Parallel Triangular Matrix System Solving on CPU-GPU System

Ryma Mahfoudhi Sami Achour Zaher Mahjoub

University of Tunis El Manar, Faculty of Sciences of Tunis
University Campus - 2092 Manar II, Tunis, Tunisia

***Abstract*— GPU-accelerated computing consists in using a graphics processing unit (GPU) together with a CPU in order to enhance the performance of scientific and engineering applications. With the increasing spread of GPUs as hardware accelerators for scientific applications, several optimized linear algebra libraries have emerged to make use of this additional computing power. In this paper we present an implementation of a recursive algorithm for triangular matrix system solving targeting a hybrid multicore + GPU system.**

Keywords— Divide & Conquer, GPU, Heterogeneous Linear Algebra, Recursive Algorithms, Triangular Matrix System Solving

I. INTRODUCTION

Multi/many core CPUs and GPUs have become a powerful many-core processor and useful tool within the computational science community. The massive parallelism of the graphics cards' architecture offers high performances in many computing applications when the algorithms map well to the characteristics of the GPU.

In linear algebra applications, the available GPU-accelerated implementations cover matrix factorization methods and basic matrix and vector operations, needed to solve most of problems of this field [2],[3].

Since the performance of linear algebra routines is memory hierarchy dependent, a solution for their optimization consists in using the divide and conquer (D&C) paradigm. This latter roughly consists in dividing the processed data into small portions which are loaded and reused by fastest levels of memory hierarchy. The Strassen method for matrix multiplication [4] is in fact a typical D&C based algorithm.

In this work, we are interested in the design of an efficient parallel algorithm for triangular matrix system solving (TMSS) based on a recursive D&C approach and targeting a hybrid CPU-GPU platform. This is an extension of a previous work [4] restricted to the presentation of a fast D&C sequential algorithm for triangular matrix system solving (TMSS) of $O(n^{\log_2 7})$ complexity, n being the system size. The importance of designing these high performance algorithms is motivated by their frequent use in many parallel blocked algorithms [1].

The remainder of the paper is organized as follows. The next section is devoted to the description of the recursive TMSS algorithm used in this implementation [4]. In section 3, we present our parallel algorithm. As to section 4, we detail our implementation on the target machine and the obtained results. Section 5 involves a discussion and generalization of the work. Some conclusions and further perspectives are presented in section 6.

II. RECURSIVE ALGORITHM FOR TRIANGULAR MATRIX SYSTEM SOLVING

In this section, we present the recursive algorithm for solving a triangular matrix system of size, say n and denoted $AX = B$ where both A (a triangular matrix) and B (a dense matrix) are known. LAPACK [5] is a high-performance linear algebra library which provides routines that cover the functionality required in the most used linear algebra algorithms. In particular, for triangular matrix system solving (TMSS) the kernel is commonly named *'trsm'* in the BLAS convention. The *trsm* routine from the level 3 Blas library solves a triangular system of equations with multiple right hand sides. Our proper approach is based on a block recursive algorithm that aims to reducing the computation to matrix multiplication (MM) in order to benefit from the well-known Strassen algorithm (SA), see figure 1. Let us recall here that TMSS is used in blocked LU factorization methods, an important kernel in solving large systems of equations [1].

The procedure we adopt in our TMSS approach is recursively applied until reaching a threshold corresponding to a size smaller than a fixed block size *blks*. The first decomposition writes as follows (see figure 1):

$$A_{11}X_{11} = B_{11},\ A_{21}X_{11} + A_{22}X_{21} = B_{21}$$

$$A_{11}X_{12} = B_{12},\ A_{21}X_{12} + A_{22}X_{22} = B_{22}$$

978-1-5090-4320-0/16/$31.00 ©2016 IEEE

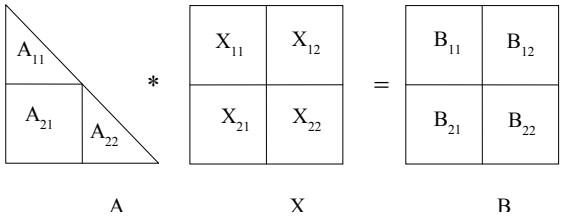


Fig.1. Matrix Splitting for TMSS Algorithm

It is easy to see that our TMSS of size n requires 4 TMSS of size *n/2* and 2 MM of size *n/2*. Thus, the resulting complexity recurrence formula is as follows:

$TMSS(n) = 4TSS(n/2) + 2MM(n/2) + O(n^2)$

We easily deduce that TMSS(n) = $O(n^{\log_2 7})$ [4].

In a previous work [4], we compared our recursive implementation and the *dtrsm (double trsm)* BLAS routine for solving the same problem. We showed that we can determine a level (i.e. number of successive decompositions) for which our algorithm outperforms *dtrsm*. Indeed, for large matrix sizes (n≥1000), the recursive algorithm we designed is about three times faster than the corresponding Blas routine.

It should be noted that the well-known standard algorithm is structured as a nest of three (DO) loops with $O(n^3)$ complexity.

The TMSS algorithm is detailed below.

TMSS Algorithm

Input: A,B,n

Output: X

Begin

If (n ≤ blks) Then

X = Trsm(A,B,n)

Else/ split matrices into four blocks of sizes n/2*/*

X_{11} = TMSS(A_{11},B_{11}, n/2)

X_{12} = TMSS(A_{11},B_{12}, n/2)

X_{21} = TMSS(A_{22}, B_{21}-MM(A_{22},X_{11}) , n/2)

X_{22} = TMSS(A_{22}, B_{22}-MM(A_{21},X_{12}) , n/2)

Endif

End

III. Parallel Triangular Matrix System Solving

A. Related Works

In the last years, several researchers have used GPUs to perform scientific computations. Indeed, in 2005, Galoppo and al. [2] presented a new algorithm to solve dense linear systems by reducing the problem to a series of so called *rasterization* (i.e. decomposition) problems on the GPU. In [6][7], different approaches for sparse matrix-vector multiplication on GPUs are discussed. In [8], the authors presented an implementation of Heller algorithm (the first known recursive triangular matrix inversion algorithm designed in 1974) on heterogeneous multi-CPU/multi-GPU systems. Recently, in [3] Benner & al. addressed the solution of three types of matrix equations and evaluated each solver targeting a hybrid CPUs-GPUs platform. We can notice that there is little research on using GPUs for fast recursive algorithms. In this paper, our aim is to contribute in this evolving new field.

B. Machine Model

We first have to precise the target parallel machine model since it has a direct impact on our approach. We assume that the target multiprocessor system we use consists of a collection of *p* heterogeneous processors (CPU or GPU), denoted P_i *(i=1…p)* each of which being provided with a local memory, a cycle time t_i *(i=1…p)* and connected by a homogeneous interconnection network [9]. Moreover, we assume that the speed of each processor (s_i for P_i) is known and does not vary during the program execution. The standard algorithm for solving a triangular matrix system denoted *AX=B*, consists in solving *n* triangular systems: *AX(i)= B(i)* of size *n* where *X(i)* (resp. *B(i)*) is the i^{th} column of *X* (resp. *B*). The parallelization of this algorithm reduces to assigning to each processor a number of columns fitted (i.e. proportional) to its speed [10].

C. Parallel Recursive Algorithm

1) Case p=2:

We use here a variant of the D&C paradigm, called non equitable (NE) we introduced in a previous work. It consists in recursively decomposing the original problem into a series of sub problems of non-equal sizes (see figure 2). Starting with the formulae seen above (see section II), the TMSS algorithm is segmented into two tasks as follows.

Task 1. T_1: $\{A_{11}X_{11} = B_{11} \cup A_{21}X_{11} + A_{22}X_{21} = B_{21}\}$

Task 2. T_2: $\{A_{11}X_{12} = B_{12} \cup A_{21}X_{12} + A_{22}X_{22} = B_{22}\}$

By this way, each task corresponds to 2 TMSS and 1 MM. We start our study by first studying the case where p=2 processors are available. So we assume that we have at one's disposal two processors P_1 and P_2 with speeds respectively denoted s_1 and s_2 such that $s_1 > s_2$.

Let $s = s_1 + s_2$ and $\rho = s_1/s_2$ *(> 1)*. We then choose a (size) decomposition factor λ *(<1)*. Let $n_1 = \lambda n$ (resp. $n_2 = (1-\lambda)n$) (see figure 2).

By using the standard matrix multiplication, the costs (computing complexity) of the two tasks are as follows (we reduce the expressions to cubic terms):

$$Cost(T_1) = n_1^3 + 2n_1^2n_2 + n_2^2n_1$$

$$Cost(T_2) = n_2^3 + 2n_2^2n_1 + n_1^2n_2$$

The main idea is to determine an adequate factor λ in order to guarantee optimality. Here, we can assign task T_1 to P_1 and task T_2 to P_2. In order to guarantee a perfect load balancing, we have to satisfy the equality: $Cost(T_1)/s_1 = Cost(T_2)/s_2$

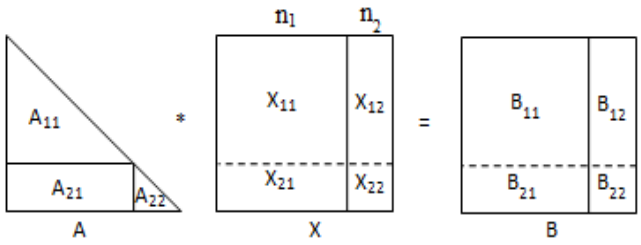


Fig.2. Matrix decomposition for p=2

This easily leads to $\lambda = s_1/s_2$. Afterwards, we'll choose for each processor and according to the size of its sub problem the best sequential routines (for TMSS and adaptive Strassen algorithm for rectangular MM) [4][11].

2) Generalization to p processors:

We recall that the target multiprocessor machine is constituted of p processors, denoted P_1, P_2, ..P_p where the speeds s_i (i=1….p) may be different and are such that $s_i \geq s_{i+1}$ $(i=1...p-1)$. Let $s = \sum s_i$ $(i=1..p)$ and S={s_1.. s_p}. We now detail how to construct an optimal load balancing. The principle consists in generalizing the approach developed for $p=2$ by recursively decomposing the p processors into two sub-sets (each one corresponding to a virtual processor) such that the global speeds of each subset are close as much as possible. Let $\mathcal{P} = \{P_1 ...P_p\} = L_1 \cup L_2$ where L_1 (resp. L_2) involves p_1 (resp. $p_2=p-p_1$) processors and corresponds to a virtual processor denoted $\mathcal{P}_1$ (resp. $\mathcal{P}_2$). Thus, we turn out to the case already seen of two processors P_1 whose speed, denoted s_1, is equal to the sum of the speeds of the processors of L_1 and $\mathcal{P}_2$ whose speed, denoted s_2, is equal to the sum of the speeds of the processors of L_2. L_1 and L_2 will then be decomposed each into two sub-sets and so on. We therefore generate a recursive binary decomposition tree (see figure 3). The solution we adopted consists in a recursive decomposition of the p processors set into two subsets L_1 and L_2 such that L_1 involves a single processor, i.e. the faster (P_1) and L_2 involves the remaining $p-1$ processors. We specify that we already adopted this strategy for designing a parallel recursive algorithm for triangular matrix inversion in a heterogeneous environment [12].

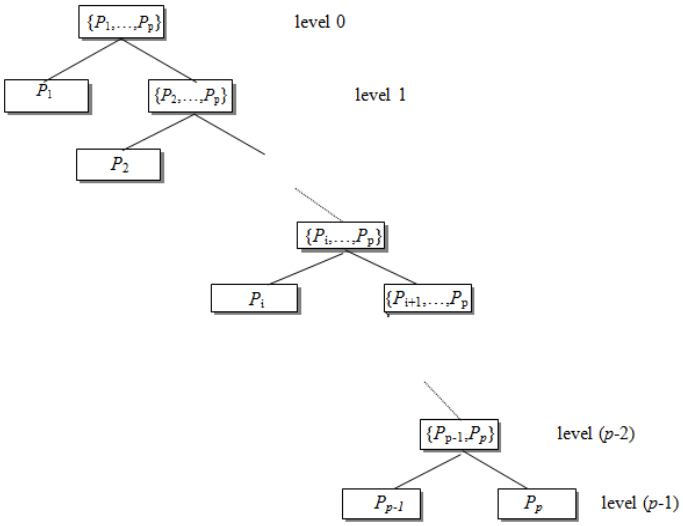


Fig.3. Recursive binary tree corresponding to the non-equitable decomposition

As an illustrative example, we consider the following case where $p = 4$ and $S = \{15, 8, 4, 2\}$. The decomposition procedure is described below (see figure 4).

- Step 1. $\mathcal{P}_2 = \{P_2,P_3,P_4\}$: speed=14, $\mathcal{P}_1=\{P_1\}$: speed =15, $\rho=15/14=1.07$
- Step 2. $\mathcal{P}_2 = \{P_3,P_4\}$: speed=6, $\mathcal{P}_1=\{P_2\}$: speed=8, $\rho=8/6=1.33$
- Step 3. $\mathcal{P}_2 = P_4$: speed=2, $\mathcal{P}_1=\{P_3\}$: speed=4, $\rho=4/2=2$

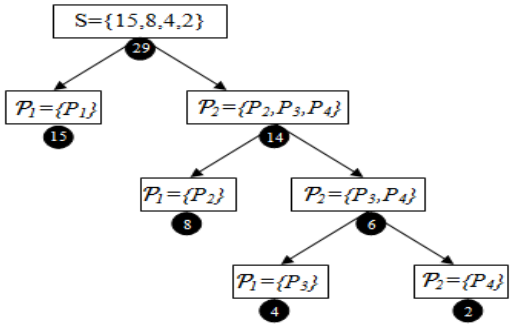


Fig.4. Recursive binary tree for $p=4$, $S=\{15, 8, 4,2\}$

IV. EXPERIMENTAL STUDY

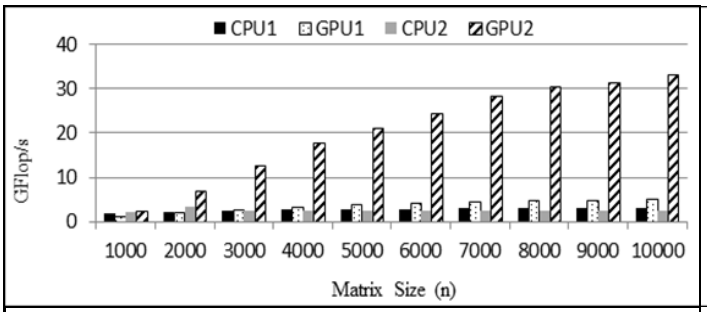
In this section we study the parallelization of our algorithm on a heterogeneous CPU-GPU platform. In fact, GPUs have been a leader of increasing parallelism over the last years, and the Cuda model has recently allowed users to exploit its power. Dense linear algebra emerges so far as a natural choice for Cuda and the GPU since they can naturally be expressed as a blocked computation [14]. The Cuda Basic Linear Algebra Subroutines (cuBLAS) library is provided by NVIDIA. It is an implementation of BLAS on top of CUDA. It includes levels 1, 2 and 3 BLAS routines that deliver 6 to 17 times faster performances than the latest MKL BLAS [14].

For the validation of our approach, we used two contemporary desktop systems; each of them is equipped with a GPU NVIDIA graphics card. Table 1 gives details on the hardware configuration of these two nodes denoted N1 and N2.

Table 1. Hardware of the target CPU-GPU system

Nodes		Reference	#cores	Frequency (GHz)	Memory (GB)
N1	Processor	I3-540	2	3.07	4
	GPU	GT 220	48	1.36	1
N2	Processor	E5400	2	2.7	2
	GPU	GT630	384	1.8	2

We specify that in our experiments we used only the single precision floating point arithmetic. The sizes of the processed systems belong to the range [1000 10000]. The developed codes are based on the use of BLAS, cuBLAS kernels and Openmp directives to explicitly operate both cores of each contributing processor. Each experiment was run 10 times and the mean execution times were kept. We started the experimental study by a preprocessing phase to determine the effective performances (speeds expressed in GFlop/s) of both CPUs and GPUs. We benchmarked our GPU and CPU based TMSS routine provided by BLAS and cuBLAS library for different matrix size. A summary of algorithm performances on the different platforms is depicted in Figure 5 (performances are expressed in GFlop/s). The model generation is done using a statistical tool which is polynomial regression. The principle of this method is as follows: if we have a set of measures (execution times collected at the profiling phase) for different instances (problem sizes), the regression is the derivation of a polynomial describing the behavior of the execution time in function of the problem size. This polynomial has the minimal variance of the series of residuals.



Performance in terms of matrix size n	
CPU1	$T = 10^{-4}n + 2.13$
GPU1	$T = -4*10^{-8}n^2 + 9*10^{-4}n + 0.36$
CPU2	$T = -5*10^{-6}n + 2.5$
GPU2	$T = -3*10^{-7}n^2 + 65*10^{-4}n - 4.26$

Fig. 5. Performance of CPU1, GPU1, CPU2 and GPU2

To summarize we can say the following.
Codes for GPU are much faster than codes for CPU. The processor performance increases with the size and becomes constant (around 3 GFlop/s for CPU1 and 2.55 GFlop/s for CPU2). Since GPU performance follows an increasing quadratic behavior in terms of n, we consider it for load balancing.
We first implemented our algorithm on each node separately, the load balancing among GPU and CPU requires that: $Cost(T_1)/s_1 = Cost(T_2)/s_2$ and consequently:

For node 1:

$$\frac{\lambda}{2(10^{-4}\lambda n + 2.13)} = \frac{1-\lambda}{-4*10^{-8}(1-\lambda)^2n^2 + 9*10^{-4}(1-\lambda)n + 0.36}$$

For node 2:

$$\frac{\lambda}{2(-5*10^{-6}\lambda n + 2.5)} = \frac{1-\lambda}{-3*10^{-7}(1-\lambda)^2n^2 + 65*10^{-4}(1-\lambda)n - 4.26}$$

So the load balancing reduces to solving a cubic equation.

For instance, if we consider the case where $n = 10000$, we have on Node1, $\lambda = 0.61$, so $n_1=6100$ and $n_2=3900$. On Node 2 we have $\lambda = 0.301$, so $n_1=3010$ and $n_2=6990$.

To verify that the load balancing is fair, we measured the execution time of the parallel algorithm on the CPU and GPU on each node (see table 2).

Table 2. Execution time (s) of parallel and sequential algorithm

	Node 1				
	Sequential		Parallel		Speed-up
Matrix Size	CPU1	GPU1	CPU1	GPU1	
1000	1.11	1.05	0.32	0.44	1.25
2000	4.42	2.65	1.02	1.25	1.77
3000	11.01	5.25	2.28	2.40	2.29
4000	22.12	9.4	4.21	4.44	2.49
5000	38.65	15.1	6.62	7.09	2.72
6000	61.64	22.96	10.05	10.62	2.90
7000	92.53	32.45	14.07	15.04	3.08
8000	132.28	44.88	20.94	20.55	3.16
9000	182.06	60.03	26.78	27.51	3.31
10000	243.79	78.86	35.28	36.09	3.38
	Node 2				
	Sequential		Parallel		Speed-up
Matrix Size	CPU2	GPU2	CPU2	GPU2	
1000	1.02	0.49	0.3	0.21	1.68
2000	3.14	0.76	0.48	0.3	3.29
3000	11.85	1.17	0.6	0.40	9.60
4000	24.47	1.77	1.55	1.40	9.80
5000	44.04	2.73	2.26	2.13	9.75
6000	71.85	3.87	3.37	3.06	10.66
7000	109.87	5.11	4.4	4.04	12.48
8000	159.46	6.96	6.12	5.96	13.03
9000	222.24	9.39	8.07	7.72	13.78
10000	299.22	11.99	10.14	9.90	14.76

In order to evaluate the performance of our implementation, we compared it with a CPU platform. In table 2 the following speed-up ratio is depicted [8]:
Speed-up = CPU execution time/(CPU+GPU execution time)

We notice that the speed-up of our implementation increases with the matrix size and reach the value of 3.38 (resp.14.76) for Node1 (resp. Node2)

A second series of experiments was performed on both nodes, connected by a fast Gigabyte network. The communication between machines is handled by the use of MPI library. The load balancing among two nodes requires that $Cost(T_1)/s_1 = Cost(T_2)/s_2$, where s_1 (resp. s_2) is the speed of node1 (resp. node2). As we have previously mentioned, we computed the factor λ ensuring load balancing between two nodes, then within each node, we determine the proportion of CPU/GPU.

The execution steps of this implementation are the following:

a) Send matrix A and the appropriate parts of B from Node1 to GPU1 and Node2

b) Node1 and GPU1 (resp. Node2 and GPU2) compute their parts

c) GPUs transfer their results to Nodes

d) Node2 send its result to Node1

A summary of algorithm performances is depicted in table 3. We notice that the speed-up of our implementation increases with the matrix size.

Table 3. Execution time (s) of parallel and sequential algorithm

Matrix Size	CPUs	CPUs+GPUs	Speed-up
1000	0.26	0.16	1.67
2000	0.92	0.38	2.40
3000	2.85	0.76	3.77
4000	5.81	1.25	4.64
5000	10.29	1.99	5.17
6000	16.59	2.91	5.70
7000	25.11	3.95	6.36
8000	36.15	5.43	6.65
9000	50.04	7.35	6.81
10000	67.17	9.48	7.09

V. LIMITATIONS AND GENERALIZATION

The proposed algorithm requires replicating the input triangular matrix A and distributing the right hand side matrix B using a column-wise block distribution. In this case the algorithm consists of using the whole triangular matrix in all threads to compute a recursive triangular solver only with the corresponding X and B column subsets. By using this data layout, no communication is needed and the algorithm is completely parallel. However, it induces large greater memory requirements due to the triangular matrix replication.

When measuring the performance of an application, one usually focus on double-precision (DP) computations, which are required by scientific applications. In our work we focused on single-precision computations since DP support was recently introduced in the GTX 200 series GPU as an add-on. To evaluate the quality of our implementation, a comparison with the theoretical peak throughput of the hardware is helpful. The theoretical peak of our system throughput is up to 45 GFlop/s. A peak performance of 40 GFlop/s was obtained by our application, though as stated in Section 4, this number is based on execution times that include all kind of overhead from various sources.

On the other hand, we underline that we can generalize our implementation for triangular matrix inversion (TMI) where X is a triangular matrix and B the identity matrix (see figure 6). Consider the case of $p=2$ processors. Choosing a decomposition fraction λ . Let $n_1=\lambda n$ (resp. $n_2=(1-\lambda)n$) be the size of matrix A_{11} (resp. A_{22}).We have to consider here a similar task decomposition as previously seen in order to ensure processor load balancing. For this purpose, we define the two following tasks:

Task 1: T_1: $\{A_{11}X_{11} = I_1 \cup A_{21}X_{11} + A_{22}X_{21} = 0\}$;

Task 2: T_2: $\{A_{22}X_{22} = I_2\}$

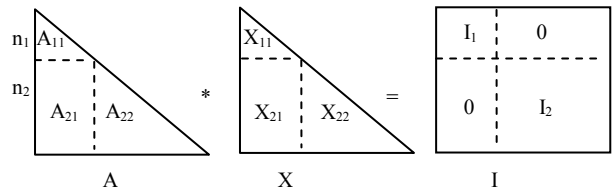


Fig. 6. Matrix decomposition for triangular matrix inversion

We have to notice that this algorithm is optimal which means that it requires no more arithmetic operations than the best-known sequential algorithm.

We remark that only sub-matrix A_{22} must be duplicated.

The same multicore cluster, described previously, was used to evaluate our algorithm.

We obtain a speedup of up to 9 compared to the use of CPUs only (figure 8). A comparison with the previous application shows an improvement of up to 10% in the best case.

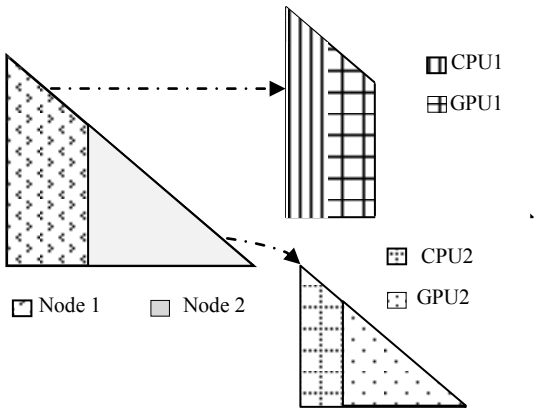


Fig.7.Matrix decomposition and corresponding processing unit

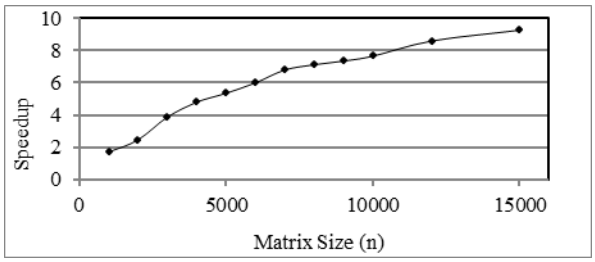


Fig.8. Speedup-TMI

We can also generalize our approach for (parallel) inverting dense matrices by the use of LU factorization. In fact, after its factorization, the inversion of a dense matrix involves the inversion of one or two triangular matrices. In [8], the authors presented an algorithm for dense matrix inversion. We have to notice that their algorithm is not optimal. In fact, the first step is the inversion of the diagonal sub-blocks of the lower/upper triangular matrix in the host memory. Nevertheless, the GPUs remain inactive. On the other hand, the two GPUs must have the same speed.

By adopting our algorithm, we would better manage the above drawbacks. In fact, the two hosts must contribute in the computation of each triangular matrix inversion after the LU factorization algorithm.

VI. Conclusion

Our study could prove clear benefits of using GPU in dense linear algebra operation (level-3 BLAS) like matrix system solving and matrix inversion.

Several questions about the computational performance improvement of recursive linear algebra with GPUs could be explored in more details. As future work, we plan to model and evaluate the performance of our algorithm for multiple levels of parallelism, and to extend the same approach to other linear algebra operations. It will also be important to evaluate the using of double precision arithmetic.

References

[1] D. Maurerand C. Wieners, A parallel block LU decomposition method for distributed finite element matrice, Parallel Computing. 37(12), pp.742–758, (2011)

[2] N. Galoppo, N. K. Govindaraju, M. Henson and D. Manocha, LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, Proceedings of the ACM/IEEE SC-05 Conference, (2005)

[3] P. Benner, P. Ezzatti, H. Mena, E. S. Quintana-Ort and A. Remn, Solving Matrix Equations on Multi-Core and Many-Core Architectures, Algorithms.6(4), pp. 857-870, (2013)

[4] R. Mahfoudhi, S. Achour and Z. Mahjoub, Efficient Recursive Implementations for Linear Algebra Operations, International Conference on Automation, Control, Engineering and Computer Science (ACECS'14), (2014)

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Green- baum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users Guide. SIAM, Philadelphia, PA, third edition, (1999)

[6] M. M. Baskaran and R Bordawekar. Optimizing sparse matrix-vector multiplication on gpus. IBM Technical Report RC24704, (2008)

[7] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. NVIDIA Technical Report NVR-2008-004, (2008)

[8] F. Ries, T and De Marco, R. Guerrieri: Triangular Matrix Inversion on Heterogeneous Multicore Systems. IEEE transactions on parallel and distributed Systems, (23), pp.177-184, (2012)

[9] W. Nasri, Z. Mahjoub and D. Trystram: Computing the inverse of a triangular matrix on heterogeneous clusters, in Algorithms and Tools for Parallel Computing on Heterogeneous Clusters, pp. 67-78, (2007)

[10] A. Legrand and Y.Robert, Algorthmique parallèle, Dunod, Paris, (2003)

[11] P.D'Alberto and A. Nicolau, Adaptive Strassen's matrix multiplication, ICS 2007, 284-292, (2007)

[12] R. Mahfoudhi, Z. Mahjoub, W. Nasri, Parallel Communication-Avoiding Algorithm for Triangular Matrix Inversion on Homogeneous and Heterogeneous Platforms, International Journal of Parallel Programming, 2014.

[13] R. Hockney, The communication challenge for MPP: Intel Paragon and Meiko CS-2, Parallel Computing, vol. 20, pp. 389–398, (1994)

[14] Nvidia, https://developer.nvidia.com/cuBLAS