UDC 519.687.1::004

V. V. ROMANUKE
Khmelnitskiy National University

MATLAB GPUARRAY METHOD OPTIMAL USE FOR SQUARE MATRIX PRODUCT

A research of efficient computation of square matrix product on GPU is represented. For this, MATLAB ***gpuArray*** *method is used on three types of NVIDIA® GPU. The method optimal use, if any, requires the matrix order be greater than 120. For a long sequence of products, when order increases, generating matrices directly on GPU is fully inefficient. The efficiency holds if matrices are generated directly on GPU just for a few times.*

Keywords: matrix product, parallelization, MATLAB, ***gpuArray*** *method, running time efficiency.*

В. В. РОМАНЮК
Хмельницький національний університет

ОПТИМАЛЬНЕ ВИКОРИСТАННЯ MATLAB-МЕТОДУ GPUARRAY ДЛЯ ДОБУТКУ КВАДРАТНИХ МАТРИЦЬ

Представляється дослідження ефективного обчислення добутку квадратних матриць на GPU. Для цього використовується MATLAB-метод ***gpuArray*** *на трьох типах NVIDIA® GPU. Для оптимального використання цього методу, якщо таке існуватиме, необхідно, щоб порядок матриці був більший за 120. Для довгої послідовності добутків генерування матриць безпосередньо на GPU є повністю неефективним. Ефективність має місце тоді, коли матриці генеруються безпосередньо на GPU лише декілька разів.*

Ключові слова: добуток матриць, паралелізація, MATLAB, метод ***gpuArray****, продуктивність часу рахунку.*

Problems and tasks of parallelizing computations

Computation is a fundamental routine in observing, modeling, forecasting, controlling, etc. Naturally, any computational routine is desired to be accomplished as rapidly as possible. If the routine has identical or similar sections which are independent, they can be accomplished or processed concurrently. Thus, computation is parallelized and sped up.

Key problems addressed by parallel computing are: 1) running parallel loops; 2) executing batch jobs in parallel; 3) partitioning large data sets. Tasks of parallelizing computations lie mostly in speedup and large array data distribution [1, 2]. However, there is no criterion of whether parallelization is expedient. Not every entry can be parallelized efficiently, after all. Parallelization expedience and its rate are undisclosed even for a rudiment of parallel computing, i. e. for matrix product being the most parallelized event [1, 3, 4].

Analysis of preceding origins on parallelization of matrix computations

Matrix computations relate to the most studied mathematical objects. Matrix product is parallelized on multicore processors, GPUs, and computer clusters. Using GPU is preferable to clustering. And in fact, GPU itself contains multicore processors.

Many different algorithms have been designed for multiplying matrices. The most studied algorithms are iterative algorithm, divide-and-conquer algorithm, sub-cubic algorithms, parallel and distributed algorithms. They have different computational complexity and running time efficiency.

Iterative algorithm directly applies the mathematical definition of matrix multiplication [1, 2]. Divide-and-conquer algorithm is an alternative to the iterative algorithm [5]. It relies on the block partitioning which works for all square matrices whose dimensions are powers of two. At the top of the partitioning, the matrix product consists of eight multiplications of pairs of submatrices, followed by an addition step. The divide-and-conquer algorithm computes the smaller multiplications recursively, using the scalar multiplication as its base case.

Iterative and divide-and-conquer algorithms are straightforward. Sub-cubic algorithms provide better running times than those straightforward ones. The Strassen's algorithm is complex enough, but it is faster for large sized matrices [6, 7]. The Le Gall's algorithm, and the algorithm of Coppersmith — Winograd on which it is based, are similar to Strassen's algorithm: a way is devised for multiplying two $N \times N$-matrices with fewer than N^3 multiplications, and this technique is applied recursively [7, 8]. However, these algorithms are only worthwhile for matrices that are too large to handle on present-day computers [2, 7, 9, 10].

Parallel and distributed algorithms spread computations over multiple processors or over a network. Here, exploiting shared-memory parallelism, the divide-and-conquer algorithm is parallelized for shared-memory multiprocessors. This parallelization algorithm is not practical due to the communication cost inherent in moving data to and from the temporary matrix. A speedup is achieved without using a temporary, though. To some contrary, communication-avoiding and distributed algorithms handle the so-called communication bandwidth. On a single machine this is the amount of data transferred between RAM and cache, while on a distributed memory multi-node machine it is the amount transferred between nodes. In this case, the Cannon's algorithm (known as the 2D algorithm), partitions each input matrix into a block matrix whose elements are $\left(\sqrt{M/3}\right)\times\left(\sqrt{M/3}\right)$-submatrices, where M is the size of fast memory [11]. The naïve algorithm is then used over the block matrices, computing products of submatrices entirely in fast memory. This reduces communication bandwidth to asymptotically optimal size. In a distributed setting with P processors arranged in $\left(\sqrt{P}\right)\times\left(\sqrt{P}\right)$ 2D mesh, one submatrix of the result can be assigned to each processor, and the product can be computed with each processor. This can be improved by

the 3D algorithm, which arranges the processors in a 3D cube mesh, assigning every product of two input submatrices to a single processor. The result submatrices are then generated by performing a reduction over each row. This algorithm can be combined with the Strassen's algorithm to further reduce runtime [12].

A large number of other algorithms use these ones as base. Their efficiency may be improved just on the parallelization paradigm. Thus, modern computational environments are enhanced with GPU device controllers.

Parallelization of matrix computations with MATLAB Parallel Computing Toolbox

MATLAB® is one of the most powerful computational environments supporting GPU devices. The MATLAB environment has `gpuArray` method in MATLAB Parallel Computing Toolbox. This method copies the input numeric data to the GPU. The resulting object is of a class concerning GPU arrays. This object can be operated on by using one of the methods defined for objects of the class.

After matrices-multiplicands are copied to GPU, a matrix product is computed straightforwardly on GPU. It is reputed that matrix computations on GPU are quicker than on CPU owing to parallelization. This is experienced for large sized matrices. But it is uncertain if GPU or, specifically, MATLAB `gpuArray` method is suitable for any matrix computations. The reason is that the likely quicker GPU computations are preceded by copying matrices from CPU to GPU, accomplished factually by `gpuArray`. The copying takes significant time as the CPU-GPU bus bandwidth is not so large. Consequently, optimal parallelization of matrix computations with MATLAB Parallel Computing Toolbox using its `gpuArray` method is to be ascertained. But now we are to solve a simpler problem — to find a threshold which delimits the domain where `gpuArray` method brings gain (speedup), and where it does not. This should be started and executed on the simplest case of matrix product.

Goal and items for its achievement

For MATLAB `gpuArray` method purposed to gain running time efficiency in matrix computations, we have to determine where it is really effective. The pattern for the determination is square matrix product. For achieving this goal, the following items are to be fulfilled:

1. Formalize the problem, i. e. state the goal using mathematical notation.
2. Define the range of square matrices order.
3. Select a few types of GPU available within MATLAB.
4. According to the formalized problem, compose a MATLAB code with cycled square matrix products over the defined range of square matrices order.
5. Appoint minimal number of cycles for initialization and computation which should ensure stable statistical estimation of the running time.
6. Run the MATLAB code for square matrices whose order increases with minimal increment step.
7. Both for CPU and GPU, estimate time for initialization of the matrix elements depending on the order.
8. Both for CPU and GPU, estimate time for square matrix product depending on the order.
9. For each type of the applied GPU, find a subrange of square matrices order, where the GPU running time is shorter than the CPU running time.

Formalization of the running time efficiency problem and the range of square matrices order

Let two $N \times N$-matrices $\mathbf{A} = \left(a_{ij}\right)_{N\times N}$ and $\mathbf{B} = \left(b_{ij}\right)_{N\times N}$ have real-valued entries. For quite certain assignment of these entries, let them be values of standard normal variates:

$$\mathbf{A} \in \mathcal{N}\left(0, 1, N\right) \text{ and } \mathbf{B} \in \mathcal{N}\left(0, 1, N\right) \tag{1}$$

by the infinite set $\mathcal{N}\left(0, 1, N\right)$ of $N \times N$-matrices in which every entry is a value drawn from the standard normal distribution. Before computing product of matrices $\mathbf{A}$ and $\mathbf{B}$ on GPU, they are copied to a GPU device. The copier is a mapping C taking a matrix $\mathbf{Z}$ on CPU and returning the matrix $\mathbf{Z}_{\text{GPU}}$ on GPU, where $\mathbf{Z}_{\text{GPU}} = \mathbf{Z}$. Hence, the GPU running time $t_{\text{GPU}}\left(N\right)$ is sum of three components: period of assignment (1), period of transferring

$$\mathbf{A}_{\text{GPU}} = C\left(\mathbf{A}\right) \text{ and } \mathbf{B}_{\text{GPU}} = C\left(\mathbf{B}\right), \tag{2}$$

and period of the product $\mathbf{A}_{\text{GPU}} \cdot \mathbf{B}_{\text{GPU}}$ computation, denoted by $p_{\text{GPU}}\left(N\right)$. The CPU running time $t\left(N\right)$ is sum of period of assignment (1) and period of the product $\mathbf{A} \cdot \mathbf{B}$ computation, denoted by $p\left(N\right)$.

The matrix initialization on GPU is assignment (1) and transferring (2), whose total time we denote by $\theta_{\text{GPU}}\left(N\right)$. The matrix initialization on CPU is just assignment (1), whose time is $\theta\left(N\right)$. Then running times are

$$t_{\text{GPU}}\left(N\right) = \theta_{\text{GPU}}\left(N\right) + p_{\text{GPU}}\left(N\right), \tag{3}$$

$$t\left(N\right) = \theta\left(N\right) + p\left(N\right). \tag{4}$$

The third way exists for the product $\mathbf{A}_{\text{GPU}} \cdot \mathbf{B}_{\text{GPU}}$ computation when matrices are generated directly on GPU:

$$\mathbf{A}_{\text{GPU}} \in \mathcal{N}\left(0, 1, N\right) \text{ and } \mathbf{B}_{\text{GPU}} \in \mathcal{N}\left(0, 1, N\right). \tag{5}$$

Here, GPU matrix initialization is just assignment (5), whose time is $\theta^{*}_{\text{GPU}}\left(N\right)$, without any transferring. Period of the product $\mathbf{A}_{\text{GPU}} \cdot \mathbf{B}_{\text{GPU}}$ computation is $p^{*}_{\text{GPU}}\left(N\right)$. And the GPU running time $t^{*}_{\text{GPU}}\left(N\right)$ for this way is

$$t^{*}_{\text{GPU}}\left(N\right) = \theta^{*}_{\text{GPU}}\left(N\right) + p^{*}_{\text{GPU}}\left(N\right). \tag{6}$$

The goal is to find those $N \in \overline{\{2, N_{\max}\}}$, at which, individually, the following three inequalities are true:

$$t_{\text{GPU}}(N) < t(N), \quad p_{\text{GPU}}(N) < p(N), \quad p^*_{\text{GPU}}(N) < p(N), \quad t^*_{\text{GPU}}(N) < t(N). \tag{7}$$

Along with (7), the relationship between $p^*_{\text{GPU}}(N)$ and $p_{\text{GPU}}(N)$ ought to be ascertained also. The range $\overline{\{2, N_{\max}\}}$ of square matrices order starts from 2, indisputably. And let the largest order $N_{\max}$ be 800, what is expected to be enough for seeing domains where `gpuArray` method brings speedup gain.

Selection of GPU available within MATLAB for clocking the running time

We select three types of NVIDIA® GPU for clocking their running time of square matrix product computation: 1) GeForce GTS 450 (Figure 1); 2) Tesla K40c (Figure 2); 3) GeForce GT 610 (Figure 3).

```
>> gpuDevice
ans =
  parallel.gpu.CUDADevice handle
  Package: parallel.gpu

  Properties:
                        Name: 'GeForce GTS 450'
                       Index: 1
          ComputeCapability: '2.1'
             SupportsDouble: 1
              DriverVersion: 6.5000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
                 MaxGridSize: [65535 65535]
                   SIMDWidth: 32
                 TotalMemory: 1.0737e+09
                  FreeMemory: 596561920
       MultiprocessorCount: 4
              ClockRateKHz: 1660000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1
  Methods, Events, Superclasses
fx >>
```

Fig. 1. MATLAB object representing GPU device GeForce GTS 450

```
>> gpuDevice(1)
ans =
  parallel.gpu.CUDADevice handle
  Package: parallel.gpu

  Properties:
                        Name: 'Tesla K40c'
                       Index: 1
          ComputeCapability: '3.5'
             SupportsDouble: 1
              DriverVersion: 6.5000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
                 MaxGridSize: [2.1475e+09 65535]
                   SIMDWidth: 32
                 TotalMemory: 1.2079e+10
                  FreeMemory: 1.1952e+10
       MultiprocessorCount: 15
              ClockRateKHz: 745000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 0
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1
  Methods, Events, Superclasses
fx >>
```

Fig. 2. MATLAB object representing GPU device Tesla K40c

```
>> gpuDevice(2)
ans =
  parallel.gpu.CUDADevice handle
  Package: parallel.gpu

  Properties:
                        Name: 'GeForce GT 610'
                       Index: 2
          ComputeCapability: '2.1'
             SupportsDouble: 1
              DriverVersion: 6.5000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
                 MaxGridSize: [65535 65535]
                   SIMDWidth: 32
                 TotalMemory: 1.0737e+09
                  FreeMemory: 848363520
       MultiprocessorCount: 1
              ClockRateKHz: 1620000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1
  Methods, Events, Superclasses
fx >>
```

Fig. 3. MATLAB object representing GPU device GeForce GT 610

Estimation of the running time of square matrix product

The composed MATLAB code is split into two sections. Each section contains initialization of two matrices and their product. The product is cycled for 1000 times what ensures stable statistical estimation of the running time. Within the first section, initialization is cycled as well. Within the second section, initialization is not cycled.

Unexpectedly for the cycled initialization, when matrices are generated directly on GPU due to (5), it takes badly increasing initialization period both for GeForce GTS 450 and Tesla K40c (Figures 4 and 5). Though there is a problem of memory (cache) of MATLAB and GPU device while matrix order monotonously increases, the computation period increases normally. Henceforth only single time initialization matters.

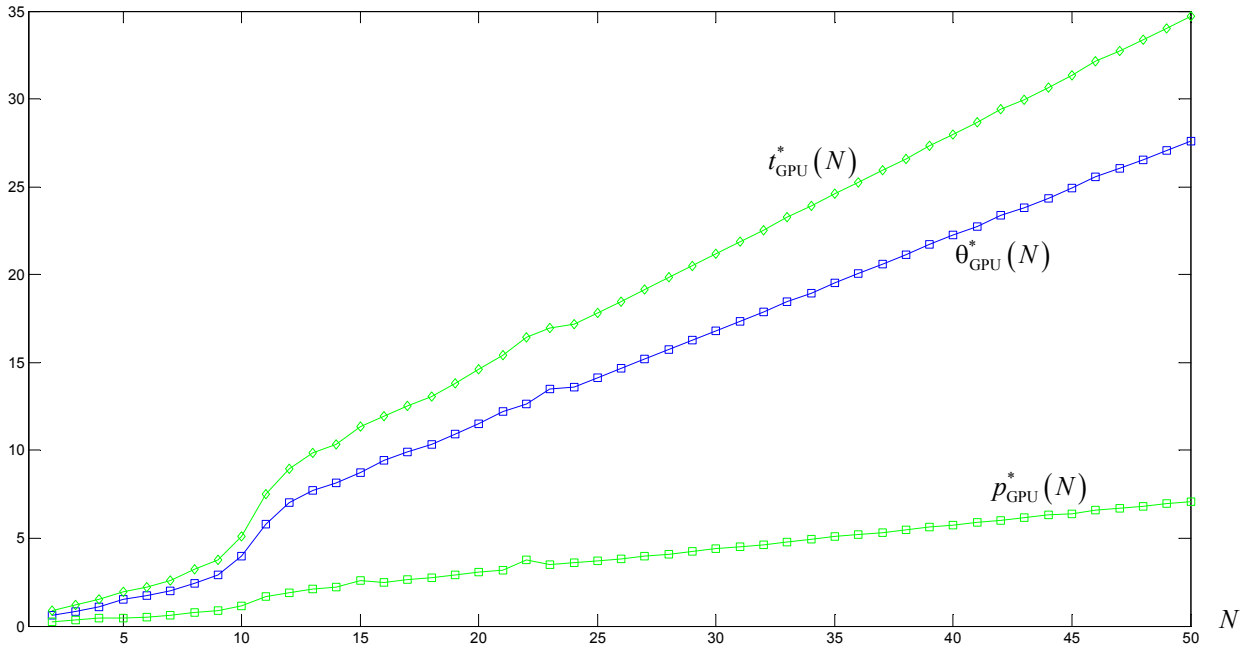

Fig. 4. Badly increasing period of matrix 1000-cycled-initialization for GeForce GTS 450

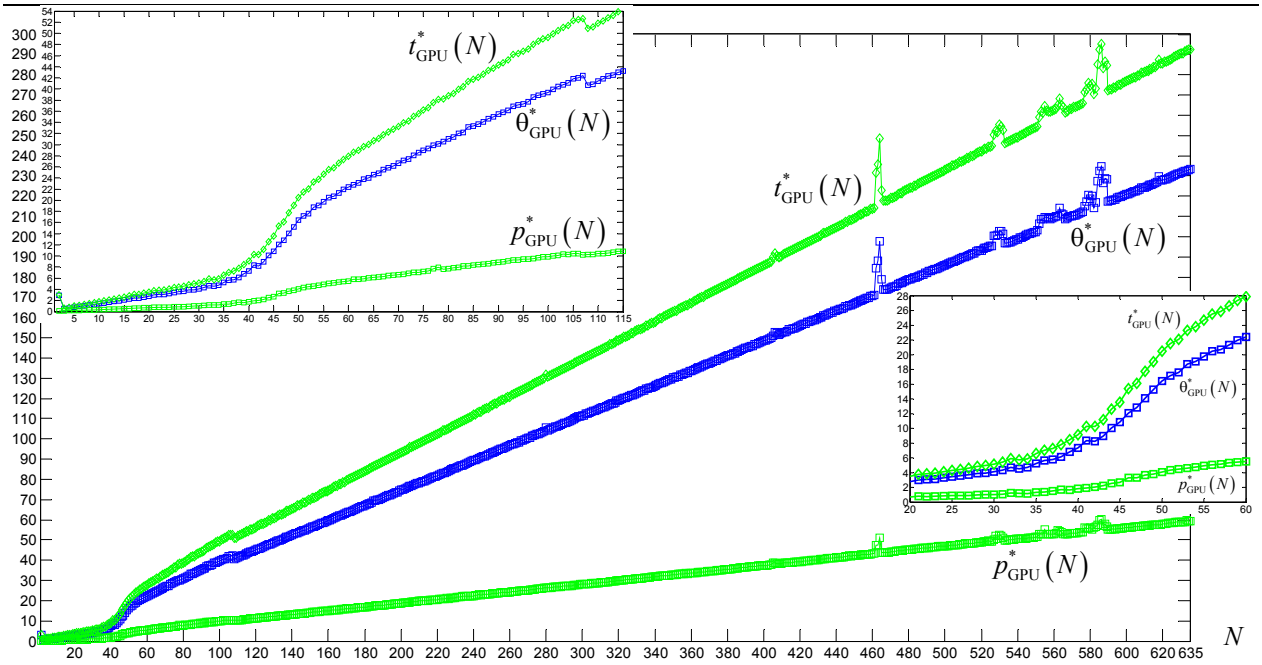

Fig. 5. Badly increasing period of matrix 1000-cycled-initialization for Tesla K40c

When 1000-cycled-initialization is on, each GPU device is conjectured to have its own gpuArray method optimal use. Pure product computation time on GeForce GTS 450 is less than on CPU by $N > 160$ (Figure 6). The same appears for single time initialization (Figure 7). But assignment (1) and transferring (2) to GeForce GTS 450 is obviously longer than assignment (1). Total GPU GeForce GTS 450 running time exceeds CPU running time or comparable till $N = 300$. Period of preassigning singly two matrices is shorter on CPU rather than on GeForce GTS 450 (Figures 6 and 8). Preassigning directly on this GPU is efficient by $N > 410$ (Figure 8).

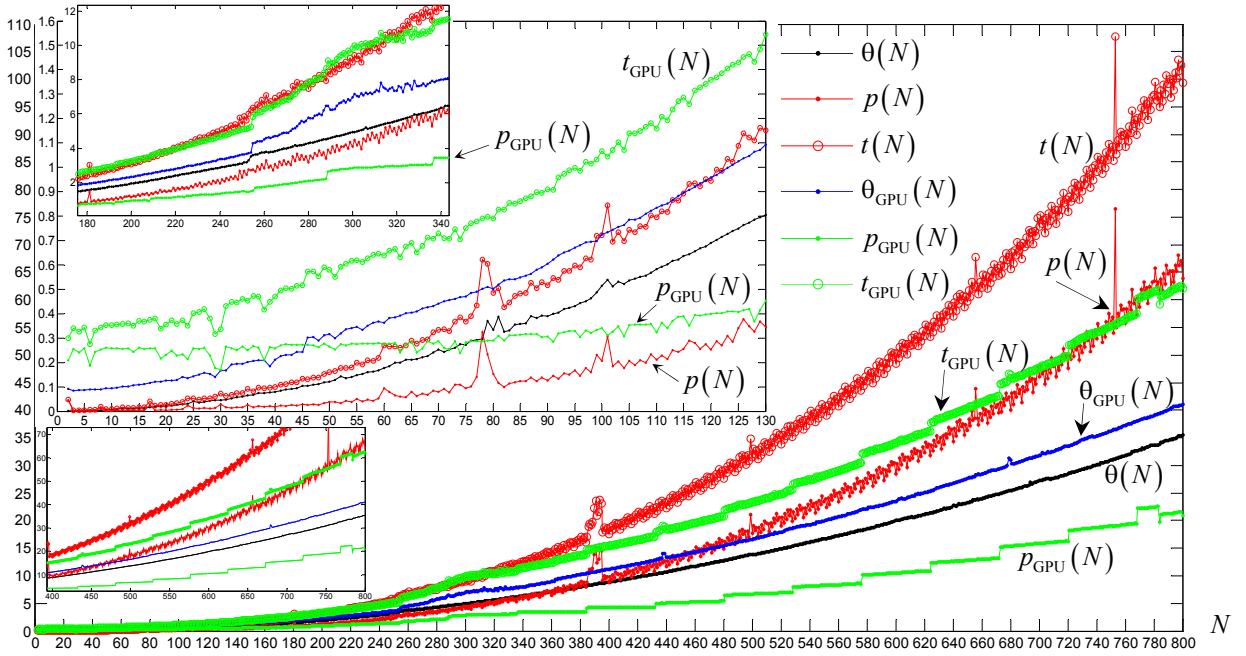

Fig. 6. Estimations of the running time for GeForce GTS 450, when matrices are preassigned during 1000 cycles

On Tesla K40c, pure product computation time is less than on CPU by $N > 130$ (Figures 9 and 10). And $t_{GPU}(N) < t(N)$ by $N > 210$ (Figure 9). Preassigning directly on Tesla K40c is efficient by $N > 120$ (Figure 11). Pure assignment (1) and transferring (2) is inefficient because gpuArray method itself takes some period [13].

The worst testing results are for GeForce GT 610. Figures 12 and 13 expose totally noneffective use of gpuArray method. Period of preassigning singly two matrices on GeForce GT 610 is weakly and unstably competitive (Figure 14) by $N \in (180; 260)$. These unexpectedly poor results may be caused by that both GeForce GT 610 and Tesla K40c were taking their shares on the same CPU and MATLAB session. Nevertheless, this confirms inefficiency of GeForce GT 610, because Tesla K40c is efficient anyhow by $N > 210$.

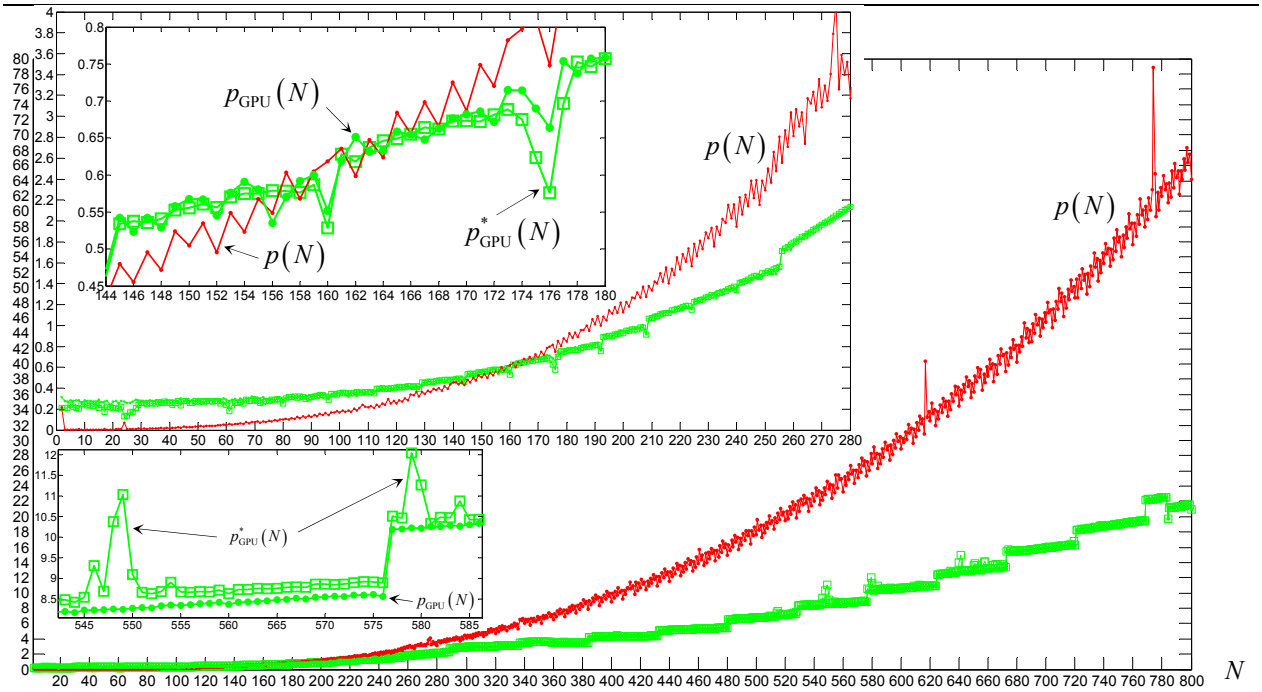

Fig. 7. Estimations of the pure product computation time for GeForce GTS 450, when matrices are preassigned singly

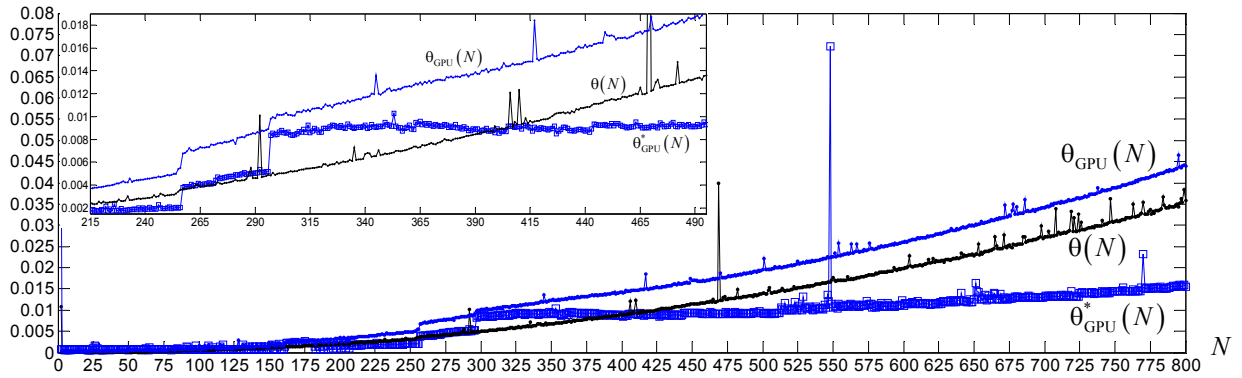

Fig. 8. Period of preassigning singly two matrices on GeForce GTS 450

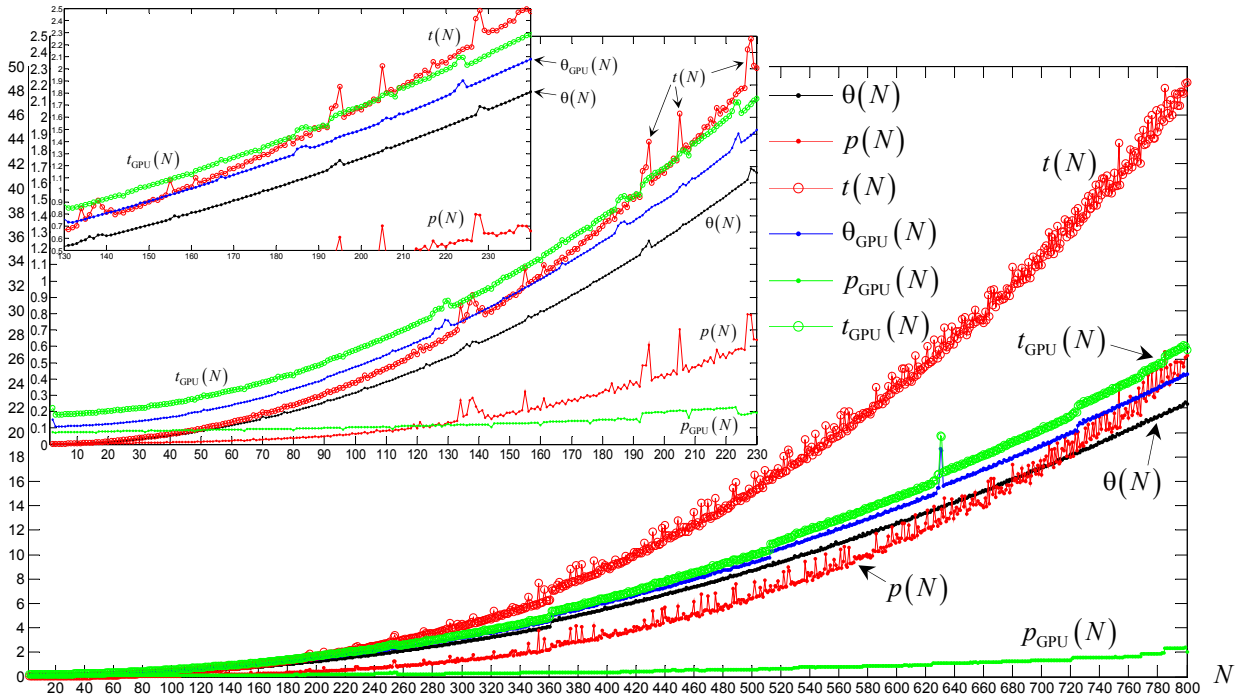

Fig. 9. Estimations of the running time for Tesla K40c, when matrices are preassigned during 1000 cycles

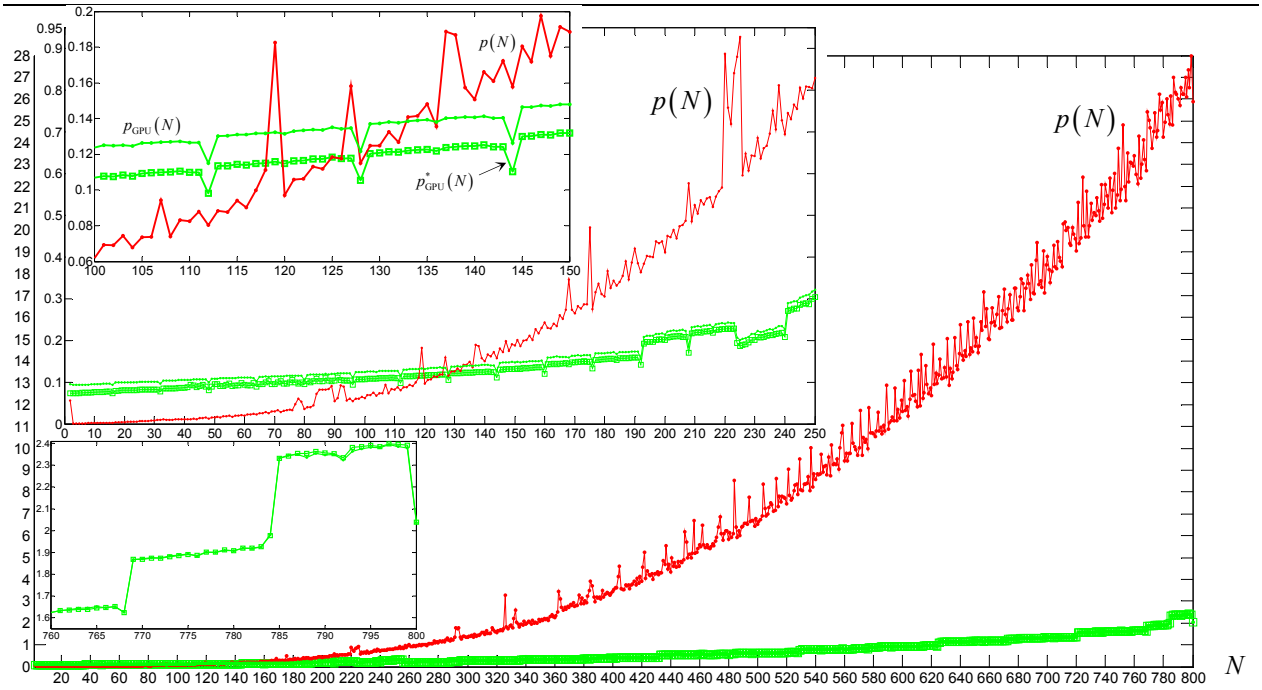

Fig. 10. Estimations of the pure product computation time for Tesla K40c, when matrices are preassigned singly

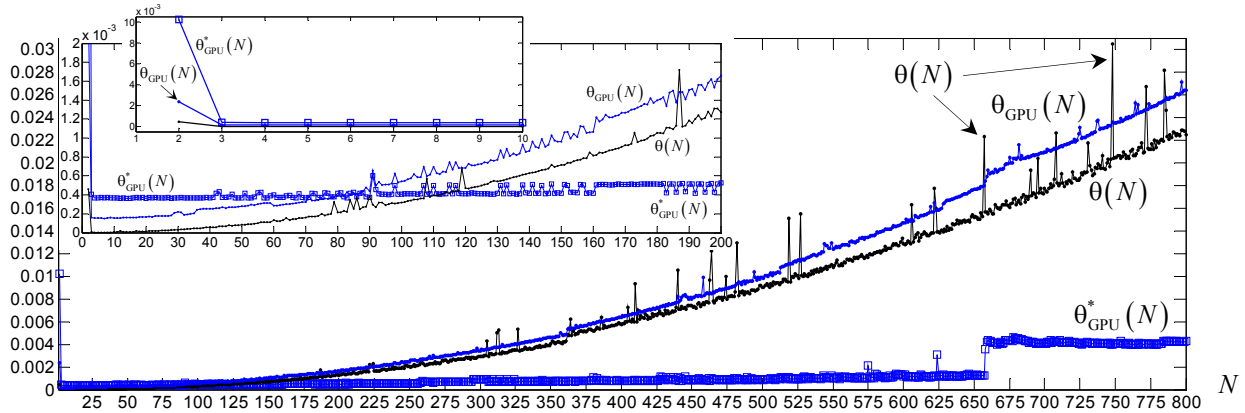

Fig. 11. Period of preassigning singly two matrices on Tesla K40c

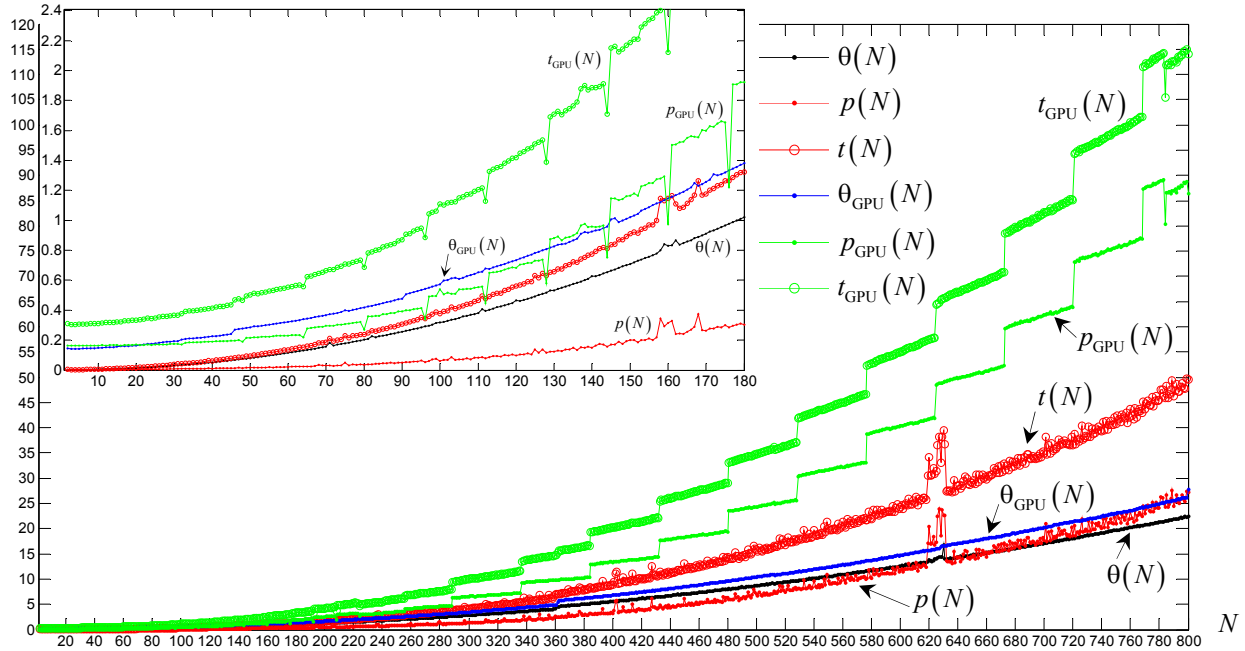

Fig. 12. Estimations of the running time for GeForce GT 610, when matrices are preassigned during 1000 cycles

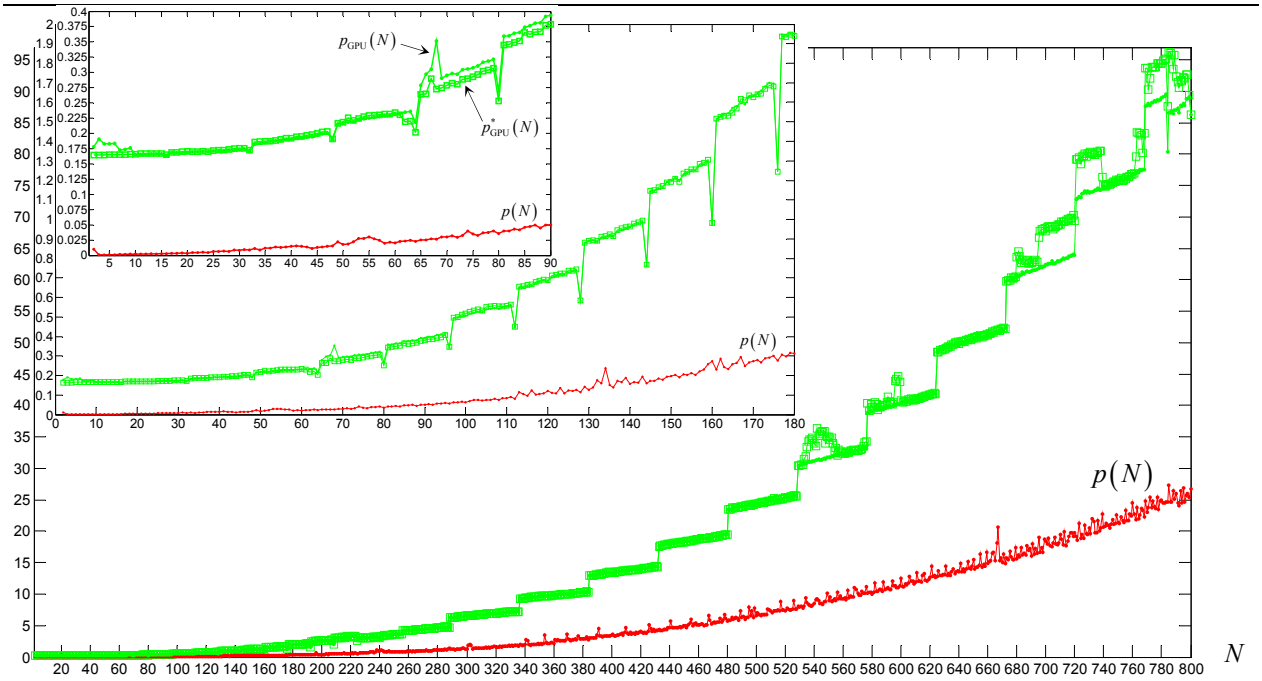

Fig. 13. Estimations of the pure product computation time for GeForce GT 610, when matrices are preassigned singly

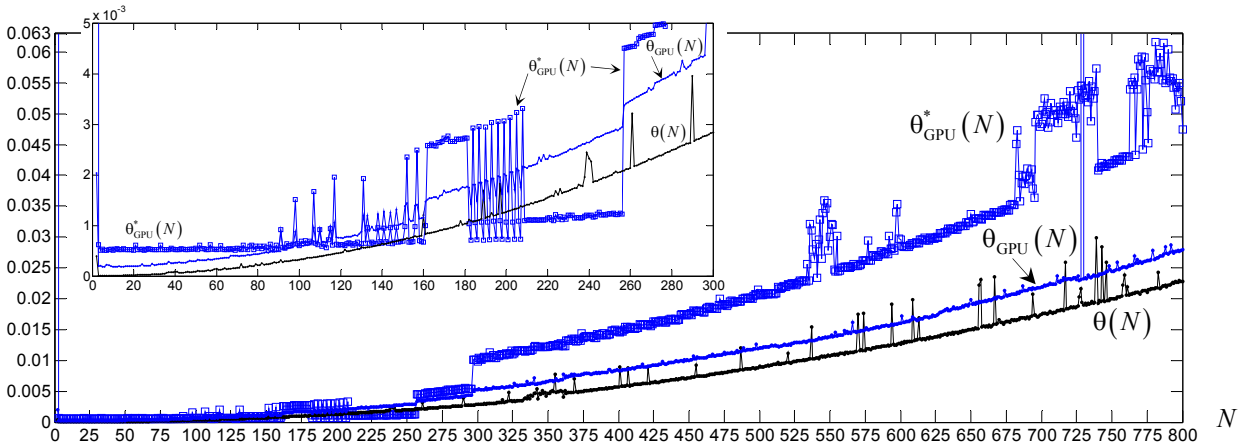

Fig. 14. Period of preassigning singly two matrices on GeForce GT 610

Obviously, the running time on GeForce GT 610 is much longer than the CPU running time. So, MATLAB gpuArray method is not suitable for any square matrix computations on this GPU device. If $N > 210$ then gpuArray method for Tesla K40c is recommended to use. If matrices are already on Tesla K40c and their order is greater than 130, then using this GPU device is efficient also. For pure product computation, efficiency of GeForce GTS 450 is roughly 10 times less than efficiency of Tesla K40c — compare polylines $p_{GPU}(N)$ and $p^*_{GPU}(N)$ in Figures 6 and 7 to polylines $p_{GPU}(N)$ and $p^*_{GPU}(N)$ in Figures 9 and 10. At that, GeForce GTS 450 is optimally used with gpuArray method when matrix order is greater than 160.

Conclusion

Subranges of square matrices order, where the GPU running time is shorter than the CPU running time, if any, seem to start from a hundred or a few hundreds. No reason to think that the subranges have limits towards infinity — the span between the couples of dependencies

$$\{t_{GPU}(N), t(N)\},$$

$$\{t^*_{GPU}(N), t(N)\},$$

$$\{p_{GPU}(N), p(N)\},$$

and

$$\{p^*_{GPU}(N), p(N)\}$$

is only increasing (Figures 6, 7, 9, 10).

Unlike GeForce GTS 450 and Tesla K40c, GeForce GT 610 is out of competitiveness (Figures 12 and 13). Even if this is caused really by peculiarities of two GPU devices on the same CPU and MATLAB session, common quality of GeForce GT 610 is questionable and problematic anyway.

MATLAB gpuArray method is optimally used with more powerful GPUs such as Tesla K40c. Comparison of GeForce GTS 450 to Tesla K40c hints that on powerful GPUs gpuArray method brings speedup earlier — for the lesser matrix order.

For the tested GeForce GTS 450 and Tesla K40c, MATLAB gpuArray method optimal use, if any, requires the matrix order be greater than 120. And for a long sequence of products, when order increases, generating matrices directly on these GPUs is fully inefficient. The efficiency holds if matrices are generated directly on GeForce GTS 450 or Tesla K40c just for a few times. Therefore, the running time is shortened when matrices are already on GPU.

In general, effectiveness of GPU computations strongly depends upon the input data representation. We should combine arrays into larger ones, if it is possible. And for nonsquare matrix product, MATLAB gpuArray method may have specific optimal use. Especially, when numbers of lines and columns are very different.

References

1. Petersen W. P. Introduction to Parallel Computing: A practical guide with examples in C / W. P. Petersen, P. Arbenz. — Oxford University Press, 2004. — 278 p.

2. Trobec R. Parallel Computing. Numerics, Applications, and Trends / R. Trobec, M. Vajteršic, P. Zinterhof (Eds.). — Springer, 2009. — 530 p.

3. Kshemkalyani A. D. Distributed Computing Principles, Algorithms, and Systems / A. D. Kshemkalyani, M. Singhal. — Cambridge University Press, 2008. — 754 p.

4. Zhang L. High accuracy digital image correlation powered by GPU-based parallel computing / L. Zhang, T. Wang, Z. Jiang, Q. Kemao, Y. Liu, Z. Liu, L. Tang, S. Dong // Optics and Lasers in Engineering. — 2015. — Volume 69. — P. 7 — 12.

5. Veroy B. S. Average complexity of divide-and-conquer algorithms / B. S. Veroy // Information Processing Letters. — 1988. — Volume 29, Issue 6. — P. 319 — 326.

6. Huang C.-H. A report on the performance of an implementation of Strassen's algorithm / C.-H. Huang, J. R. Johnson, R. W. Johnson // Applied Mathematics Letters. — 1991. — Volume 4, Issue 1. — P. 99 — 102.

7. Chou C.-C. Parallelizing Strassen's method for matrix multiplication on distributed-memory MIMD architectures / C.-C. Chou, Y.-F. Deng, G. Li, Y. Wang // Computers & Mathematics with Applications. — 1995. — Volume 30, Issue 2. — P. 49 — 69.

8. Coppersmith D. Matrix multiplication via arithmetic progressions / D. Coppersmith, S. Winograd // Journal of Symbolic Computation. — 1990. — Volume 9, Issue 3. — P. 251 — 280.

9. Takaoka T. Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication / T. Takaoka // Electronic Notes in Theoretical Computer Science. — 2002. — Volume 61. — P. 191 — 200.

10. Lingas A. Bit complexity of matrix products / A. Lingas // Information Processing Letters. — 1991. — Volume 38, Issue 5. — P. 237 — 242.

11. Bae S. E. A Faster Parallel Algorithm for Matrix Multiplication on a Mesh Array / S. E. Bae, T.-W. Shinn, T. Takaoka // Procedia Computer Science. — 2014. — Volume 29. — P. 2230 — 2240.

12. Barth D. Parallel matrix product algorithm in the de Bruijn network using emulation of meshes of trees / D. Barth // Parallel Computing. — 1997. — Volume 22, Issue 12. — P. 1563 — 1578.

13. Silber-Chaussumier F. Generating data transfers for distributed GPU parallel programs / F. Silber-Chaussumier, A. Muller, R. Habel // Journal of Parallel and Distributed Computing. — 2013. — Volume 73, Issue 12. — P. 1649 — 1660.

References

1. Petersen W. P., Arbenz P. Introduction to Parallel Computing: A practical guide with examples in C, Oxford University Press, 2004, 278 p.

2. Trobec R., Vajteršic M., Zinterhof P. (Eds.). Parallel Computing. Numerics, Applications, and Trends, Springer, 2009, 530 p.

3. Kshemkalyani A. D., Singhal M. Distributed Computing Principles, Algorithms, and Systems, Cambridge University Press, 2008, 754 p.

4. Zhang L., Wang T., Jiang Z., Kemao Q., Liu Y., Liu Z., Tang L., Dong S. High accuracy digital image correlation powered by GPU-based parallel computing, Optics and Lasers in Engineering, 2015, Volume 69, pp. 7 — 12.

5. Veroy B. S. Average complexity of divide-and-conquer algorithms, Information Processing Letters, 1988, Volume 29, Issue 6, pp. 319 — 326.

6. Huang C.-H., Johnson J. R., Johnson R. W. A report on the performance of an implementation of Strassen's algorithm, Applied Mathematics Letters, 1991, Volume 4, Issue 1, pp. 99 — 102.

7. Chou C.-C., Deng Y.-F., Li G., Wang Y. Parallelizing Strassen's method for matrix multiplication on distributed-memory MIMD architectures, Computers & Mathematics with Applications, 1995, Volume 30, Issue 2, pp. 49 — 69.

8. Coppersmith D., Winograd S. Matrix multiplication via arithmetic progressions, Journal of Symbolic Computation, 1990, Volume 9, Issue 3, pp. 251 — 280.

9. Takaoka T. Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication, Electronic Notes in Theoretical Computer Science, 2002, Volume 61, pp. 191 — 200.

10. Lingas A. Bit complexity of matrix products, Information Processing Letters, 1991, Volume 38, Issue 5, pp. 237 — 242.

11. Bae S. E., Shinn T.-W., Takaoka T. A Faster Parallel Algorithm for Matrix Multiplication on a Mesh Array, Procedia Computer Science, 2014, Volume 29, pp. 2230 — 2240.

12. Barth D. Parallel matrix product algorithm in the de Bruijn network using emulation of meshes of trees, Parallel Computing, 1997, Volume 22, Issue 12, pp. 1563 — 1578.

13. Silber-Chaussumier F., Muller A., Habel R. Generating data transfers for distributed GPU parallel programs, Journal of Parallel and Distributed Computing, 2013, Volume 73, Issue 12, pp. 1649 — 1660.

Рецензія/Peer review : 9.5.2015 р. Надрукована/Printed : 15.5.2015 р.
Рецензент: д.т.н., проф. Троцишин І.В.