A Performance Comparison of Computing LU Decomposition of Matrices on the CPU and the GPU

Dušan B. Gajić[1], Radomir S. Stanković[1], Miloš Radmanović[1]

Abstract – **This paper presents a comparison of time efficiency of different implementations of LU decomposition of matrices, performed on either central processing units (CPUs) or graphics processing units (GPUs). For processing on the CPU, we use the Eigen C++ linear algebra template library, Intel Math Kernel Library (MKL), and MATLAB (MATrix LABoratory). For performing LU decomposition on the GPU, we employ MATLAB's Parallel Computing Toolbox and Nvidia CUDA (Compute Unified Device Architecture) augmented with CULA (CUDA Linear Algebra) programming library. Processing times are compared using randomly-generated single-precision floating point matrices of size up to 16384×16384. The experiments show that the CUDA/CULA GPU implementation offers best performance for matrices of size up to 8192×8192. This implementation is on average 2.03 times faster than for the second-best performing implementation (MATLAB's Parallel Computing Toolbox on the GPU) and 11.82 times faster than the worst-performing implementation (Eigen on the CPU). For problem instances which cannot be stored in the global GPU memory (matrices larger than 8192×8192 on the used GPU), the LU decomposition is performed only on the multicore CPU, where Intel MKL proved to be 1.38 times faster than MATLAB and 2.72 times faster than Eigen.**

Keywords – **Performance comparison, LU decomposition, parallel computing, general purpose computations on graphics processing units, GPGPU.**

I. Introduction

The LU decomposition or LU factorization decomposes a square matrix into a product of a lower triangular unit matrix (unit matrix has all entries on the main diagonal equal to 1) and an upper triangular matrix [7, 15]. This method, introduced by Turing in 1948 [18], is widely used for problems such as solving systems of linear equations, inverting matrices, and computing matrix determinants, which are critical parts of many problems in science and engineering [7, 15, 16]. Therefore, efficient computation of LU decomposition is of considerable importance in scientific computing to the degree that this task is one of the standard benchmarks used to measure the performance of top supercomputers in the world [2].

Currently, two main computing platforms are available for performing numerical algorithms, such as LU decomposition that is discussed in this paper. These are the central processing unit (CPU) and the graphics processing unit (GPU). Current CPUs are still based on the single instruction, single data (SISD) von Neumann architecture, although they are now multicore [8]. GPUs have a single instruction, multiple data (SIMD) manycore architecture, which became programmable for non-graphics general-purpose algorithms only in the last ten years [1, 10]. Distinctions in computing architectures lead to different performance when implementing the same algorithms. Further, specific characteristics of architectures motivated development of various programming frameworks tailored to take advantage of some of these characteristics. Therefore, it is compelling to perform a comparison of different implementations of LU decomposition computed on CPUs and GPUs.

This paper presents a performance comparison of computing the LU decomposition using three different programming frameworks for the CPU and two for the GPU. The aim of the paper is to identify the computing platform and programming framework which produces the fastest LU decomposition of single-precision real matrices.

The paper is organized as follows. First, we briefly review the method of LU decomposition in Section 2. Section 3 provides basic information about the Eigen, Intel MKL, and MATLAB programming environments for the CPU, and MATLAB's Parallel Computing Toolbox and CUDA/CULA programming frameworks for the GPU. Section 4 offers some implementations details. The experimental settings and results are presented in Section 5. Section 6 summarizes the results of the research reported in this paper.

II. LU Decomposition

LU decomposition is a method for factorizing a square matrix **A** into a product of two matrices - a lower triangular unit matrix **L** and an upper triangular matrix **U**. The method of LU decomposition can be extended to non-square matrices by adding the requirement that **U** must be a row echelon matrix [15]. In the general case, partial row reordering (pivoting) of **A** before decomposition is performed when necessary to ensure existence and numerical stability of LU factorization [7, 16].

Using the LU decomposition, a square matrix **A** of size $N \times N$ is factorized as

[1]Dušan B. Gajić, Radomir S. Stanković, and Miloš Radmanović are with the University of Niš, Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Niš, Serbia, E-mails: dusan.b.gajic@gmail.com, radomir.stankovic@gmail.com, milos.radmanovic@elfak.ni.ac.rs.

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} =$$

$$=\mathbf{LU}=\begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{N1} & l_{N2} & \cdots & 1 \end{bmatrix}\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1N} \\ 0 & u_{22} & \cdots & u_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{NN} \end{bmatrix}. \quad (1)$$

Example 1. A given (3×3) matrix **A**

$$\mathbf{A}=\begin{bmatrix} 2 & 1 & 3 \\ 4 & -1 & 3 \\ -2 & 5 & 10 \end{bmatrix}$$

can be factorized by using the LU decomposition as

$$\mathbf{A}=\mathbf{LU}=\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -3 & 1 \end{bmatrix}\begin{bmatrix} 2 & 1 & 3 \\ 0 & -3 & -3 \\ 0 & 0 & 4 \end{bmatrix}.$$

The LU decomposition is chosen for the research reported in this paper because of its generality. If we take the problem of solving a system of linear equations as an example, the LU decomposition can be directly applied for solving any square system of linear algebraic equations of the form

$$\mathbf{Ax}=\mathbf{b}, \quad (2)$$

where **A** is a given $N \times N$ matrix with the coefficients of the system, **b** is a given vector with the N constant terms, and **x** is a vector with the unknown solutions to be computed. After applying the LU decomposition, the system becomes

$$\mathbf{LUx}=\mathbf{b}. \quad (3)$$

The lower triangular system **Ly** = **b** is then solved by forward substitution. Subsequently, the upper triangular system **Ux** = **y** is solved by back-substitution to obtain the solution **x** of the original system [15].

III. Programming Environments

In this research, the LU decomposition is computed using Eigen, Intel MKL, and MATLAB on the multicore CPU. On the GPU, we employ MATLAB Parallel Computing Toolbox and CUDA extended with the CULA programming library.

A. Eigen

Eigen is an open source C++ library for linear algebra which includes functions for vector and matrix operations, numerical solvers, and other related algorithms [4]. It offers clean and expressive interface and supports explicit vectorization for programs using instruction set extensions such as Intel's Streaming SIMD Extensions (SSE) [4]. However, it lacks an in-built support for multithreaded processing on multicore CPUs. Only some of the Eigen's functions can exploit parallelism using Open Multi-Processing (OpenMP) [5, 14].

B. Intel Math Kernel Libraries

Intel MKL is a library of mathematical functions for Intel and compatible CPUs [9]. It includes Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) routines, fast Fourier transform (FFT) algorithms, and vectorized math functions. Functions implemented in Intel MKL are optimized for Intel multicore processors and allow automatic multithreaded execution on available CPU cores [9]. Intel MKL has compilers for C, C++, and Fortran, and it is available for Windows, Linux, and Mac OS X.

C. MATLAB

MATLAB is an interactive numerical computing environment and a programming language developed by MathWorks [11]. It allows numerical computations, data analysis and visualization, as well as algorithm implementation using a high-level programming language [11]. It also contains a large set of toolboxes for performing computations in mathematics, signal and image processing, computational finance, parallel computing, etc. Therefore, it is widely used by both engineers and scientists.

D. MATLAB Parallel Computing Toolbox

MATLAB's Parallel Computing Toolbox allows the use of multicore CPUs, GPUs, and clusters, for parallel processing of computationally intensive algorithms [17]. This toolbox includes special high-level data types, parallel loops, and numerical algorithms, which permit concurrent execution of programs, through translation and execution of code using Message Passing Interface on the CPU [Pacheco] and CUDA on the GPU [17].

For more details on parallel computing on CPUs, as well as MPI and OpenMP programming frameworks, used by Eigen, Intel MKL, and MATLAB for parallel processing, we refer to [8, 14].

E. CUDA/CULA

CUDA is a parallel programming architecture, framework, and language, developed by Nvidia, for the purposes of implementing and processing general-purpose algorithms on graphics processing units (GPGPU or GPU computing). It supports a massively-parallel programming model constructed around the high-throughput, high-latency GPU architecture.

CULA is a GPU-accelerated library for linear algebra, built on top of the Nvidia CUDA programming framework [3]. It is developed by EM Photonics as two separate tools for dense and sparse linear algebra – CULAdense and CULAsparse [3].

For more details on GPU computing, which attracted a fast-growing interest of researchers in recent years, as well as the CUDA programming framework, we refer to [1, 10, 12, 13].

IV. Implementation Details

For the implementation of LU decomposition, we used the following approach.

We generated square matrices with random single-precision floating numbers using the *rand* function available in MATLAB and MATLAB's Parallel Computing Toolbox. For the same purposes in the Eigen, Intel MKL, and CUDA/CULA implementations, we used the *rand* function from *cstdlib*, with pseudo-random generator number seed set using *srand*(*time*(*NULL*)) function.

For computing the LU decomposition on the CPU, we used the *partialPivLu* function in Eigen, and the [L, U, P] = *lu(A)* command in MATLAB. In Intel MKL, we called the *LAPACKE_dgesv* routine, with matrix **A** set to be in the row major format. For performing the LU decomposition using MATLAB's Parallel Computing Toolbox on the GPU, we stored the matrix **A** in the *gpuArray* data structure [17]. In CUDA/CULA, we first transferred **A** to the GPU using pinned memory, in order to effectively use the PCIe bus between the CPU and the GPU [6, 10], and then called the *culaDeviceSgetrf* function from the CULA library.

V. Experiments

A. Experimental Settings

The experiments were carried out using the computing platform and software presented in Table I. The LU decomposition was performed on matrices of size $N \times N$, for N = 1024, 2048, 4096, 8192, 16384. The elements of the matrices were single-precision floating point numbers, each represented by 4 bytes in the memory. The presented computational times represent average values for 10 program executions for each size of the input matrix.

Table I
The experimental platform and software versions

CPU	*Intel Core i7-920*
core frequency	*2.66 GHz*
number of cores	*4*
Memory	*12 GB DDR3-2000 MHz*
Operating System	*Windows 7 Ultimate 64-bit*
GPU	*Nvidia GTX 650 Ti*
core frequency	*900 MHz*
memory	*1 GB GDDR5 4.2 GHz*
number of cores	*384*
compute version	*2.1*
Library/Software version	
Eigen	*3.2.4*
MATLAB	*2015a*
Intel MKL	*11.2 (in Intel Parallel Studio 2015XE)*
CUDA	*6.5*
CULA	*CULAdense R18*

B. Experimental Results

Results of the experiments preformed as described in the previous subsection are shown in Table 2. Dashed lines in the table indicate that the computation could not be performed for the corresponding matrix size. Figure 1 shows the same information graphically.

Table II
Processing times for different implementations of LU decomposition on the CPU and the GPU

N	Processing time [ms]				
	CPU			GPU	
	Eigen	MATLAB	Intel MKL	MATLAB GPU	CUDA/ CULA
1024	73	36	24	21	17
2048	485	254	204	98	55
4096	3409	1779	922	739	240
8192	24561	12843	5965	-	1230
16384	189188	96813	46374	-	-

When computing on the multicore CPU, the Intel MKL is the fastest implementation. This is due to the automatic multithreaded processing of MKL's numerical routines, if a multicore processor is present in the system, and its optimization for execution on Intel processors. The Intel MKL implementation was on average 1.38× and 2.72× faster than MATLAB and Eigen, respectively. The Eigen implementation is the slowest among the considered implementations because it lacks multithreading capability [5]. Therefore, the LU decomposition using Eigen was performed only on a single core of a quad-core CPU available in our test platform. Note that, for the largest considered matrices (N = 16384), we were able to perform the LU decomposition only on the CPU, due to the GPU memory limitations [6, 10].

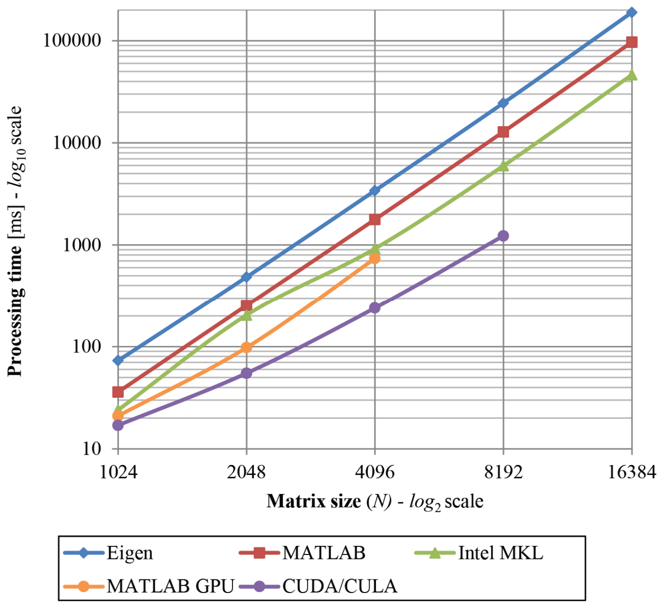

Fig. 1. Processing times for different implementations of LU decomposition on the CPU and the GPU.

The CUDA/CULA GPU implementation is the fastest among all considered CPU and GPU implementations, with the exception of the largest considered matrices, which could not be computed on the GPU, due to the lack of memory. This is a major limitation for computing on the GPU, since it offers shortest processing times, but only for matrices which can be stored in its global memory. This restraint is even stronger when using the MATLAB Parallel Computing Toolbox GPU implementation, which could not handle matrices of size N = 8192, as a consequence of additional memory requirements for storing high-level MATLAB matrix representation (*gpuArray*).

The CUDA/CULA implementation is, for the considered matrix sizes, on average 2.03× faster than the second-best implementation – the MATLAB Parallel Computing Toolbox on the GPU. The difference in speed between these two implementations increases with the size of matrix – from 1.23× for N = 1024 to 3.07× for N = 8192.

We can see that a significant difference in speed exists between the two GPU implementations, even though they both, on the low-level, use CUDA. This can be attributed to the additional time needed to translate high-level MATLAB code to CUDA, as well as to inefficiencies in implementation due to the automatic generation of CUDA code. On the other hand, the CUDA/CULA implementation uses CUDA functions and data structures directly and allows full control of the corresponding code. We can conclude here that the price to pay for faster program execution is extended CUDA/CULA program development time in comparison to using MATLAB's Parallel Computing Toolbox.

Further, when we compare the CUDA/CULA implementation with the considered CPU implementations, we can observe that it is, on average, 11.82×, 6.15×, and 4.34× faster than the Eigen, MATLAB, and Intel MKL, respectively.

VI. Conclusions

In this paper, we presented a comparison of time efficiency of computing the LU decomposition using three different implementations for CPUs and two for GPUs. The experiments performed on matrices with randomly-generated single-precision floating-point numbers showed that the CUDA/CULA GPU implementation offers shortest computation times, but only for matrices which can be stored in the GPU global memory ($N \leq 8192$ on our test platform). The LU decomposition on larger matrices could be performed only on the available multicore CPU, where Intel's MKL is the fastest implementation due to multithreaded execution optimized for Intel processors.

We can conclude that, when processing time is a critical parameter, the LU decomposition should be performed on the GPU using highly-optimized linear algebra libraries like CULA, whenever matrices can be stored in the GPU's memory. In other cases, Intel's MKL is the best out of the considered solutions for performing the LU decomposition on multicore CPUs.

Acknowledgements

The research reported in this paper is partly supported by the Ministry of Education and Science of the Republic of Serbia, projects ON174026 (2011-2015) and III44006 (2011-2015).

References

[1] A. R. Brodtkorb, M. L. Sætra, T. R. Hagen, "Graphics processing unit (GPU) programming strategies and trends in GPU computing", J. of Parallel and Distributed Computing, vol. 73, no. 1, pp. 4-13, 2013.

[2] J. Dongarra, M. Faverge, H. Ltaief, P. Luszczek, "Achieving numerical accuracy and high performance using recursive tile LU factorization", Concurrency and Computation: Practice and Experience, vol. 26, no. 7, pp. 1408–1431, 2014.

[3] EM Photonics, CULA Programming Guide, available from: *http://www.culatools.com/cula_dense_programmers_guide/* [accessed 16 March 2015], version 18, April, 2014.

[4] Eigen C++ Template Library for Linear Algebra, available from: *http://eigen.tuxfamily.org/index.php?title=Main_Page* [accessed March 16, 2015], version 3.2.4, January 2015.

[5] Eigen and Multithreading, available at: http://eigen.tuxfamily.org/dox/TopicMultiThreading.html [accessed March 20, 2015].

[6] D. B. Gajić, R. S. Stanković, "Computing the Vilenkin-Chrestenson transform on a GPU", in J. of Multiple-Valued Logic and Soft Computing, Old City Publishing, Philadelphia, USA, vol. 25, no. 1-4, pp. 317-340, 2015.

[7] G. Golub, C. Van Loan, *Matrix Computations*, 3rd edition, The Johns Hopkins University Press, 1996.

[8] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th edition, Morgan Kaufmann, 2011.

[9] Intel Corporation, Intel Math Kernel Library Reference Manual, available from: *https://software.intel.com/sites/default/files/managed/9d/ c8/mklman.pdf* [accessed March 20, 2015].

[10] D. Kirk, W. M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, 2010.

[11] MathWorks, MATLAB, available from: http://www.mathworks.com/products/matlab [accessed March, 16, 2015], version 8.5 (R2015A), March, 2015.

[12] Nvidia, *Nvidia CUDA Programming Guide*, available from: *http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf* [accessed March, 16, 2015], version 6.5, August, 2014.

[13] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips, "GPU computing", Proc. of the IEEE, vol. 96, no. 5, pp. 279–299, 2008.

[14] P. Pacheco, *An Introduction to Parallel Programming*, Elsevier, 2011.

[15] D. Poole, *Linear Algebra - A Modern Introduction*, 2nd edition, Brooks/Cole, Thomson, 2006.

[16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd edition, Cambridge University Press, 2007.

[17] J. W. Suh, Y. Kim, *Accelerating MATLAB with GPU Computing – A Primer with Examples*, Morgan Kaufmann – Elsevier, 2014.

[18] A. M. Turing, "Rounding-off errors in matrix processes", The Quarterly J. Mechanics and Applied Mathematics, vol. 1, part 3, pp. 287–308, September, 1948.