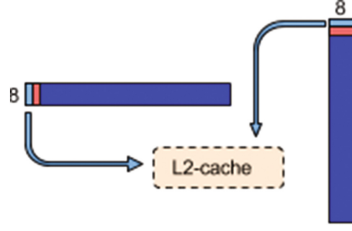
*However, this solution is not perfect. The results of calculations for individual threads are not independent. In addition, it becomes necessary to synchronize the calculations. Both factors negatively affect the efficiency of computing  $X^TY$ . We emphasize, that the problem arises only for small  $K$  and we are mostly interested in situations with  $K$  large. In this case the problem is not so critical.*

Finally, due to the large number of registers in use, the number of threads on SM will be noticeably less than the maximum possible. For older architectures only 20 registers could be used to achieve full occupancy, for new – about 30. However, since the number of downloads is smaller than the number of calculations, only the instructions loading is worse compensated, which leads to uncritical decrease in performance.

For 512 bit numbers achieved occupancy is good enough — performance profiler show that instruction and data fetch are successfully hid and more

than 90 percent of time is spent by arithmetic operations. Nevertheless, in case of fields with more than 512 bits per element for more optimal use of GPU resources, it is necessary to further consider the algorithms that use fewer registers, that is, multiplying long numbers in several stages.

**Matrix Multiplication with Winograd Approach.** The organization of matrix multiplication with Winograd approach is similar to the “naive algorithm” (see Sect. 2.2) (Fig. 3).



**Fig. 3.** Matrix multiplication algorithm with Winograd approach: data loading for one executable block

Analogously to the “naive algorithm” each executive block performs the calculation of  $8 \times 8$  submatrix  $C_j^i = A^i B_j$ , and each thread calculates one element of  $C_j^i$ . The difference is that for Winograd approach a two columns of  $A^i$  and two rows of  $B_j$  are loaded to L2 and L1 Cache. This is necessary for the following

$$\text{elementary calculation} = (a_{i,2k-1} + b_{2k,j})(a_{i,2k} + b_{2k-1,j}). \quad (12)$$

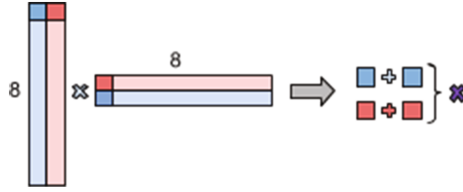
Below we describe the main ideas of the algorithm for GPU.

**Algorithm 2. Multiplication of  $N \times 8$  blocks with Winograd approach**

1. Two  $2 \times 8$  blocks are loaded from the device’s memory in the shared memory (could be considered as equivalent of shared L2 cache of multicore CPU) of streaming multiprocessor (SM) : one block corresponds to the column of the row-block, and another is the row of the column-block;
2. Each of the 64 threads loads 4 numbers (elements of a large prime field) into the registers (Cache L1);
3. Each thread executes (12) for its own 4 numbers and sums it with the current value of the result;
4. Montgomery conversion is performed once at the end of all calculations; the necessary constants are loaded from the constant memory.

Let us consider the necessary resources for the Algorithm 2 execution. Each executable thread is associated with:





**Fig. 4.** Calculations in matrix multiplication algorithm with Winograd approach.

1.  $2W + 1$  32-bit registers for storing the result. Recall that  $W$  is the number 32-bit words required per element of the large field.
2.  $4W$  registers for storing the inputs.

Analogously to the “naive algorithm”, only  $2W + 2$  registers are enough to store the inputs without loss of performance. Moreover we can reduce the number of input data registers to  $W + 3$  by a slight increase of the operations number (not more than  $3W$  extra additions for one multiplication of numbers) (Fig. 4).

However, the most estimate gives  $4W + 2$  registers per an executable block, that is:

- 512-bit field:** not less than 67 registers;
- 768-bit field:** not less than 99 registers;
- 1024-bit field:** not less than 131 registers.

Note that Winograd algorithm requires a larger number of registers, so the need of economical algorithms for it is more critical.

### 2.3 Important Realization Details

An important feature of GPU is the instruction *madc* in the pseudo-assembler (CUDA PTX). This instruction multiplies two numbers with obtaining the first or the last word of the result, and adding it to the third number, taking into account the carry flag. Also it can change the carry flag in case of overflow. This makes it easy to implement the arithmetic with numbers from large prime finite fields [8].

Note that for architectures of the second and third generations, the instruction for 32-bit numbers is translated into assembler instruction which is slower than 32-bit instructions with floating-point numbers (2 – 3 times for the second generation, and 6 times for the third one). For the newer architectures, it is translated into a set of 16-bit instructions that are executed with the same speed as 32-bit floating-point instructions. Thus, in general, the instruction is executed 4 times slower than the instructions with a floating point numbers. And the peak performance of calculations with long numbers is 2 – 6 times lower than the one for floating-point numbers.

However, the performance of CPUs for this task is also far from peak:

- due to the lack of integer instruction for simultaneous multiplication and addition;
- due to the lack of a vector instructions for addition with a carry flag, and for multiplication with obtaining the major word.

As a result, for modern CPUs performance for long arithmetic is 8 - 32 times below the peak performance for 32-bit floating point numbers. Thus, theoretically, in case of such tasks GPU should be so many times faster than CPU, as GPU single-precision peak performance is higher than CPU one. Thus, theoretically, in case of such tasks the gain of GPU performance to CPU performance coincides to the proportion of the peak performances for single-precision tasks.

An important task for the GPU programming is to get rid of branches. Since all threads within a group (warp) must perform the same instruction, branching (with threads executing different branches) is converted to a sequential code, where each thread executes all branches. This leads to more registers and slower execution.

For matrix multiplication over a large prime field, such branching occur only at the stage of reduction, and can significantly affect the performance only for small block sizes. These branching compare two long numbers, and subtract if the first one is greater. However, it is easy enough to get rid of it. To do this, we subtract the second number from the first one, and then add the second number multiplied by the carry flag occurred in the subtraction.

## 2.4 Numerical Experiments

We compare results for CPUs and GPUs of different generations on the following problems:  $2^{21} \times K$  block by  $K \times K$  matrix multiplication (with  $K = 8, 16$ ), and multiplication of square matrices of order 1024.

We use implementations of Winograd approach and Strassen method for CPU (Strassen only for square matrix multiplication), and “naive” implementation and Winograd method for GPU.

The experiments were performed on the following devices (note, that due to frequency boost technologies peak performance of the newest hardware is given approximately):

- 4-core CPU Intel Core i5-4440, 3.1 GHz, power consumption 84W.  
It is CPU of 4th generation of Intel Core microarchitecture. From that generation (and till the latest available at the moment) CPU core could execute per clock 2 fused multiply-add vector instruction with 256-bit vector.  
For single precision floats each such instruction performs 16 floating point operation (8 multiplications and 8 additions). Thus, single core executes up to 32 floating point operations per clock, 4 cores — 128 flop per clock.  
As considered CPU has 4 cores and its clock frequency is 3.1 GHz, its theoretical single precision peak performance is 396.8 Gflops.
- Nvidia Tesla C2070, power consumption 250W, compute capability 2.0, peak single precision performance — 1.03Tflops.

- Nvidia Tesla K40, power consumption 235W, compute capability 3.5, peak single precision performance — 4.2Tflops.
- Nvidia GeForce GTX 1050, power consumption 75W, compute capability 6.1, peak single precision performance — 2 Tflops.

Results of multithread experiments on CPU are presented in Table 1.

The results of operations on GPU are given in Table 2 ( $2^{21} \times 8$  block by  $8 \times 8$  matrix multiplication), in Table 3 ( $2^{21} \times 16$  block by  $16 \times 16$  matrix), and in Table 4 (for matrices of order 1024).

The considered large prime fields required 512 bits, 768 bits, and 1024 bits per element.

**Table 1.** Time for matrix multiplications on CPU (sec.)

Matrix size	$2^{21} \times 8$	$2^{21} \times 16$	$1024 \times 1024$	$1024 \times 1024$ , Strassen
512 bits	3.98	13.41	20.92	12.62
768 bits	7.24	24.28	39.53	23.57
1024 bits	12.6	54.97	69	40.9

**Table 2.** Time for  $2^{21} \times 8$  block by  $8 \times 8$  matrix multiplications on GPU (sec.)

GPU	C2070	K40	GTX1050
Naive algorithm, 512 bits	0.35	0.28	0.41
Winograd approach, 512 bits	0.26	0.19	0.28
Naive algorithm, 768 bits	0.85	0.58	1.15
Winograd approach, 768 bits	0.8	0.63	1
Naive algorithm, 1024 bits	2.04	1.06	2.83
Winograd approach, 1024 bits	1.53	1.07	2.12

**Table 3.** Time for  $2^{21} \times 16$  block by  $16 \times 16$  matrix multiplications on GPU (sec.)

GPU	C2070	K40	GTX1050
Naive algorithm, 512 bits	1.31	0.89	1.56
Winograd approach, 512 bits	0.89	0.6	0.95
Naive algorithm, 768 bits	3.1	2	3.88
Winograd approach, 768 bits	2.86	2.07	3.5
Naive algorithm, 1024 bits	7.59	3.91	10.82
Winograd approach, 1024 bits	5.47	3.57	6.8

**Table 4.** Time for  $1024 \times 1024$  matrix multiplications on GPU (sec.)

GPU	C2070	K40	GTX1050
Naive algorithm, 512 bits	2.38	1.57	2.53
Winograd approach, 512 bits	1.48	0.91	1.42
Naive algorithm, 768 bits	5.75	3.67	6.49
Winograd approach, 768 bits	5.38	3.31	5.52
Naive algorithm, 1024 bits	13.55	7.16	18.11
Winograd approach, 1024 bits	9.37	5.77	9.74

It is noticeable that, due to the use of a larger number of registers, the Winograd approach usually accelerates the computation much less than twice, especially in case of 768 bit numbers. In addition, the advantage of the GPU over the CPU becomes smaller with increasing the field sizes. This proves the necessity of algorithms for long numbers that require smaller number of registers (with multiplying via several stages).

Nevertheless, all the GPUs significantly outperform CPUs for this problem both in terms of computing speed and performance per watt of power. Note that the algorithm used on CPU is quite efficient, and although the CPU is not the most modern, but has almost the same performance on this task as the most modern Intel CPUs (especially at the same clock frequency).

Now consider the results on the Tesla C2070 adapter for 512 bit numbers. This accelerator is similar to the Tesla X2070, which is used on “Lomonosov”. Matrix multiplication with Winograd method is 15 times faster than on CPU. Note, than CPU used in our experiments is even slightly faster on such task than 2 Intel Xeon X5570 4-core CPUs (which “Lomonosov” node contains). It has a slightly higher clock frequency and the execution of 64-bit instructions ADC and MUL (which dominate in the algorithms) requires 2 and 3 times less cycles, respectively.

Thus, the matrix multiplication on GPU of supercomputer “Lomonosov” will be no less than 15 times faster than the one on its CPU. This means that in case of the same time spent on the above operations, the block size in the algorithm can be increased in 15 times, and the time spent for data exchanges will be reduced approximately in 15 times too.

For a linear system of order about 2 million, with 82 nonzero elements, over a large simple field of size 512 bits, on 128 nodes of “Lomonosov”, the time for data exchanges was about 55%. Thus, the implementation of matrix operations on the GPU reduce calculation time in approximately 2 times.

### 3 Conclusion

The possibility of using GPU for a significant acceleration of computations with dense matrices and blocks with elements from the large prime fields is experimentally substantiated. The implementation of “naive algorithm” and the algorithm

using the Winograd approach for multiplication number reduction are described. Numerical simulations were made for various graphics accelerators architecture and performance. It is shown that for the prime fields with more than  $2^{512}$  elements, in order to obtain the greatest possible acceleration, the multi-stage algorithm should be implemented for the multiplications of long numbers.

**Acknowledgments.** The work was supported by the Russian Science Foundation, grant 14-11-00806.

## References

1. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J.W., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., te Riele, H., Timofeev, A., Zimmermann, P.: Factorization of a 768-Bit RSA modulus. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 333–350. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-14623-7\\_18](https://doi.org/10.1007/978-3-642-14623-7_18)
2. Thome, E., et al.: Factorization of RSA-704 with CADO-NFS. Preprint, pp. 1–4 (2012)
3. Dorofeev, A.Ya.: Vychislenie logarifmov v konechnom prostom pole metodom lineinogo resheta. [Computation of logarithms over finite prime fields using number sieving]. Trudy po diskretnoi matematike, vol. 5. pp. 29–50 (2002)
4. Dorofeev, A.Y.: Solving systems of linear equations arising in the computation of logarithms in a finite prime field. Math. Aspects Crypt. **3**(1), 551 (2012). Russian
5. Popovyan, I.A., Nestrenko, Y.V., Grechnikov, E.A.: Vychislitelno slozhnye zadachi teorii chisel. Uchebnoe posobie [Computationally hard problems of number theory. Study guide] Publishing of the Lomonosov Moscow State University (2012)
6. Zamarashkin, N., Zheltkov, D.: Block Lanczos–Montgomery method with reduced data exchanges. In: Voevodin, V., Sobolev, S. (eds.) RuSCDays 2016. CCIS, vol. 687, pp. 15–26. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-55669-7\\_2](https://doi.org/10.1007/978-3-319-55669-7_2)
7. Zamarashkin, N.L.: Algoritmy dlya razrezhennykh sistem lineinykh uravneniy v  $GF(2)$ . Uchebnoe posobie [Algorithms for systems of linear equations over  $GF(2)$ . Study guide]. Publishing of the Lomonosov Moscow State University (2013)
8. Efficient basic linear algebra operations for solution of large sparse linear systems over finite fields. Russian Supercomputing Days (2016)
9. Nath, R., Tomov, S., Dongarra, J.: An improved MAGMA GEMM for Fermi graphics processing units. Int. J. High Perform. Comput. Appl. **24**(4), 511–515 (2010)
10. Nvidia Corporation, CUDA C. Programming guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide>

Supercomputing

Third Russian Supercomputing Days, RuSCDays 2017,

Moscow, Russia, September 25–26, 2017, Revised

Selected Papers

Voevodin, V.; Sobolev, S. (Eds.)

2017, XVI, 532 p. 232 illus., Softcover

ISBN: 978-3-319-71254-3