On the Inversion of Multiple Matrices on GPU in Batched Mode[1]

***Nikolay M. Evstigneev*[2], *Oleg I. Ryabkov*[3], *Eugene A. Tsatsorin*[4]**

In the research we are considering the benchmarking of batched matrix inversion and solution of linear systems. The problem of multiple matrix inversion with the same fill sparsity is usually considered in problems of fluid mechanics with chemistry. In this case the system is stiff and an implicit method is required to solve the problem. The core of such method is the multiple matrix inversion. We benchmark different methods based on cuSPARSE and MAGMA libraries and CPU LAPACK version depending on the matrix filling. We also provide our own experimental code that implements GaussJordan elimination on GPU using register shuffle. It is shown that the fastest method is the QR matrix inversion for single precision calculations. We also show that the suggested Gauss–Jordan elimination method looks promising being about 8–10 times faster than cuSPARSE QR method. We also demonstrate the application of batch solvers in the coupled reactive flow problem.

Keywords: QR algorithm, LU Matrix Inversion, Batched Solver, Matrix solver, GPU Batched Solver.

1. Introduction

In many application, such as astronomy, chemistry and approximate preconditioning design (e.g. block Jacobi preconditioning in implicit Discontinuous Galerkin methods), one must find solutions of many small linear systems of equations. Let us consider one situation that is very common in CFD where a chemical or plasma–chemical reactions are essential and included into multicomponent system of equations, e.g. see [1]. Chemical reactions are governed by systems of ODEs, typically of small or medium size $M \sim \mathcal{O}(10)$ and the problem complexity is multiplied by the discretization of the CFD problem size $N \sim \mathcal{O}(10^6)$. These systems of ODEs are stiff and implicit methods must be applied to find numerical solutions, e.g. Rosenbrock method [2] is a very popular choice. This leads to the solution of the following linear systems:

$$\mathtt{A}_j \mathbf{x}_j = \mathbf{b}_j, j = 1, ..., N, \tag{1}$$

where $\mathtt{A}_j \in \mathbb{R}^{M \times M}$ are matrices of the numerical method for systems of chemical reactions, $\mathbf{x}_j \in \mathbb{R}^M$ are the vectors of unknowns (concentrations) and $\mathbf{b}_j \in \mathbb{R}^M$ are the right hand sides. Methods for the solution of this problem type are called batch methods. In this paper refer to the size N as **batch size** or simply **batch** and M as **matrix size**. Matrices $\mathtt{A}_j$ are, in general, nonsymmetric, nonsingular and usually sparse with filling up to 50%.

We are aimed at multiple GPU architecture to be used for the solution of (1). There are no communication between systems (1) so we can analyse performance on a single GPU device and assume linear scaling of the problem for multiple GPUs. There are some papers related to

[1]The paper is recommended for publication by the Program Committee of the International Scientific Conference "Parallel Computing Technologies (PCT) 2018".

[2]Federal Research Center "Informatics and Control", Institute for System Analysis, Russian Academy of Science, Moscow, Russian Federation

[3]Federal Research Center "Informatics and Control", Institute for System Analysis, Russian Academy of Science, Moscow, Russian Federation

[4]Lomonosov Moscow State University, Moscow, Russian Federation

the problem. In [3] authors give design and implementation of batched matrix-matrix multiplication on GPUs. It is shown that for relatively small matrices (8×8) one achieves performance of 60 GLFOPS in single precision, while for rather big matrices 32×32 a performance of 260 GFLOPS is achieved, both on k40 GPUs. Analysis of symmetric matrices of linear systems is performed in [5] during the solution of the problem (1). A comparison of MKL LAPACK, MAGMA library [6] and self-written routine. It is stated, that 80% of the practical *dgemm* peak of the machine is achieved with the self-written code, while MAGMA achieves only 75%, and finally, in terms of energy consumption MAGMA is outperformed by 1.5 times in performance-per-watt for larger matrices. However MAGMA is assumed to be a fair good alternative. Batched matrices LU decomposition is discussed in [4]. Batched mode is compared with the streamed one and it is shown that the premire is superior. A batched LU factorization for GPUs is proposed that uses a multi-level blocked right looking algorithm. It preserves the data layout but minimizes the penalty of partial pivoting. As a result 2.5 speedup is achieved, compared to the alternative CUBLAS solution on a K40c GPU. Another new paper is [7] where MAGMA is compared with CUBLAS for the solution of million linear systems. It is shown that MAGMA is an efficient library and is significantly faster than CUBLAS for the considered problems, scoring speedups between 4.3 – 16.8 in single precision and between 3.4 – 14.3 in double precision. Performance is around 250–350 GFLOPS in single precision for P100 NVIDIA GPU.

All these results give great insights into performance, but we found no good comparison of libraries that are designed to be used in batch mode. Besides, there are no comparison in terms of wall time which is what a developer looking for in the first place when trying to speed-up the problem with GPU usage. One can use **cuSOLVER** NVIDIA CUDA library [8] to perform batch solution of many small linear systems. The goal of the paper is to perform as many tests as possible, related to the problem (1), and give an insight on using different libraries for future reference and solution strategies using modern compute capability of GPUs.

2. Benchmark problem and metrics

In the paper we use four different methods to solve the problem (1). The CPU version is LAPACK routines **SGETRS** for single precision and **DGETRS** for double precision. The OpenMP is used to divide the stream and run independent solvers.

Next we use MAGMA library by calling **magma_(S/D)gesv_batched** routine that executes the linear solution of systems on a GPU in single or double precision. Magma uses full matrix storage and we refer to magma library below as *magma_float* and *magma_double*, respectively, for single and double precision call.

We also test two cuSOLVER libraries. The first one is using QR matrix decomposition and is called by **cusolver(S/D)pXcsrqrsvBatched** routine in single or double precision. Some additional preparations must be made that analyse system matrices connectivity graph and transfer reordering permutations to increase the efficiency on the device, for more information we refer a reader to the manual [8]. The second one is called refactored solver that uses LU matrix decomposition and is called by **cusolverRfBatchSolve** routine. Note that some additional preparations must be made on a host part of the program. Besides, the interface of the call is assumed to be only in double precision (with respect to CUDA Toolkit Version 8.0). Both routines use sparse CRS matrix storage format. We refer to these libraries as *QR_float*, *QR_double* and *RF_double*, respectively.

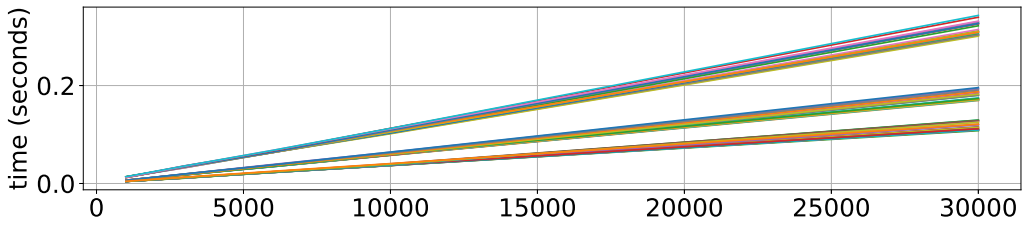

Figure 1. Time for inversion of all matrices on CPU using 1,2 and 4 threads as function of batch size. Different colors corresponds to different matrices.

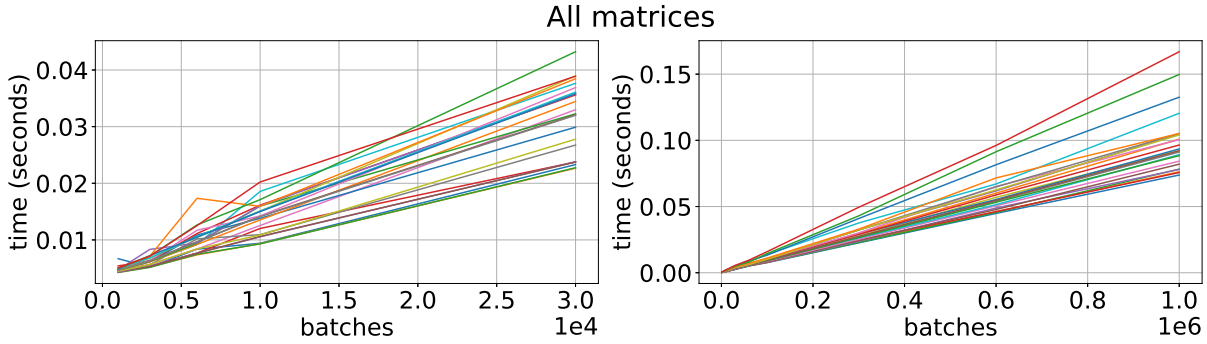

Figure 2. Time for inversion of all matrices on GPU using magma_float (left) and QR_float as function of batch size on Device 0. Different colors corresponds to different matrices.

Our test is performed in the following manner. We generate set of matrices with sizes $M = \{4, 5, 8, 9, 10, 11, 14, 16\}$ with random sparsity patterns having filling 10%, 25% and 50% of all matrix elements, except diagonal elements. We also fill diagonal elements in such a way, that matrices are invertable. It is checked on the stage of matrix generation. We assume that pivoting can be used to these systems to increase the stability of system solution to perturbations. We use the following set of batch sizes: $N = \{1, 3, 6, 10, 30, 60, 100, 300, 600, 1000\} \cdot 10^3$. We also check the performance of two different GPUs with single and double precision calculations, where possible. The *test set* is generated as a tensor product of all possible configurations. For each test in the *test set* we make 10 runs of the code with different generated matrices that share the same sparsity pattern and execution time is averaged. Performance is measured in FLOPS obtained from **nvprof** utility. All tests are generated, executed and logged by an automated python script.

We used CPU device INTEL XEON E5-2609 2.4GHz, 4 cores and two GPU devices. The first one designated **Device 0** is a GTX TITAN X NVIDIA card with 12GB RAM, 11 TFLOPS peak single precision, 1/32 multiplier for double precision and 5.2 compute capability. The second **Device 1** is a GTX TITAN Black NVIDIA card with 6GB RAM, 5.1 TFLOPS peak single precision, 1/3 multiplier for double presision and 3.5 compute capability.

3. Gauss–Jordan elemiation using register shuffle

The code is based on simple Gauss-Jordan elemiation algorithm. A distinctive feature of the implementations is the aplication of register shuffle that is supported from compute capability 3.0 and above. It allows us to share data between threads that are part of the same warp. It insures a speedup of about 3 times to the shared memory access speed but limits current implementation in using only single precision arithmetics (double precision requires spliting into

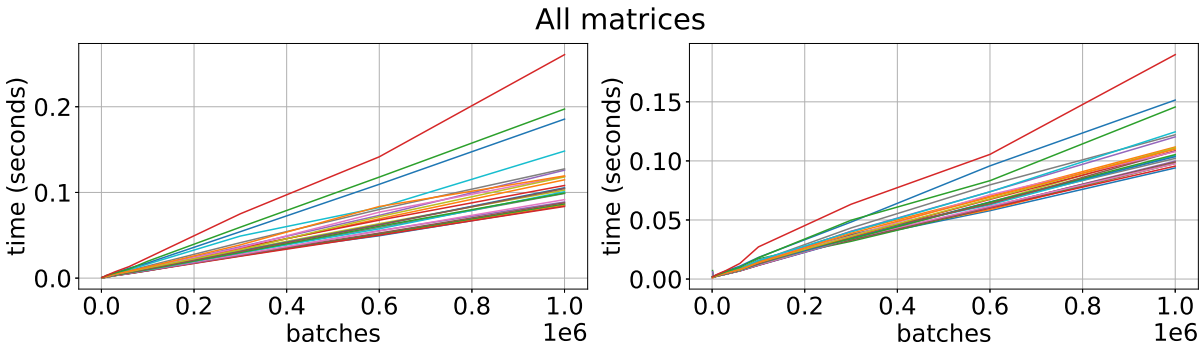

Figure 3. Time for inversion of all matrices on GPU using QR_double (left) and RF as function of batch size on Device 0. Different colors corresponds to different matrices.

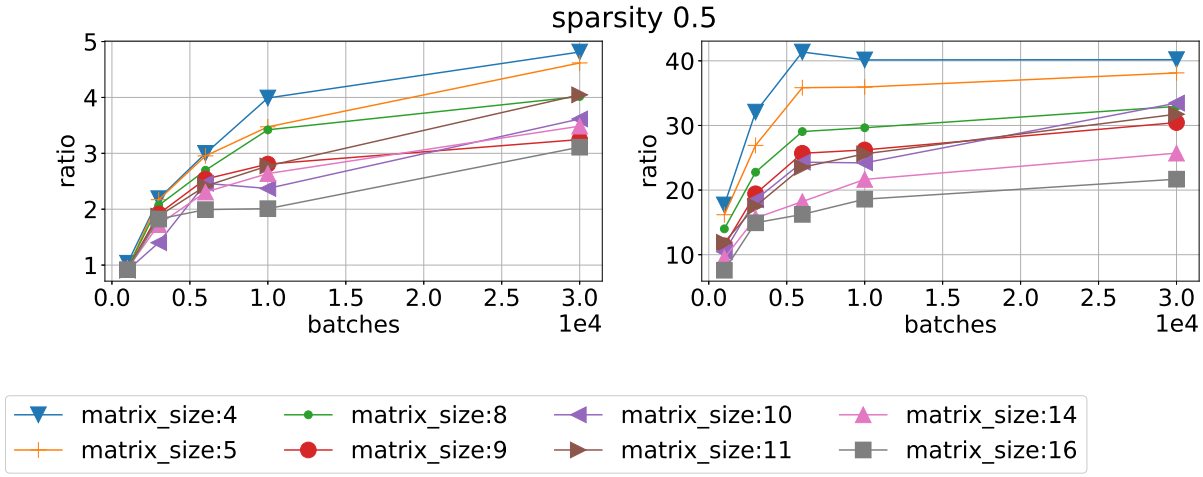

Figure 4. Ratio of mean time execution vs batch size using sparsity 0.5 for different matrix sizes. CPU/magma_float left and CPU/QR_float right on Device 0.

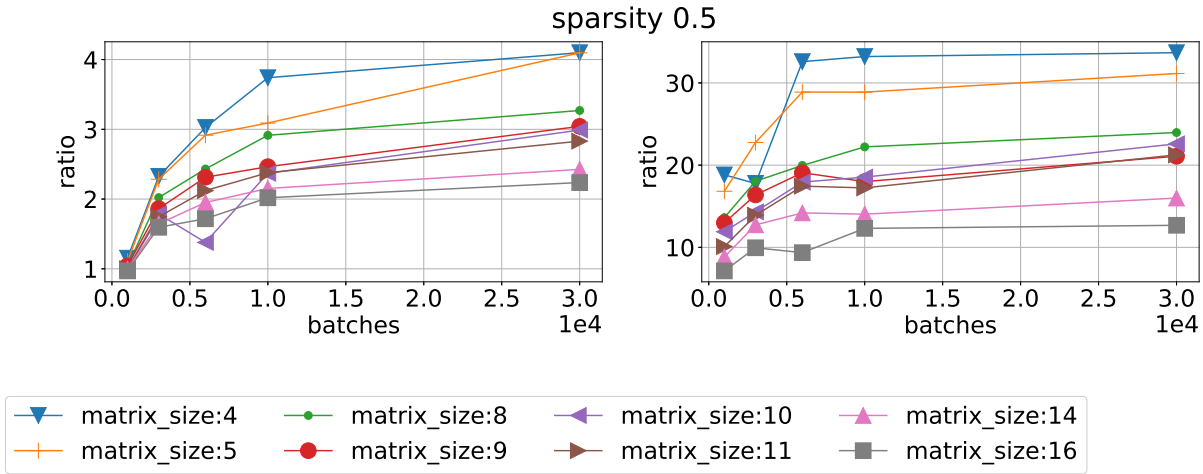

Figure 5. Ratio of mean time execution vs batch size using sparsity 0.5 for different matrix sizes. CPU/magma_float left and CPU/QR_float right on Device 1.

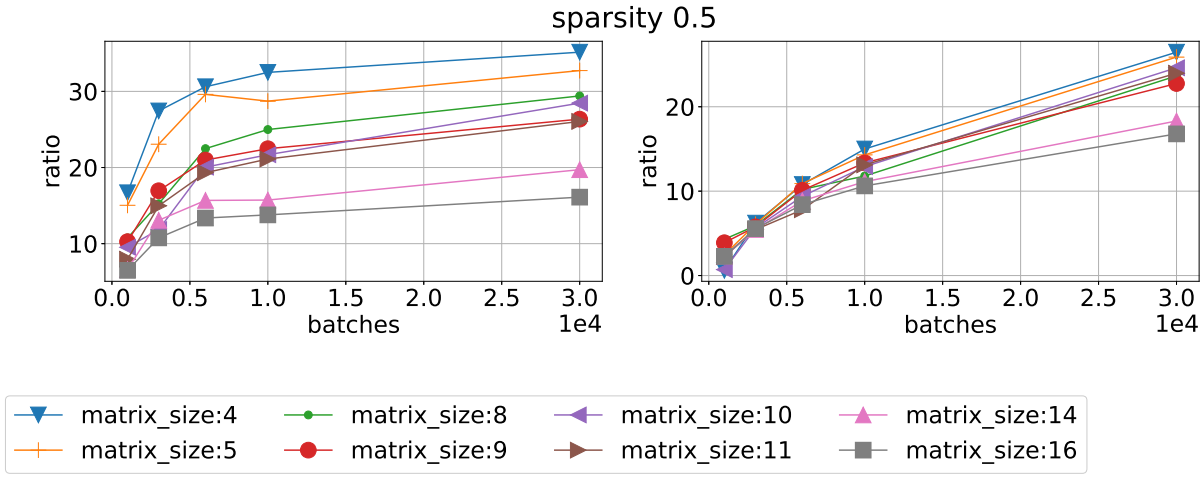

Figure 6. Ratio of mean time execution vs batch size using sparsity 0.5 for different matrix sizes. CPU/QR_double left and CPU/RF right on Device 0.

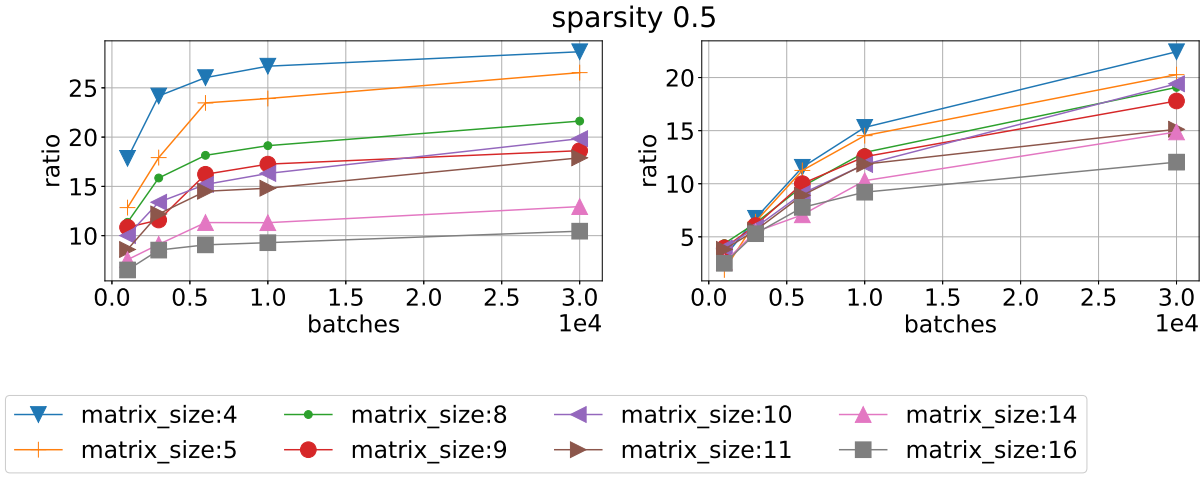

Figure 7. Ratio of mean time execution vs batch size using sparsity 0.5 for different matrix sizes. CPU/QR_double left and CPU/RF right on Device 1.

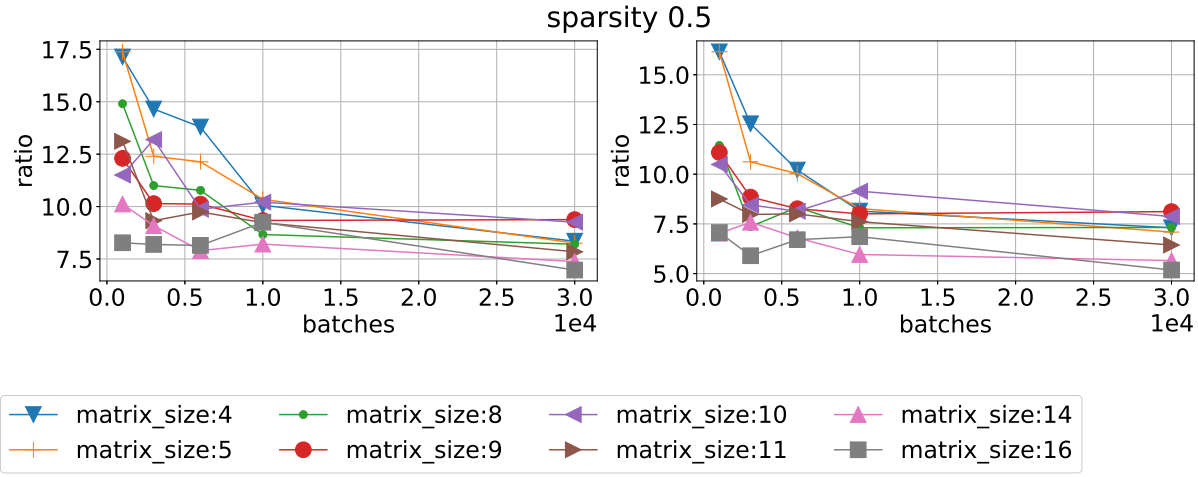

Figure 8. Ratio of mean time execution vs batch size using sparsity 0.5 for different matrix sizes. magma_float/QR_float left and magma_double/QR_double right on Device 0.

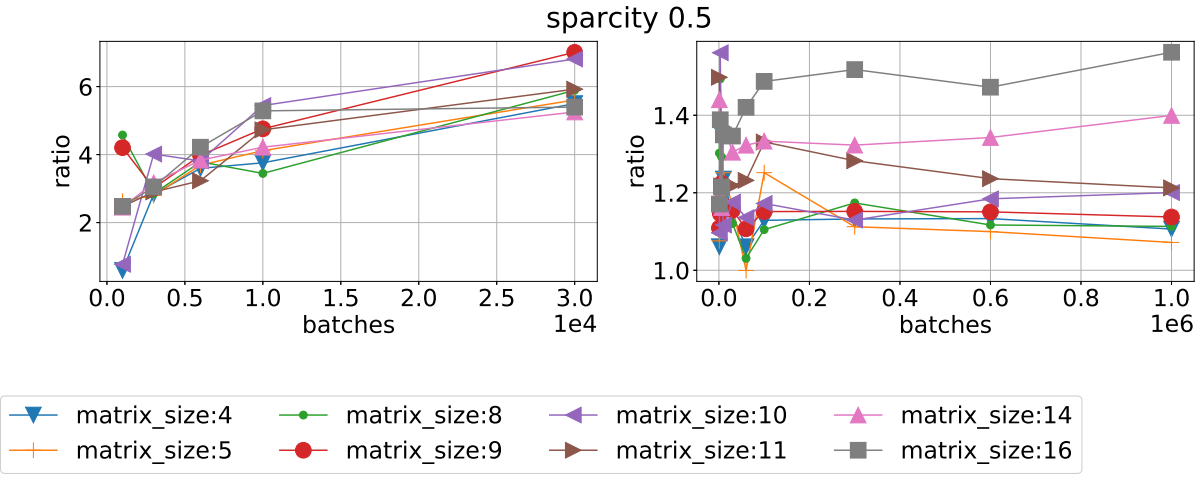

Figure 9. Ratio of mean time execution vs batch size using sparsity 0.5 for different matrix sizes. magma_float/RF left and QR_double/QR_float right on Device 0.

two 32b registers) and matrix size $M \leq W$, where W is a warp size. For more information see [9]. This code is now being tested and is in alpha version, we designate this code as **shuffleGJ**. It is also benchmarked against best results of selected libraries in the end and performance Gflops are provided.

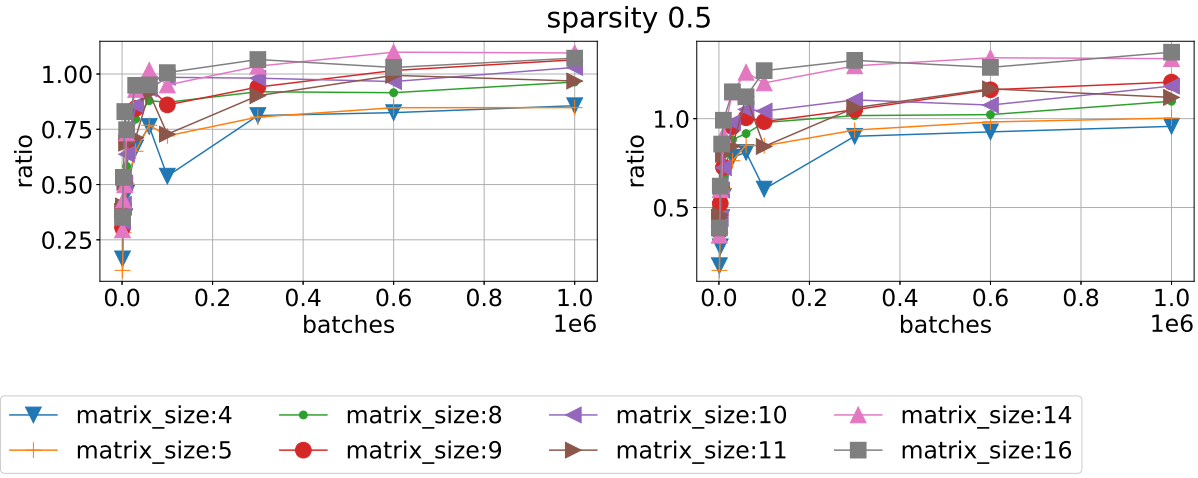

Figure 10. Ratio of mean time execution vs batch size using sparsity 0.5 for different matrix sizes. QR_float/RF left and QR_double/RF right on Device 1.

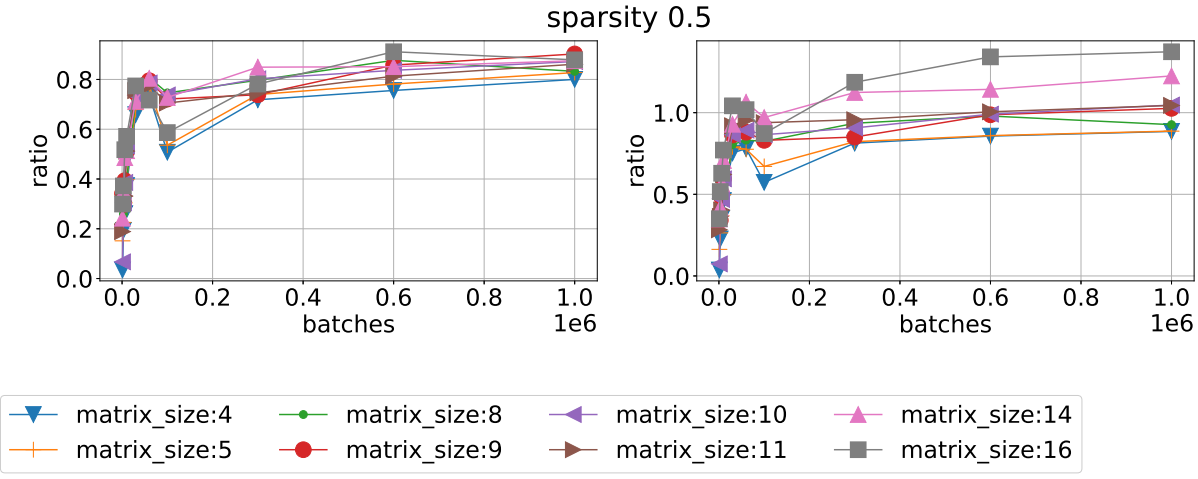

Figure 11. Ratio of mean time execution vs batch size using sparsity 0.5 for different matrix sizes. QR_float/RF left and QR_double/RF right on Device 0.

4. Results

4.1. Libraries analysis

All results are brought into figures that represent sections and projections of multidimensional data from the *test set*. All figures are self–explanationary, but we comment on some of most essential results. In figure 1 we can see the time needed to solve systems of equations for all matrices as function of batch size (up to 30000 matrices) using 1,2 and 4 threads with single precision. We can see linear scaling with batch size since all matrices are treated as dense and have no dependence on sparsity. Analogous results by different libraries on Device 0 are provided in figures 2, 3 showing significant reduction of time. Interesting to note that batched RF method having no interface with single precision is almost as efficient as QR_float. Result are a little bit different on Device 1. We checked the occupancy of GPU RAM. QR_float and RF use about the same amount of RAM, for example it requires 1845 MB using QR_float and 3590 MB using RF for $N = 1 \cdot 10^6, M = 11$ and sparsity 0.5. MAGMA requires way more memory due to some internal memory allocation instructions, for example magma_float takes about 831 MB for $N = 3 \cdot 10^4, M = 11$ and about 5264 MB for $N = 2.4 \cdot 10^5$. This memory is dynamically allocated during the solution process and can cause exception of insufficient memory despite that memory for all matrices storage is sufficient, so we limit MAGMA batch size up to 30000.

Speed ups against CPU 4-thread code for different libraries and different devices are shown in figures 4, 5, 6, 7. We can see that MAGMA speed up varies from 5 to 3 times for Device 0 and $4 - 2$ times for Device 1. QR_float achieves speed up 40 times for small matrices and 20 for big matrices on Device 0 and $35 - 15$ times for Device 1. RF is about the same results, lower by approximately 10% and QR_double lower by another 10%. We can see that RF library has a narrower spread between matrix sizes.

We than benchmark one library against another in terms of execution time. Results for MAGMA library are presented in figures 8 and 9 (left). One can see that other libraries are faster, so taking memory demands into account we scratch out MAGMA from comparison. We also compare QR_float and QR_double with RF on different devices. We can see in figure 10 that RF version is more efficient on Device 1 compared with both QR_double and QR_float. However, QR_float is more efficient on Device 0, see figure 11.

Further investigation is conducted in term of speed dependence from sparsity and matrix size for all batches. For this test we check only Device 0 because we found that the difference in scaling on these axis is negligible. One see the scaling of QR_float on sparsity 0.1 in figure 12 and sparsity 0.5 in figure 13. The factor of speed loss is about 2 3 times for the matrix size increase from 4 to 16 when sparsity is 0.5 which implies better performance on bigger matrices. This effect is even stronger when sparsity is 0.1 and also for RF library, see figures14 and 15. Another dependence on sparsity is given in terms of matrix size and provided for QR_float in figures 18, 19, and for RF in figures 16, 17.

Analysis of performance in terms of floating point operations is provided in the table 1. One can see that MAGMA has very small performance results in the batch mode, except using magma_float on Device 0 with 267.78 GLFOPS. This is, probably, due to the usage of new device features of compute capability 5.2. However, timing for this library mode remain almost the same. Notice, that when MAGMA is executed as magma_float, some calculations are performed in double precision, and visa versa. Best flops performance (41.17 GLFOPS) is achieved on QR_float on matrix size $M = 16$. We also checked asymptotics by considering $M = 128$. We can

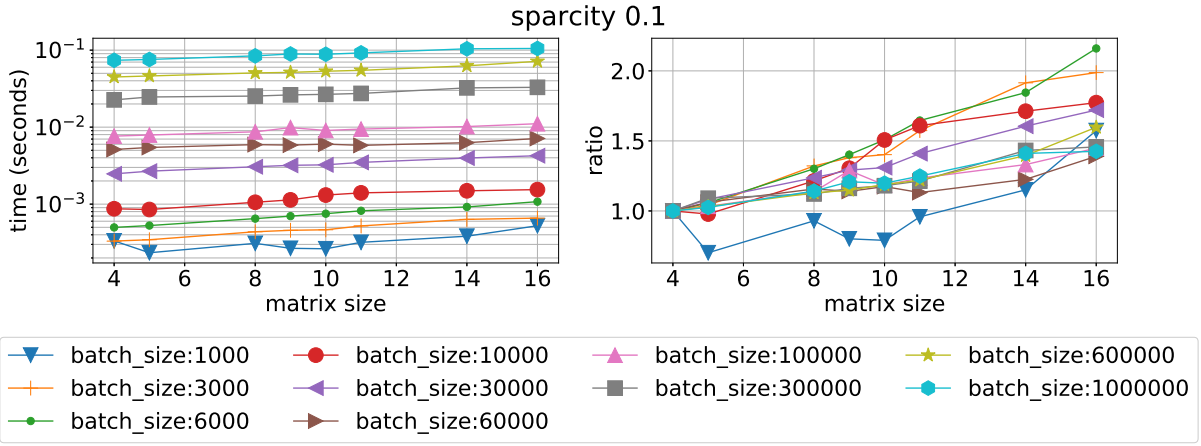

Figure 12. Mean time execution (left) and ratio of time execution to the smallest matrix vs matrix size for different batch sizes and sparsity 0.1 using QR_float on Device 0.

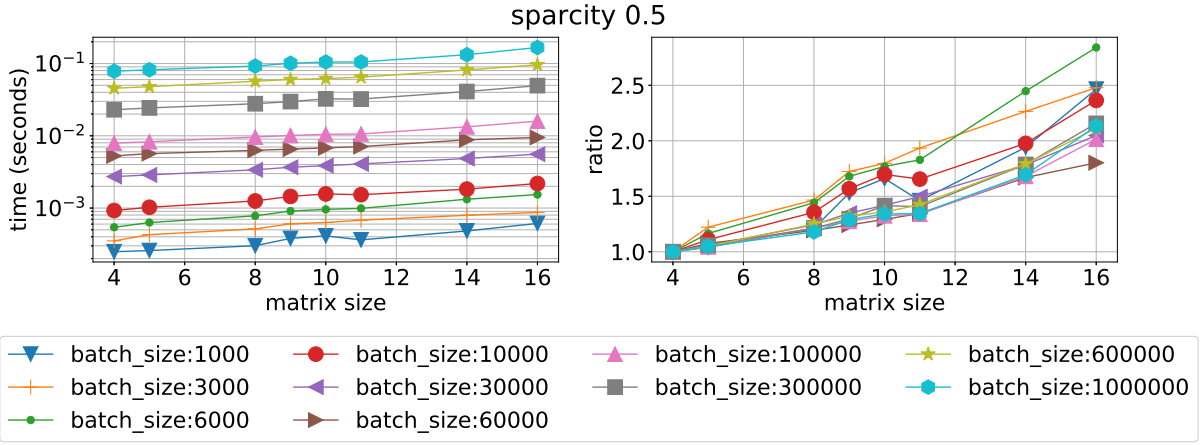

Figure 13. Mean time execution (left) and ratio of time execution to the smallest matrix vs matrix size for different batch sizes and sparsity 0.5 using QR_float on Device 0.

see, that QR_float achieves maximum performance of 39.12 GFLOPS, so we can assume that the metrics is correct for $M = 16$. RF method also uses some single precision calculations but bulk part of all calculations is done in double precision with maximum of 20.35 GFLOPS.

4.2. Shuffle Gauss–Jordan analysis

We test shuffleGJ method for $M = \{11, 16\}$ with sparsity 0.5. Ratio of time execution is shown in figures 20 for Device 0 and in figure 21 for Device 1. One can clearly see that the suggested method outperforms libraries on about 20 times for $M = 11$ and about 8–10 times for $M = 16$. This twofold decrease of performance is related to the algorithm requirements for matrix size in the shuffleGJ method. Still this gives us about 400 times acceleration compared to 4 threaded CPU version. We also calculated flops for this method that is provided in table 1. One can see that we managed to achieve about 11.3% performance compared to the maximum theoretical performance of the device.

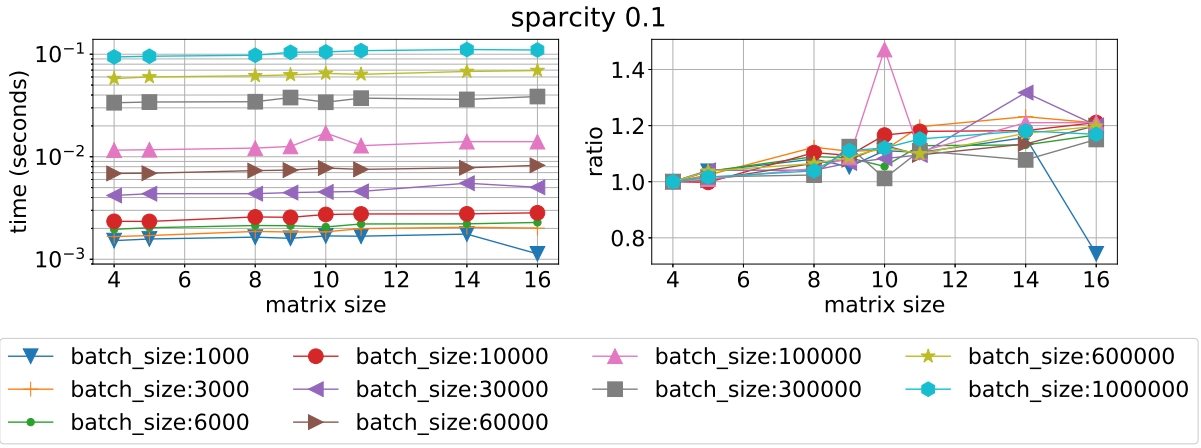

Figure 14. Mean time execution (left) and ratio of time execution to the smallest matrix vs matrix size for different batch sizes and sparsity 0.1 using RF on Device 0.

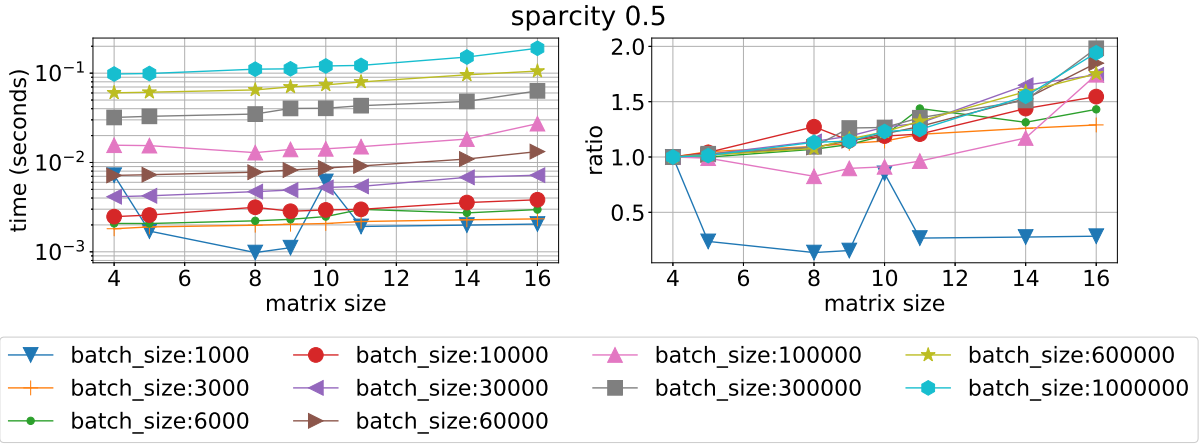

Figure 15. Mean time execution (left) and ratio of time execution to the smallest matrix vs matrix size for different batch sizes and sparsity 0.5 using RF on Device 0.

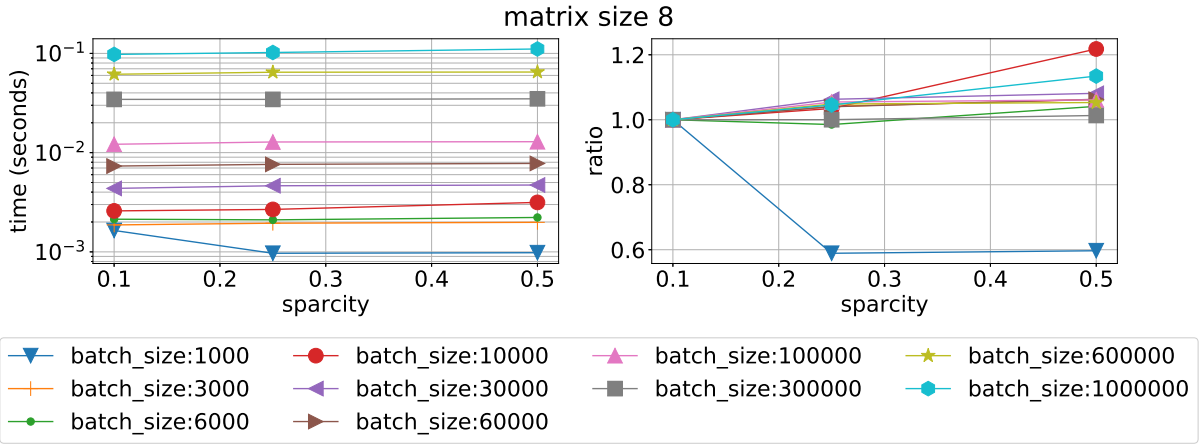

Figure 16. Mean time execution (left) and ratio of time execution to the smallest sparsity vs sparsity fill for different batch sizes and matrix size 8 using RF on Device 0.

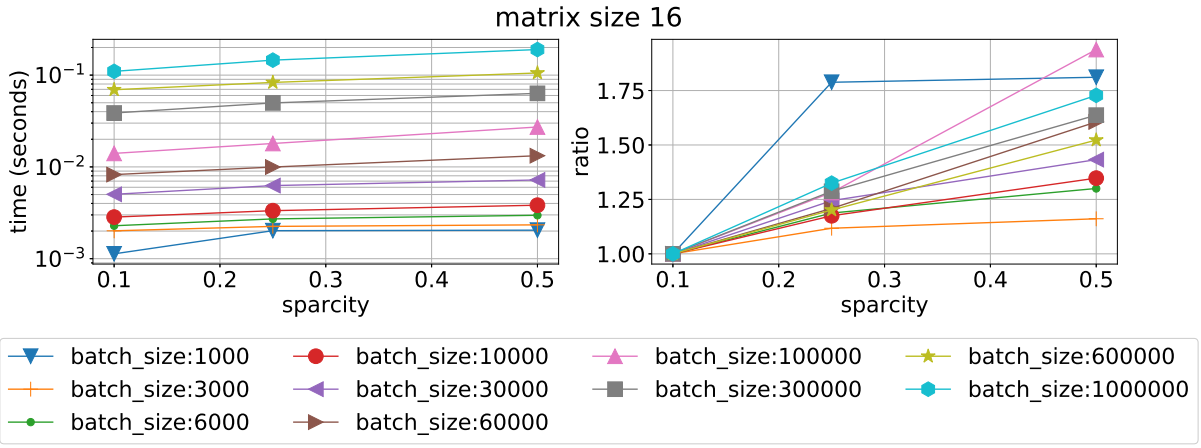

Figure 17. Mean time execution (left) and ratio of time execution to the smallest sparsity vs sparsity fill for different batch sizes and matrix size 16 using RF on Device 0.

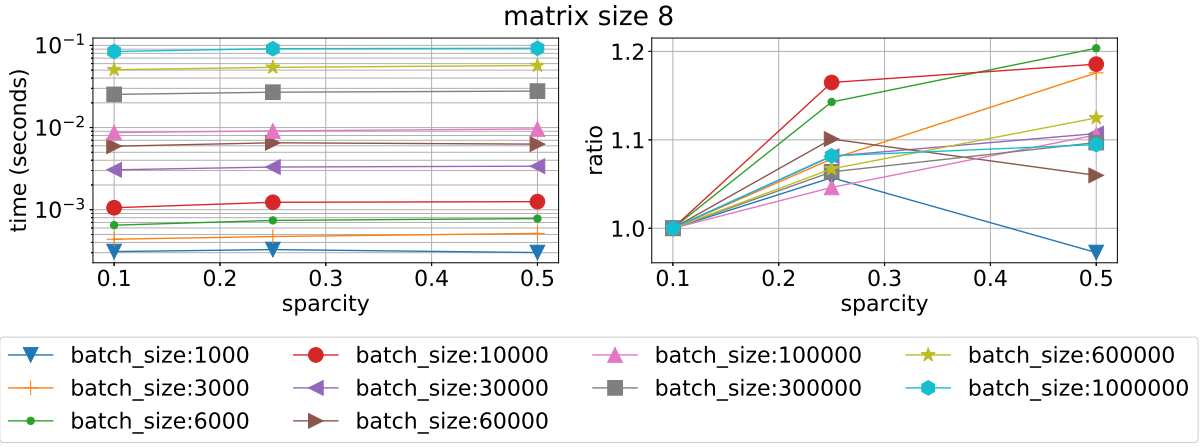

Figure 18. Mean time execution (left) and ratio of time execution to the smallest sparsity vs sparsity fill for different batch sizes and matrix size 8 using QR_float on Device 0.

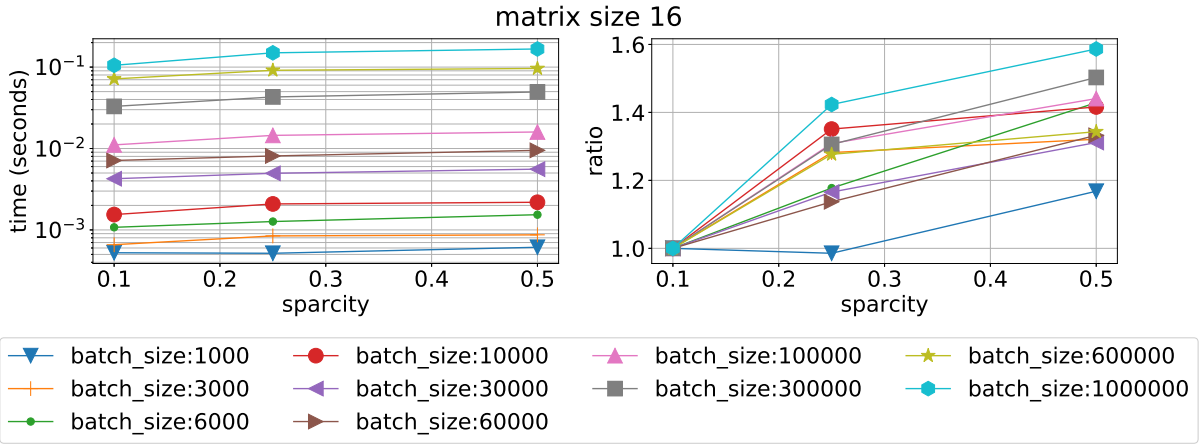

Figure 19. Mean time execution (left) and ratio of time execution to the smallest sparsity vs sparsity fill for different batch sizes and matrix size 8 using QR_float on Device 0.

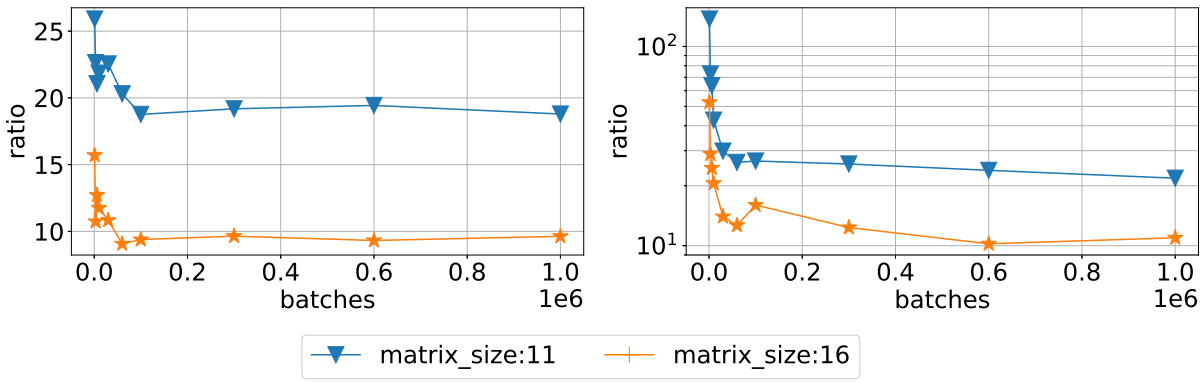

Figure 20. Ratio of time execution of QR_float/shuffleGJ (left) and RF/shuffleGJ (right) as function of batch size for sparsity 0.5 on Device 0.

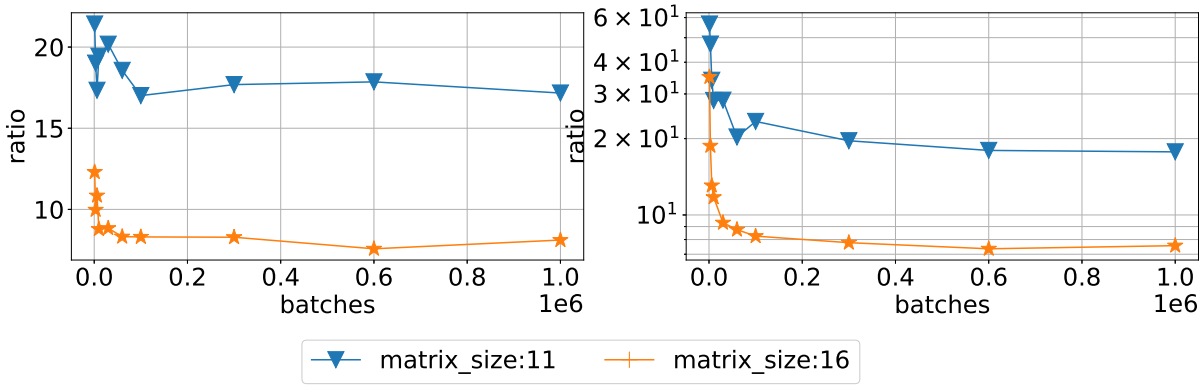

Figure 21. Ratio of time execution of QR_float/shuffleGJ (left) and RF/shuffleGJ (right) as function of batch size for sparsity 0.5 on Device 1.

Table 1. Performance of different batch solver implementations on selected problems with sparsity 0.5 and one million batch size.

solver	**device**	**matrix_size**	**Gflops(float)**	**Gflops(double)**
QR_float	0	4	4.85	0
QR_float	0	11	25.34	0
QR_float	0	16	41.17	0
QR_float	1	4	3.73	0
QR_float	1	11	15.16	0
QR_float	1	16	23.69	0
QR_double	0	4	0.18	7.97
QR_double	0	11	0.24	24.73
QR_double	0	16	0.119	28.04
QR_double	1	4	0.071	6.21
QR_double	1	11	0.105	15.51
QR_double	1	16	0.084	19.58
RF_double	0	4	0.3317	4.2
RF_double	0	11	0.497	14.5
RF_double	0	16	0.33	20.35
RF_double	1	4	0.136	3.45
RF_double	1	11	0.244	10.36
RF_double	1	16	0.234	14.5
magma_float	0	11	*267.78*	*0.14*
magma_float	1	11	9.22	*0.148*
magma_double	0	11	*0.139*	9.09
magma_double	1	11	*0.102*	7.31
QR_float	0	128	38.79	0
QR_float	1	128	39.12	0
shuffleGJ	0	11	955.49	0
shuffleGJ	0	16	1251.88	0
shuffleGJ	1	11	544.25	0
shuffleGJ	1	16	595.38	0

5. Application example

We are considering a standard chemical reaction flow benchmark problem, called ZND ([10], [11], [12]). The problem is formulated for compressible perfect gas equations with chemical reaction. Detonation wave is propotaged with constant velocity D. The wave has the following structure: gas shock wave is propotaged, followed by reaction domain. Detonation reaction velocity is behind the shock wave. Governing equations are given bellow:

$$\begin{cases} \rho_t + \nabla \cdot (\rho \mathbf{u}) = 0, \\ (\rho \mathbf{u})_t + \nabla \cdot (\mathbf{u} \otimes (\rho \mathbf{u})) + \nabla p = \mathbf{0}, \\ \rho E_t + \nabla \cdot (\mathbf{u}(\rho E + p)) = 0, \\ (\rho Y_j)_t + \nabla \cdot (\rho Y_j \mathbf{u}) = \dot{\omega}_j. \end{cases} \tag{2}$$
$$\rho E = \tfrac{1}{2}\rho \mathbf{u}^2 + \rho e, j = \overline{1, M}, \sum_{j=1}^{M} Y_j = 1.$$

The chemical source term is given by:

$$\dot{\omega_1} = -\dot{\omega_2} = \begin{cases} -\rho Y_1 A \exp\left(\frac{-E_{act}}{\mathcal{R}T}\right), T \geq T_{ign}, \\ 0, T < T_{ign}. \end{cases} \tag{3}$$

Above equations are coupled by the equation of state:

$$\begin{gathered} p = \rho \widetilde{R} T, \\ h = C_p T + h_0, h_0 = \textstyle\sum_{j=1}^{M} h_{0,j} Y_j, \\ C_p = \frac{\gamma}{\gamma - 1} \widetilde{R}. \end{gathered} \tag{4}$$

Here ρ is density of gas mixture, ρY_j is the mass fraction of gas species j, $M = 2$ is the number of species, A is the Arrhenius frequency factor, $\mathbf{u}$ is the velocity vector, p is the pressure, E is the total energy density, h is the specific enthalpy, e is the specific internal energy, $\dot{\omega}_j$ is the reaction rate of species j, T is temperature, $\mathcal{R} = 8.31451$ is the universal gas constant, $\widetilde{R} = 1$ is the specific gas constant, γ is the specific heat ratio of the gas mixture, $h_{0,j}$ is the reference enthalpy of formation for the species j. In all calculations we set $T = 1$ and $T_{ign} = 1.01$. The fluid dynamics is solved using discontinuous Galerkin method [13] and the chemical part is solved using Crank-Nicolson method.

We define two domains - 1D segment and 2D plane. For the 1D segment we set boundary conditions as supersonic outflow conditions on the right and subsonic outflow conditions on the left. For the 2D plane we add two boundary conditions of sleep walls on top and bottom.

Initial conditions are defined as:

- $x \geq 0$: $(\rho, \rho u_x, \rho u_y, \rho E, \rho Y_1, \rho Y_2)^T = (\rho Y_1^*, -\rho Y_1^* D, 0, \rho E, \rho Y_1^*, 0)^T$.
- $x < 0$:

$$\frac{dY_1(x)}{dx} = -\frac{\dot{\omega}(Y_1(x))}{\rho(Y_1(x)) Y_1^* D}. \tag{5}$$

Here Y_1 is burned species and $Y_2 = 1 - Y_1$ is unburned species, $Y_1^* = 1$ is a constant value. All other values for initial conditions are calculated in accordance with [14] as functions of $Y_1(x), \Delta h_{0,2}, D$. The Arrhenius frequency factor A is obtained from the given half–reaction length of a detonation wave as:

$$L_{1/2} = -\rho_2 D \int_{1/2}^{1} \frac{1}{\dot{\omega}(z)} dz. \tag{6}$$

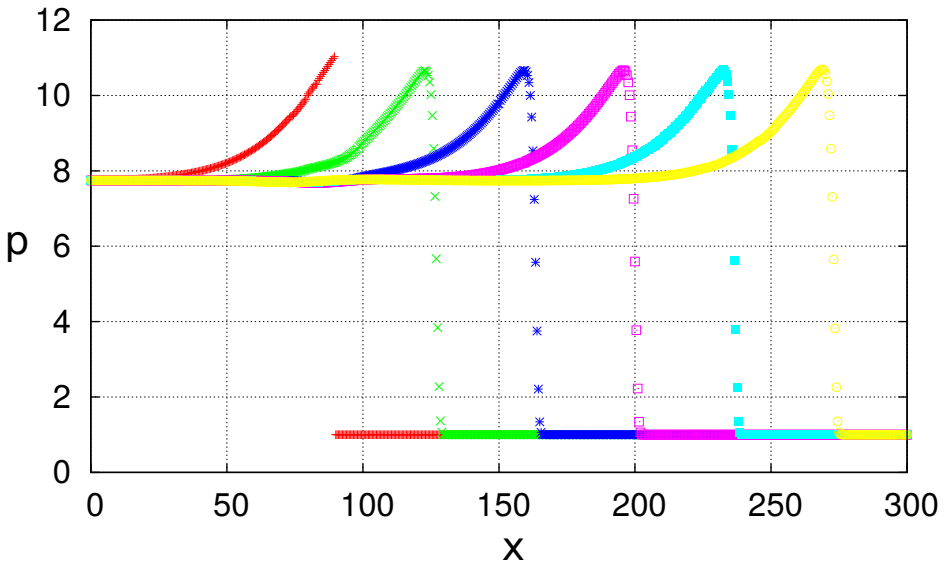

Figure 22. Pressure distributions for different time steps (10 seconds per step). Propagation velocity $D_{num} \sim 3.664275$ m/s.

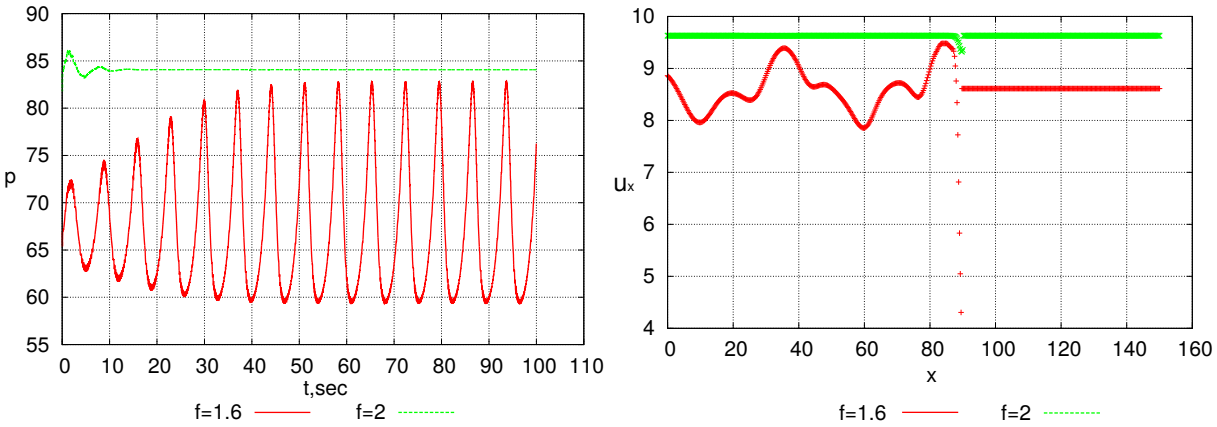

Figure 23. Temporal(left) and spatial(right) evolution of the detonation wave as function of initial detonation velocity parameter f.

The initial value problem (5) and value of A cannot be solved analytically, but the solution to these problems can be computed numerically for any given accuracy.

Three tests are performed - 1D comparison of computed velocity of detonation wave D, stability and instability of detonation wave for different initial D and 2D unstable detonation wave propagation. First results of the detonation wave propagation are presented in figure 22 for parameters $\gamma = 1.4$, $L_{1/2} = 12.5448$. One can observe that the obtained velocity $D_{num} \sim 3.664275$ m/s is close to the reference propagation velocity $D = 3.66931$ m/s. The other test verifies the stability of the detonation wave under provided value of D. We use parameter f to define the propagation velocity $D = \sqrt{f} D_{CJ}$ from the analytical speed D_{CJ} given by the Chapman – Jouguet theory [14]. The results demonstrate that for $L_{1/2} = 1$, $\gamma = 1.4$ and different values of parameter f one obtains different stability properties for the detonation wave. The results in figure 23 fully agree with reference results from [15], page 40.

The third test demonstrates spatial heterogeneity of the detonation wave (see figure 24), calculated for the same parameter values as the previous test. Again, we can check these results with reference data from [15], page 157.

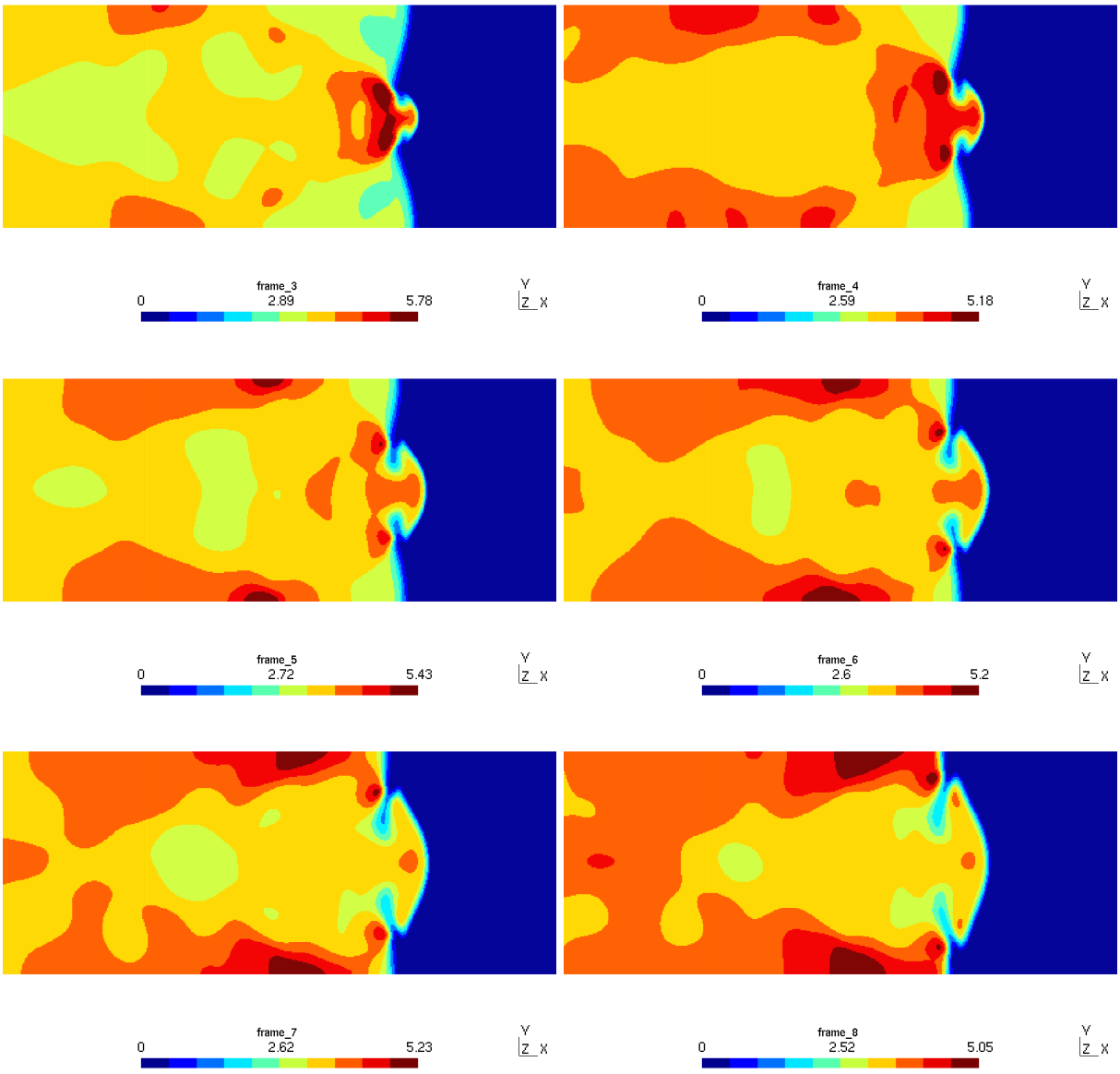

Figure 24. Spatial evolution of ρY_1 for different time steps in the reference frame moving with velocity D.

Table 2. Acceleration of the 2D ZND problem for reactive flow (chem) and pure gas dynamics (no chem).

solver	time per step(ms)	acceleration
CPU, 1 thread, chem	8708.158	1.00
GPU, Device 0, chem	46.339	187.92
GPU, Device 1,chem	65.381	133.19
CPU, 1 thread, no chem	5922.279	1.00
GPU, Device 0, no chem	27.935	212.00
GPU, Device 1, no chem	43.538	136.03

To check the performance of the chemical solver, we use the **RF** batch solver for the ODE batch Crank-Nicolson method. The performance results are presented in the table 2. We can observe that the overall performance is satisfactory, though the chemical solver sightly degradates the performance.

6. Conclusion

First, we wish to note that we don't recommend using MAGMA batch library call for the solution of the problem (1) at least for library version 2.2. It is clear from all metrics that we collected, especially from table 1.

For the available libraries we can conclude the following. It is beneficial to use RF method if your code uses double precision arithmetics and QR_float if you are using single precision. Both these methods perform graph connectivity analysis on CPU of provided matrices before calling batched routine for the solution of the problem on GPU. So these methods would require additional CPU work if your matrices connectivity is changing from one execution to anther. Note that the GFLOPS achieved by both of these methods is about 0.3-0.5% of the peak GFLOPS performance of GPUs.

If your GPUs support compute capability 3.0 and higher we can recommend using shuffle Gauss–Jordan method as an alternative to libraries for batch solution of linear systems. Achieved GLOPS and accelerations look promising. Debugging of this code and using it to solve plasma–chemistry on GPU is our next goal.

In the last section we demonstrate a successful application of the batched solver in the coupled gas dynamics reacting flow ZND problem.

All benchmark data, benchmark source codes and python scripts to generate test matrices and figures are availablable at github [16].

This work is supported by RFBR grant no. 17-07-00116 and by subprogram 0063–2016–0018 of the program III.3 ONIT RAS.

References

1. Asaithambi, R., Muppidi, S., Mahesh, K.: A numerical method for DNS of turbulent reacting flows using complex chemistry. 42nd AIAA Fluid Dynamics Conference and Exhibit, AIAA 2012-3252, (2012), DOI: 10.2514/6.2012-3252.

2. Rosenbrock, H.H.: Some general implicit processes for the numerical solution of differential equations. The Computer Journal, 5(4): 329–330, (1963), DOI: 10.1093/comjnl/5.4.329.

3. Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.: Performance, Design, and Autotuning of Batched GEMM for GPUs. Chapter in book: High Performance Computing, (2016), 21–38. DOI: 10.1007/978-3-319-41321-1_2.

4. Dong, T., Haidar, A., Luszczek, P., Harris, J.A., Tomov, S., Dongarra, J.: LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), (2014), DOI: 10.1109/HPCC.2014.30.

5. Dong, T., Haidar, A.,Tomov, S., Dongarra, J.: A Fast Batched Cholesky Factorization on a GPU. Proc of 43-rd International Conference on Parallel Processing, (2014).

6. MAGMA Library documentation `http://icl.cs.utk.edu/magma/`

7. Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.: Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures. Procedia Computer Science, vol. 108, pp. 606, (2017), DOI: 10.1016/j.procs.2017.05.250.

8. cuSOLVER CUDA Toolkit Documentation `http://docs.nvidia.com/cuda/cusolver/index.html`

9. Demouth, J.: Shuffle: Tips and Tricks. GPU Technology conference, (2013). `http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf`

10. Y. B. Zeldovich. On the theory of the propagation of detonation in gaseous systems.// Zh. Eksp. Teor. Fiz., 10:542568, 1940. Engl. transl.: NACA TM 1261, 1960.

11. J. von Neumann. Theory on detonation waves.// In A. J. Taub, editor, John von Neumann, Collected Works, Vol. 6. Macmillan, New York, 1942.

12. W. Doring. On detonation processes in gases.// Ann. Phys., 43:421436, 1943.

13. Evstigneev, N.M. and Ryabkov O.I. On The Development of High-Order Discontinuous Galerkin Method on 3D Unstructured Grid for Hyperbolic and Parabolic Problems Using Graphics Processors.// proc. PCT-2017, pp. 63-77, 2017.

14. W. Fichett, W.C., Davis. Detonation, Theory and Experiment. - New York, 2000.

15. Geßner, T. Dynamic Mesh Adaption for Supersonic Combustion Waves modeled with Detailed Reaction Mechanisms. - Doctoral Dissertation, Universitat Freiburg im Breisgau, 2001.

16. `https://github.com/oryabkov/cuda_batch_linsolvers_test.git`