Block Lanczos-Montgomery method over large prime fields with GPU accelerated dense operations

Nikolai Zamarashkin ✉ and Dmitry Zheltkov

INM RAS, Gubkina 8, Moscow, Russia
{nikolai.zamarashkin,dmitry.zheltkov}@gmail.com
http://www.inm.ras.ru

Abstract. Solution of huge linear systems over large prime fields is a problem that arises in such applications as discrete logarithm computation. Lanczos-Montgomery method is one of the methods to solve such problems. Main parallel resource of the method us the size of the block. But computational cost of dense matrix operations is increasing with block size growth. Thus, parallel scaling is close to linear only while complexity of such operations are relatively small. In this paper block Lanczos-Montgomery method with dense matrix operations accelerated on GPU is implemented. Scalability tests are performed (including tests with multiple GPU per node) and compared to CPU only version.

Keywords: linear systems over prime fields · parallel computations · GPGPU

1 Introduction

The papers [1, 2] describe the block Lanczos method for solving huge sparse linear systems over large prime finite fields that was developed in INM RAS. It was shown that the parallel efficiency of this method is limited by the unscalability of operations with the dense matrices and blocks. A qualitative explanation of this property is not so difficult.

Let K be a block size. Assume that the number of independent computational nodes is proportional to K. For the system size N the number of iterations does not exceed $\frac{N}{K}$. The complexity of operations with the dense $K \times K$ matrices and $N \times K$ blocks is proportional to NK^2. This kind of operations is performed on every iteration.

Thus, the following complexity estimate is valid:

$$\{\textbf{Time for dense operations}\} \sim \frac{N}{K} NK^2/K \sim N^2. \qquad (1)$$

In spite of the fact that the calculations with dense matrices are usually ideally parallelized, the complexity estimate is *non-scalable by the number of nodes.*

In practice, things are as follows. While the block size K is small, the complexity of the dense block operations is hidden by the greater complexity of the huge sparse matrix by $N \times K$ block multiplications. Since the operation of sparse matrix by block multiplication has a significant parallel resource, for the small values of K the method has almost linear scalability [1, 2]. However, the scalability is close to linear only up to block sizes about $10 - 20$, which corresponds to the simultaneous use of up to $100 - 200$ independent nodes [2]. The only way to extend the *linear scalability* area of the parallel block Lanczos method is *reducing the time for the operations with the matrices and blocks.*

In this paper we consider the use of GPUs to accelerate operations with dense matrices and blocks with elements from a large finite field. The principal possibility of significant acceleration of this kind of calculations using GPU was proved in [2]. The considered case of the large block size K varying from 100 to 1000 is important from the theoretical point of view. In practice, however, such block sizes are still very rare and purely exploratory in nature. At the same time, blocks of small sizes $K < 10$, are generally used. It will be shown that even in this case the use of GPUs allows to significantly speed up computations.

The paper describes the GPU implementations of two algorithms:

1. multiplication of "tall" $N \times K$ block by $K \times K$ matrix (section 2);
2. multiplication of $K \times N$ by $N \times K$ blocks (section 3).

From the mathematical point of view, we solve the problem of mapping an algorithm onto non-trivial parallel computing systems. And while choosing an algorithm the simplest one is preferable.

To reach high efficiency of GPU computations for our problem we must take in account properties of arithmetic operations in large prime fields, especially the multiplication of two field elements. Due to the design features of GPUs, their maximum performance is achieved if there is no conditional branches in the code. But the formal description of the multiplication and addition algorithms in the large prime fields implies the use of conditional branching operators, for example, in transfer bits [3]. The 3.3 section describes some of the techniques that are used to exclude conditional branches. The corresponding part of software implementation is written in Cuda PTX.

For more efficient use of GPUs computations are performed asynchroniously, where possible. Also, support of multiple GPUs per node is implemented.

The numerical experiments (see section 4) were performed with and without CUDA. To distinguish between the implementations the following notations are used:

– the software without CUDA is denoted by **P0**;
– the software using single GPU per node is denoted by **P1**.
– the software using 2 GPUs per node is denoted by **P2**.

2 Multiplication of dense $N \times K$ block by $K \times K$ matrix for small K

2.1 General algorithm structure

Let $A \in \mathbb{F}^{N \times K}$ be a block and $B \in \mathbb{F}^{K \times K}$ be a square matrix with elements in a large prime field $\mathbb{F}$. We assume that the elements of $\mathbb{F}$ are specified using W computer words (in the experiments $W = 8$, and the size of the machine word is 64 bits), and that A, B, and the resulting block $C = AB$ are stored in the global memory of the GPU.

The purpose of this section is to describe the effective implementation of the *naive algorithm* of multiplying A by B on GPUs.

As a rule, effective implementation on GPU implies a large number of independent *executable blocks.* This is our immediate goal.

We represent the block A as a union of blocks $A_i \in \mathbb{F}^{64 \times K}$ (see figure 1). Each $64 \times K$ matrix block is associated with an independent executable 64-thread block on GPU. Elements of A_i are loaded into the shared memory by column vectors of 64 elements; and from B just one element is loaded. The independent threads multiply 64-column of A by the number from B, and the result is collected in a column of C.

Now we give a detailed description of the $N \times K$ block by $K \times K$ matrix multiplication algorithm.

Algorithm 1. $N \times K$ block by $K \times K$ matrix multiplication. "Naive" approach.

1. *Loop over the block size;*
2. *Load a column of A and an element of B into the shared memory;*
3. *each of the 64 threads loads on the registers of its Stream processor (SP) the following two $\mathbb{F}$ numbers: the number stored in the shared memory, and the column element corresponding to the thread number;*
4. *each thread executes the multiplication of its pair, and adds it to the current value of the result;*
5. *The Montgomery reduction is performed once at the end of all calculations. The necessary constants are loaded from the constant memory.*

2.2 Some details of block-by-matrix multiplication

In order to achieve the optimal performance, one should pay attention to (a) the organization of data loading on the GPU registers; (b) the organization of downloading from the registers into memory. The optimal loading from the global memory is possible only if the loaded data is stored in 32-byte blocks. And the efficiency of loading becomes higher if the neighboring threads of the same warp use different memory banks. In other words, the numbers of machine words used by warp should give the maximum possible number of different residues when dividing by 32.

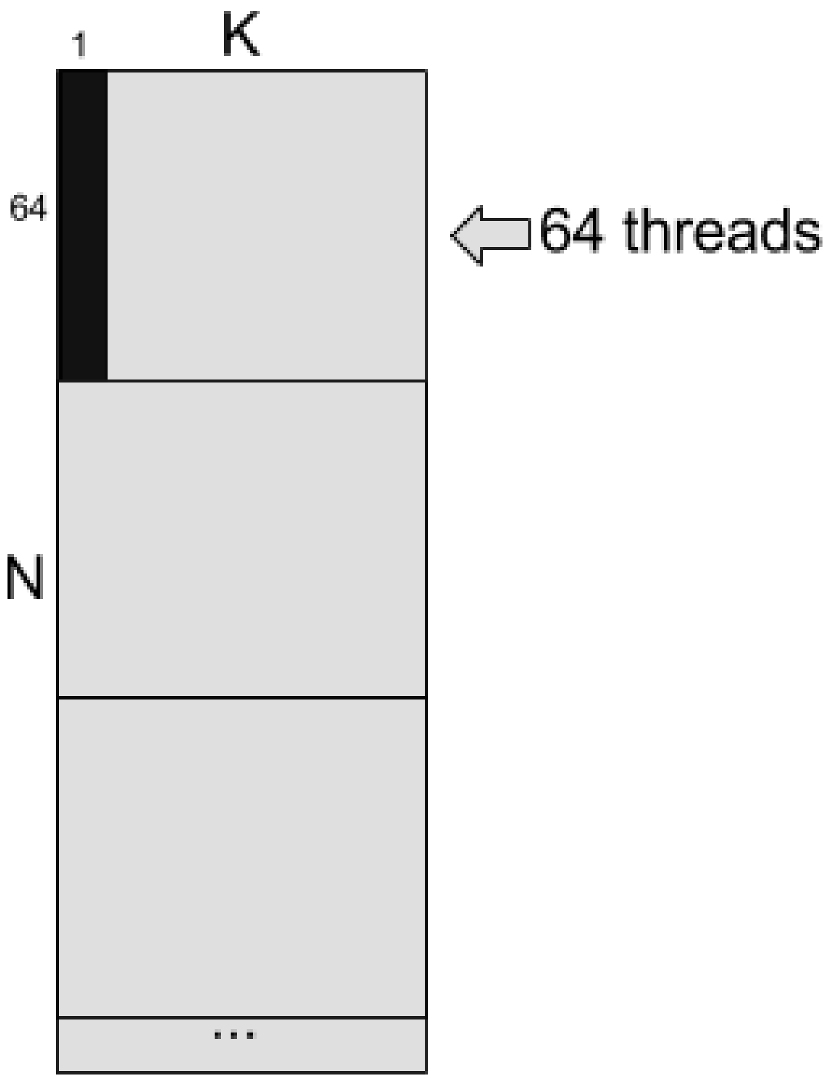

Fig. 1. *Representation of A for the executable blocks.*

A similar observation is true for loading data from shared memory to the registers. If the threads of one warp load either the same element, or elements from different memory banks, the best results are obtained.

To make loads from global memory faster data matrices is stored columnwise. Indeed, in this case each of 64 SP could load at once one of the consecutive 64 machine words. Thus, all threads in warp use different memory banks. After performing such operation W times all needed 64 large numbers are stored in shared memory.

Another important detail is that pseudo-assembler has instructions for the combined operations of increased accuracy. For example, *madc.lo.cc* adds the lower word of the product of two numbers to the third number, taking into account the transfer flag, and generates the transfer flag of the result. The use of such instructions allows to avoid branching, and efficiently implement the arithmetic in large prime fields on GPU.

Usage of multiple GPUs on such operation is straitforward — each GPU stores and computes only part of block.

3 Multiplication of dense $N \times K$ blocks

3.1 General algorithm structure

In this section, multiplication of two dense $N \times K$ blocks with the resulting square $K \times K$ matrix is considered. It will be shown that this operation is less convenient for implementation on the GPU: it is difficult to create a large number of independent blocks (if the block size is not large enough). Nevertheless, even in this case (including $K = 1$), it is possible to obtain a significant acceleration.

Consider the product of two $N \times K$ blocks A and B. Assume that the blocks are divided into sub-blocks (see figure 2). Namely, every column of size N is considered as a union of r short vectors.

In calculating $A^T B$, the corresponding short columns are multiplied by each other, giving a number (element of $\mathbb{F}$). Since in each case only one of r parts of product is obtained, the calculations are not absolutely independent. After obtaining the results for all pairs of corresponding short vectors, one requires to make a reduction (sum of r numbers). In ideal case for binary tree reduction $r = 2^k$.

The choice of r affects the efficient use of GPU resources. At the beginning one need to get a sufficiently large number of independent *executable blocks.* The high capacity of modern GPU means the simultaneous use of several tens of thousands threads. Since the number of threads in our the executable block is 64, the number of independent executable blocks should be about 1000 (or even higher). In our case it is easy to see, that

$$\{\textbf{Number of blocks}\} = K^2 r \geq 1000. \tag{2}$$

Therefore, for $K < 30$ one should not split blocks into parts. Note that for convenience, we choose the block size that is multiple of the number of threads (64), that is:

$$N \approx 64 l K^2 r. \tag{3}$$

with integer l that defines the number of multiplications of long words computed by each threads.

Note that the algorithm consists of three successive parts:

1. Each thread calculates the sum of l products of long numbers, and the incomplete Montgomery reduction (the result is the number of length $W + 1$

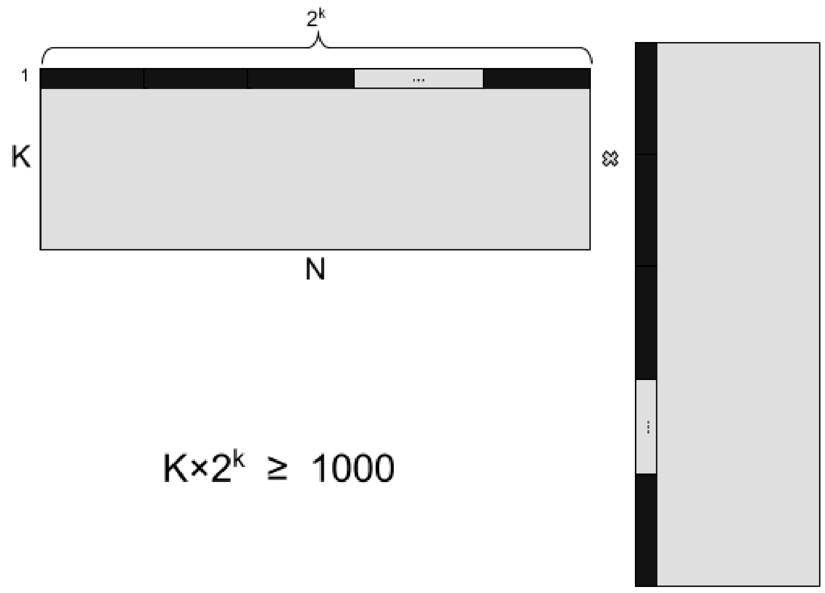

Fig. 2. *Division of independent blocks for calculation $A^T B$.*

). The number obtained by each thread is written to its place in the global memory.
2. Parallel binary tree reduction. Each computational block sums all the numbers necessary for the result. The resulting number (with in length $W + 1$) is written to the global memory.
3. Finding the remainder of dividing each element by prime number of length W.

The second part of the algorithm is standard; its implementation is known. The third one is trivial. In addition, it is assumed that $K \ll N$, and therefore the second and third parts have relatively low complexity. Thus, only the first part is of interest.

Algorithm 2. Multiplication of $N \times K$ blocks.

1. *In loop for l;*
2. *Loading the corresponding bits of the 64×1 vectors of the blocks A and B from the device memory into shared memory of the stream multiprocessor (SM);*
3. *Each of the 64 threads loads two numbers into the registers of its stream processor (SP);*
4. *Each thread executes the multiplication of its pair, and adds it to the current value of the result;*
5. *The incomplete Montgomery reduction is performed once at the end of all calculations; the necessary constants are loaded from the constant memory.*

3.2 Some details of block-by-block multiplication

Our representation of the algorithm in several parts is not arbitrary. Formally, the parts could be combined together. However, the combined algorithm would consume a significant amount of multiprocessor resources, both registers and shared memory. The competition for resources would lead to less efficiency of large field numbers multiplication, while this operation brings the basic complexity of the algorithm.

In addition, in the first computational part it is possible to limit the amount of necessary shared memory by the price of additional synchronization. Namely, one can use only the memory necessary for 64 numbers of W machine words. That is, first a vector from A is loaded into the shared memory; then the required number is loaded into the registers of each thread; and after that a vector from B is loaded into the same place of the shared memory.

When multiple GPUs are available blocks are divided into corresponding number of parts. Each GPU compute its own product. The result (which is equal to sum of the results on all GPUs) is computed on CPU.

3.3 Some details of arithmetic operations in $\mathbb{F}$

Commonly the descriptions of Montgomery reduction, and the reduction modulo a prime number use conditional branches. As for GPUs, the conditions depending on the processed data lead to execution of all paths of the branch by the warp. This significantly reduces the performance of the device. In addition, processing of the different paths of the branch requires additional registers. As a result, fewer threads can created on the Stream multiprocessor, so data and instruction loading is worse hidden. This leads to an additional decrease in performance.

Let us explain on an example how to get rid of branching in Montgomery reduction. A typical case of branching here is the following:

$$\textit{if } a > b \textit{ , then subtract } b \textit{ from } a \textit{ , else do not change } a.$$

Now let's perform the subtraction $a - b$ with the transfer flag generating. First we sum two zeros with the transfer flag, and set the result c (so now c contains the flag value). Then we add $c * b$ to $a - b$, getting the correct value of $a - b$. Note that in multiplication c by b, one may calculate just the lower word of the products.

4 Numerical experiments

Before describing the numerical experiments, let's focus on the properties of the block Lanczos method implementation, created in the INM RAS. Namely, the implementations have two main parallel resources.

First one is the efficient parallel procedure for sparse matrix by vector multiplication [6]. The matrix is split into blocks, which are processed in parallel.

However, this parallelism is limited. For example, the time for data exchanges increases as the node number grows.

The second parallelism resource is connected with the block size K (note that the known implementations [4, 5] do not have it). The independent computations are executed for each individual vector in the block. In [1] an implementation proposed such that increasing K leads to the decrease of data exchanging time proportional to K. Thus, the block size is not only an independent parallel resource, but also expands the parallelism of sparse matrix by vector multiplication. Increasing K is restricted by the growth of algorithmic complexity for operations with dense matrices and blocks at each iteration. It means that acceleration of block operations is *a priority task for the most efficient parallel implementation.*

We used the following two matrices for the numerical experiments:

1. Matrix 1 (**M1**): (a) size 64446×65541; (b) number of nonzeros 1588524; (c) average number of nonzeros in a row $\rho = 24.65$; (d) 5 dense blocks.
2. Matrix 2 (**M2**): (a) size 2097152×2085659; (b) number of nonzeros 182117529; (c) average number of nonzeros in a row $\rho = 86.84$; (d) 5 dense blocks.

Numerical experiments were performed on the following computer systems:

1. GPU cluster of INM RAS (**T**): each node is equipped with 4-core processor Intel Core i7-960 3,2 GHz and 2 graphical adapters Nvidia Tesla C2070. The nodes are interconnected with Infiniband 10 Gbit/s.
2. Supercomputer "Lomonosov" (**L**): each node is equipped with two 8-cores processors Intel Xeon X5570 2,93 GHz. Some nodes in addition are equipped with graphical adapters Nvidia Tesla X2070. The nodes are interconnected with Infiniband 40 Gbit/s.
3. Supercomputer "Lomonosov-2" (**L2**): each node is equipped 14-cores processors Intel Xeon E5-2697v3 2,6 GHz and with graphical adapter Nvidia Tesla K40M. The nodes are interconnected with Infiniband 56 Gbit/s.

The resuplts are shown in the tables 5, 6, 1, 2, 3, 4.

Table 1 gives the time required to compute one vector of A-orthogonal basis of the Krylov space on cluster **L**. The first column of table indicates the number of nodes. As we see, implementation **P1** with CUDA allows block size increasing easier than implementation **P0**. Thus, the parallel resource associated with the procedure for multiplying extra-large sparse matrix by vector is preserved and can be exploited later. Thus, the almost linear scalability of **P1** persists wider. This conclusion is vividly confirmed in Table 2. Indeed, the results for **P1** show better acceleration relative to the calculations on one node. That is, **P1** has better parallel properties.

Similar results are obtained for the cluster **L2**. Finally, we note that the difference in the results of **P0** and **P1** becomes more and more evident with the growth of K.

On cluster **T** results shows effect of multiple GPU usage. For cluster with more nodes available impact of multiple GPU usage is expected to be much more significant and to allow close to linear parallel scaling for larger K.

Table 1. Cluster **L**. Time spent to calculate one vector of Krylov subspace. The optimal block size K is in brackets.

Nodes \Prog.	**P0, M1,** ms	**P1, M1,** ms	**P0, M2,** s	**P1, M2,** s
1	56.9 (1)	41.1 (1)	3.81 (1)	3.10 (1)
2	29.2 (1)	21.5 (2)	2.07 (1)	1.61 (2)
4	21.8 (1)	11.2 (4)	1.14 (1)	0.815 (4)
8	13.8 (2)	6.9 (8)	0.709 (1)	0.417 (8)
16	8.0 (4)	4.3 (16)	0.401 (4)	0.231 (16)
32	–	–	0.216 (8)	0.128 (32)

Table 2. Cluster **L**. Acceleration compared to one node

Nodes \Prog.	**P0, M1**	**P1, M1**	**P0, M2**	**P1, M2**
1	1	1	1	1
2	1.95	1.91	1.84	1.93
4	2.61	3.67	3.34	3.80
8	4.12	5.96	5.37	7.43
16	7.11	9.56	9.5	13.42
32	–	–	17.64	24.22

Table 3. Cluster **L2**. Time spent to calculate one vector of Krylov subspace. The block size K is in brackets.

Nodes \Prog.	**P0, M1,** ms	**P1, M1,** ms	**P0, M2,** s	**P1, M2,** s
1	38.3 (1)	29.4 (1)	1.66 (1)	1.41 (1)
2	26.1 (1)	19.2 (2)	0.87 (1)	0.725 (2)
4	15.8 (1)	9.4 (4)	0.500 (1)	0.381 (4)
8	9.9 (1)	5.8 (8)	0.314 (2)	0.202 (8)
16	7.3 (1)	4.0 (16)	0.193 (4)	0.119 (16)
32	6.5 (2)	2.8 (16)	0.116 (8)	0.0698 (32)

Table 4. Cluster **L2**. Acceleration compared to one node

Nodes \Prog.	**P0, M1**	**P1, M1**	**P0, M2**	**P1, M2**
1	1	1	1	1
2	1.47	1.53	1.91	1.95
4	2.42	3.13	3.32	3.70
8	3.87	5.07	5.29	6.98
16	5.25	7.35	8.6	11.85
32	5.89	10.5	14.31	20.2

Table 5. Cluster **T**. Time spent to calculate one vector of Krylov subspace. The block size K is in brackets.

Nodes \Prog.	**P0, M1,** ms	**P1, M1,** ms	**P2, M1,** ms	**P0, M2,** s	**P1, M2,** s	**P2, M2,** s
1	87.9 (1)	52.2 (1)	49.8 (1)	5.91 (1)	4.7 (1)	4.65 (1)
2	50.5 (1)	29.9 (2)	28.2 (2)	3.19 (1)	2.48 (2)	2.43 (2)
4	31.1 (1)	16.8 (4)	15.3 (4)	1.79 (2)	1.28 (4)	1.25 (4)

Table 6. Cluster **T**. Acceleration compared to one node.

Nodes \Prog.	**P0, M1**	**P1, M1**	**P2, M1**	**P0, M2**	**P1, M2**	**P2, M2**
2	1.74	1.75	1.77	1.85	1.90	1.91
4	2.82	3.11	3.25	3.30	3.67	3.72

Acknowledgments.

The work was supported by the RAS presidium program №1 "Fundamental Mathematics and its applications".

References

1. Zamarashkin N., Zheltkov D. Block LanczosMontgomery Method with Reduced Data Exchanges. In: Voevodin V., Sobolev S. (eds) Supercomputing. RuSCDays 2016. Communications in Computer and Information Science, vol. 687. P. 15-26. Springer, Cham
2. Zamarashkin N., Zheltkov D. GPU Acceleration of Dense Matrix and Block Operations for Lanczos Method for Systems over Large Prime Finite Field. In: Voevodin V., Sobolev S. (eds) Supercomputing. RuSCDays 2017. Communications in Computer and Information Science, vol 793. P. 14-26. Springer, Cham
3. Zheltkov D.A. Effectivnie basovye operacii lineinoi algebry dlya reshenia bolshyh razrezennyh sistem nad konechnymi polyami. 2016. RuSCDays. (In Russian)
4. Dorofeev A. Reshenie sistem lineinyh uravnenii pri vichislenii logarifmov v konechnom prostom pole. Matematicheskie voprosy kriptographii. 2012. Vol. 3. N. 1. P. 5–51. (In Russian)
5. Popovyan I., Nesterenko Yu., Grechnikov E. Vychislitelno sloznye zadachi teorii chisel. 2012. MSU Publishing. (In Russian)
6. Zamarashkin N. Algoritmy dlya razrezennyh sistem lineinyh uravnenii v GF(2).2013. MSU Publishing. (In Russian)
7. Nath R., Tomov S., Dongarra J. An improved MAGMA GEMM for Fermi graphics processing units. The International Journal of High Performance Computing Applications. 2010. V. 24. N. 4. P. 511-515.
8. CUDA C. Programming guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide