Acta Polytechnica **53**(2):70–74, 2013

available online at `http://ctn.cvut.cz/ap/`

CLOCK MATH — A SYSTEM FOR SOLVING SLES EXACTLY

Jakub Hladík, Róbert Lórencz, Ivan Šimeček*

Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague, Czech republic

* corresponding author: `xsimecek@fit.cvut.cz`

Abstract. In this paper, we present a GPU-accelerated hybrid system that solves ill-conditioned systems of linear equations exactly. Exactly means without rounding errors due to using integer arithmetics. First, we scale floating-point numbers up to integers, then we solve dozens of SLEs within different modular arithmetics and then we assemble sub-solutions back using the Chinese remainder theorem. This approach effectively bypasses current CPU floating-point limitations. The system is capable of solving Hilbert's matrix without losing a single bit of precision, and with a significant speedup compared to existing CPU solvers.

Keywords: integer arithmetic, modular arithmetic, Hilbert's matrix, error-free, GPGPU, solver, OpenCL, optimizations.

1. Introduction

Solving linear algebraic equations solution is quite a frequent task in numerical mathematics. One may often find difficulties, while solving problems of the ill-conditioned matrix. Stability of the solution cannot be ensured for large dense sets of linear equations. Rounding error during the numerical computation cannot be tolerated. Methods have been developed that minimize the influence of rounding errors on the solution.

One method that we use relies on modular arithmetics [2] in order to solve dense systems of linear equations precisely. The underlying idea sounds quite simple – bypass floating point rounding limitations by using integer arithmetics. It consists of three parts – converting floating point numbers into integers, solving multiple systems of linear equations within their modules, and finally converting sub-solutions back using the Chinese remainder theorem.

Nowadays, GPGPU (General-Purpose computing on the GPU) is a trending topic in high-performance computing. Since GPGPU began in 2006 hundreds of articles have been published every year. GPGPU is usually used to accelerate data-parallel algorithms, and that is our case. Dozens of systems of linear equations emerge during the second step of our computation. Each of them can be solved in parallel on the GPU, hence speedup is achieved.

In this paper, we present a GPU-accelerated solver of ill-conditioned systems of linear equations. Section 3 gives a brief overview of the mathematics that is used. Section 4 describes the GPU architecture in general, and presents issues that we faced while optimizing the computation. Finally, Section 5 shows our measured results – the speedup and a comparison with existing systems solving similar problem.

2. State of the Art

Nearly one half of the problems solved in numerical mathematics lead to problems in linear algebra. The primary objective of the numerical methods used in linear algebra is to solve sets of linear equations ... They are solved in a chosen computer arithmetic, most often floating-point arithmetic. Floating-point arithmetic's well known advantages have led to its widespread usage, yet it also has its important disadvantages, sometimes resulting in severe problems.

The disadvantages of floating-point arithmetic include mainly the generation and accumulation of rounding errors during the calculation, and non-uniform distribution of values in the real number subset. There are a variety of alternatives to floating point representation and associated arithmetic operations, including modular, logarithmic, p-adic, and continued fraction arithmetic.

Many numerical problems lead to SLEs with dense and large matrices. The rounding error sensitivity of such systems can be so grave that they can be considered to be ill-conditioned. To solve the problems, many numerical direct and iterative methods have been developed that tackle these systems with higher or lower success.

Apart from many numerical methods, various programming and computing tools have been developed for solving rounding error sensitive SLEs. There are many computer libraries that allow the user to choose an optimal virtual length of a computer word that fully or partially eliminates the destructive influence of rounding errors. There are also arithmetic units that tackle this problem by grouping arithmetic operations, and thus they create a longer computer word for operations sensitive to rounding. Software tools that eliminate rounding errors include libraries for precision computing, such as GMP (GNU Multiple

Precision Arithmetic Library). Unfortunately, these tools lead to such great time complexity when they are used to solve rounding error sensitive SLEs that they may not be applicable to practical problems.

Another limiting factor when solving SLEs with a square matrix is the inherent algorithmic time complexity $O(n^3)$, where n is the dimension of the matrix. To solve large SLEs with a large number of equations, special computers and processors have been developed. These are vector computers. Until recently, the vector processors available on the market were CRAY SV1ex (2001), Fujitsu VPP5000 (1999) and VMIPS (2001). IBM Virtual Vector Architecture brings vector processing features to its most recent POWER6 mainframe processors (2007). Another suitable architecture for solving rounding error sensitive SLEs is SIMD (Single Instruction, Multiple Data). Today the SIMD features can be found as extensions of the standard processor instruction set, for example, Intel-SSE, AMD-3DNow! or Motorola-Altivec. These are used in multimedia applications such as video, 3D graphics, animation, audio, etc. The recent trend is GPGPU. Today, at least three of ten top supercomputers in the top 500 list use GPU for numerical acceleration. Both major GPU manufacturers, NVIDIA and AMD, offer the possibility to run general purpose computation on their GPUs.

At present there are no single-purpose numerical accelerators for solving rounding error sensitive SLEs. We use common NVIDIA Fermi graphics cards together with Intel Core processors to accelerate the solution.

3. Mathematical Background

Let us take a system of linear equations (see [1]):

$$\mathbf{Ax} = \mathbf{b}, \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{N\times N}$ is the matrix of a system of N equations of N unknowns, $\mathbf{b} \in \mathbb{R}^N$ is the right-side vector and $\mathbf{x} \in \mathbb{R}^N$ is the desired vector, the solution of our system of linear equations.

3.1. Matrix Scaling

Matrix scaling is the first thing to do. The goal of matrix scaling is to adjust all the floating-point numbers of the matrix to their corresponding integer versions. Basically, every matrix row is multiplied by its scaling constant (scalar multiplication). This has to be done without losing a single bit of precision. This condition will be achieved only when the scaling constant is 2^n, where $n \in \mathbb{N}_1$.

The question now is how to determine the scaling constant. First, we should continue by finding the smallest absolute value element (closest to zero) in each row. Then, we extract the absolute value of its exponent and multiply it by the 2^{53} constant, because a significant mantissa bit size in the IEEE 754 [6] double precision floating point number format is 53 bits long. Finally, the scaling constant s is computed:

$$s = 2^{53} * 2^{|\exp(\min_{\text{row}})|}, \tag{2}$$

where exp is a function that extracts and returns the exponent (as an integer – power of 2) and $\min_{\text{row}}$ is the element in the row closest to zero. The approach being used is explained in greater detail (and with alternatives) in [2].

3.2. Solving a System of Linear Equations

From the previous step we have an properly scaled-up system of linear equations:

$$\mathbf{Ax} = \mathbf{b}, \tag{3}$$

where $\mathbf{a}_{ij}$ matrix $\mathbf{A}$ elements and $\mathbf{x}_i$ and $\mathbf{b}_i$ vector elements are just big integers.

We solve it using a multi-modulus arithmetics over the commutative ring $(\mathbb{Z}_\beta, \oplus, \odot)$ with a base vector β that is equivalent to the single-modulus arithmetics over $(\mathbb{Z}_M, +, \cdot)$ and module M.

M has to be a big enough positive integer to avoid rounding errors during our computation. Hadamard's determinant D estimate of matrix A can be used to estimate the maximum M value:

$$|D|^2 \le \prod_{i=1}^{n} \left(|a_{i1}^2| + |a_{i2}^2| + \cdots + |a_{in}^2|\right). \tag{4}$$

The highest value of M that could appear during the computation:

$$M > 2\max\left\{\begin{array}{c} n^{\frac{n}{2}} \max(a_{ij}^n), \quad i,j=1,2,\ldots,n, \\ n(n-1)^{\frac{(n-1)}{2}} \max(a_{ij})^{n-1} \max(y_i), \\ i,j=1,2,\ldots,n \end{array}\right\} \tag{5}$$

and

$$\gcd(M, D) = 1. \tag{6}$$

We also need the following conditions for the vector $\beta = (m_1, m_2, \ldots, m_r)$ and module M to be satisfied:

- $\prod_i^r m_i = M$;
- $m_1 < m_2 < \cdots < m_r$;
- $m_1, m_2, \ldots . m_r$ are prime numbers.

The following condition for the SLE (given by Eq. (3)) determinant is satisfied when:

$$|D|_{m_i} \neq 0, \quad i = 1, 2, \ldots, r, \tag{7}$$

then the SLE (from Eq. (3)) solved within $(\mathbb{Z}_\beta, \oplus, \odot)$ due to vector β can be expressed as:

$$|\mathbf{Ax}|_{m_i} = |\mathbf{b}|_{m_i}, \tag{8}$$

or, for individual modules m_i of vector β:

$$|\mathbf{Ax}|_{\mathbf{m_i}} = |\mathbf{b}|_{\mathbf{m_i}}, \quad i = 1, 2, \ldots, r. \tag{9}$$

The following expression is also valid for Eq. (3) within $(\mathbb{Z}^\beta, \oplus, \odot)$:

$$|\mathbf{A}\mathbf{A}^{-1}|_\beta = |\mathbf{A}^{-1}\mathbf{A}|_\beta = \mathbf{E} \tag{10}$$

and

$$|\mathbf{x}|_\beta = |\mathbf{A}^{-1}\mathbf{b}|_\beta, \tag{11}$$

where $\mathbf{E}$ is the identity matrix. To solve the SLE from Eq. (9) within the specific modular arithmetics we will use the Gauss-Jordan elimination algorithm with *non-zero pivotization*. The difference from the original Gauss-Jordan elimination is the usage of modular arithmetics for all algorithm steps. There is also pivotization simplification – we do not need to find the greatest element in the elimination step, a non-zero element is good enough.

Using the GJ elimination algorithm, let us have an matrix $\mathbf{W}$ of dimensions $n \times (n+1)$ consisting of matrix $\mathbf{A}$ and vector $\mathbf{b}$:

$$\mathbf{W} = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right). \tag{12}$$

The goal of the algorithm is to eliminate all $\mathbf{W}$ elements one by one to get the resulting $\mathbf{x}$ vector to Eq. (11) after $\approx n^3$ elimination steps.

3.3. Inverse Transformation

After completing the two previous steps, we have a set of vectors $\mathbf{x}$ within each module from $\beta = (m_1, m_2, \ldots, m_r)$. They represent a sub-solution for the specific $(\mathbb{Z}_{m_i}, +, \cdot)$ arithmetic. Now we advance to the inverse transformation back to $(\mathbb{Z}_\beta, \oplus, \odot)$. The necessary condition for each β module solution $\mathbf{x}$ is its non-zero determinant from Eq. (7).

The algorithm used for this transformation is the Chinese remainder theorem. The product of our last step consists of the vector $\mathbf{x}$ elements in the form of fractions that contain the solution of SLE (3).

4. GPU and Optimizations

GPGPU is the area of our research, so we optimize on the PC graphical hardware. There were several platforms to choose from:

- NVIDIA CUDA, the mainstream platform today, was rejected; it is proprietary, and its usage is limited to NVIDIA hardware;
- Microsoft DirectX 11 DirectCompute was rejected; it is bound to the Microsoft Windows platform only (not used in HPC in general);
- OpenCL is an open standard that works well across platforms (GNU/Linux, Microsoft Windows and Apple Mac) and on all latest graphical hardware (NVIDIA, AMD and latest Intel GPUs); we use OpenCL for the implementation.

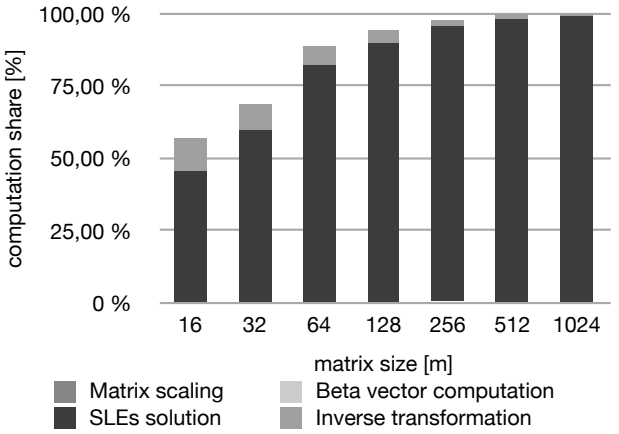

Figure 1. Measured computation shares.

4.1. Profiling

The whole SLE solving process is a rather time-consuming task (Fig. 1), so optimizations had to be made. First, all the big number ALU operations – Eq. (5) and Section 3.3 – are performed using the GMP library, which is tuned for this platform.

Solving dozens of systems of linear equations using modular arithmetics (Section 3.2) within the β vector is the most computation-intensive part of our task, see Fig. 1.

According to Ahmdal's Law (expressed in Eq. (13), the benefit (overall speedup) from optimizing part of the computation is equal to:

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{S}}, \tag{13}$$

where P is the proportion of the computation we are optimizing and S is its speedup. Because modular arithmetics SLEs take a great amount of time to solve, we optimize the matrix elimination.

4.2. GPU architecture

All modern common GPUs have a similar architecture (from our optimization type point of view). They consist of a global memory (1–4 GB) and several (2–24) SMP[1] processors (example in Fig. 2). SMP features a scheduler of lightweight threads that have zero overhead. Each SMP contains dozens (8–48) of processor cores that share the SMP memory (shared memory ≈ 64 kB) and execute the same code. If it happens that one processor core branches differently, the performance goes down dramatically. Each processor core also has its own integer and floating-point unit. More details with examples and a description of the CUDA platform are given in [5].

The processor cores have access to the GPU global RAM, but the memory access has to be aligned. Otherwise performance will be affected. We used SMP processor cores to load part of the matrix (the row

[1] SMP – symmetric multiprocessor

Matrix size [n]	***Clock Math*** [s]	**linSolve-0.7.15** [s]	**Speedup** [%]
16	0.006	0.006425	16.82
32	0.006	0.014921	42.10
64	0.029	0.046871	61.97
128	0.117	0.195461	67.06
256	1.043	1.401498	34.40
512	11.252	16.215546	44.11
1024	124.375	212.557002	70.90

TABLE 1. Measured results: comparison of execution times for *Clock Math Solver*, for linSolve-0.7.15 and corresponding speedups.

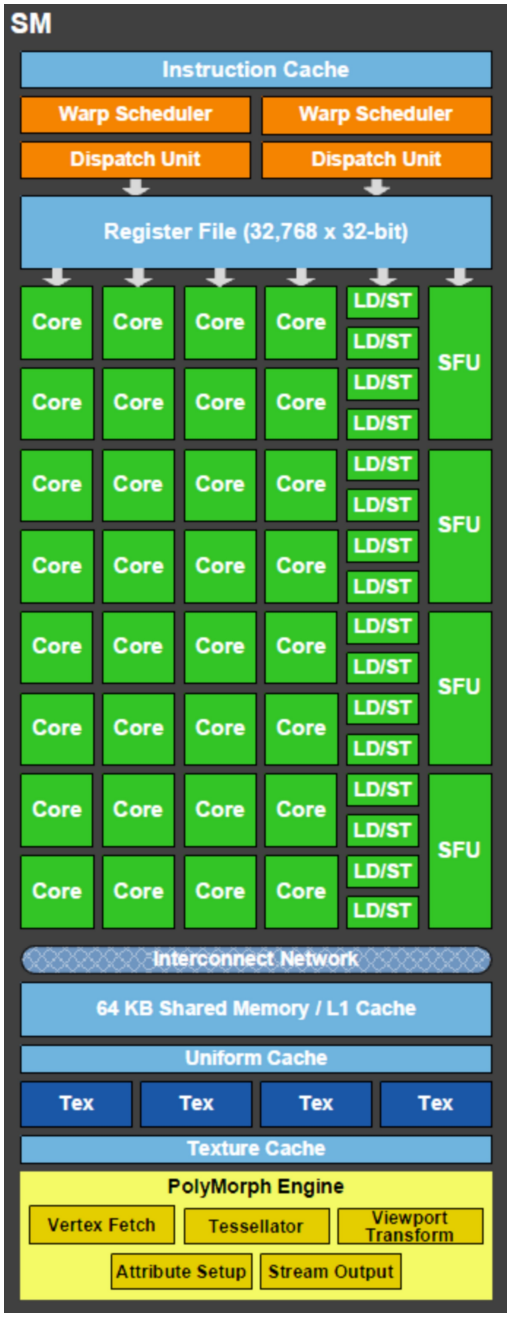

FIGURE 2. NVIDIA Fermi SMP [5].

that contains the current pivot). We used it for computation and then stored the elimination step results back to global memory.

4.3. GPU KERNELS, OPTIMIZATIONS

The OpenCL framework defines the term *kernel*. It is an function written in slightly modified C99 language which is to be executed on an OpenCL capable device. In parallel, OpenCL runtime executes the same kernel on every processor core within an SMP. In the example of the *SAXPY* function, we are able to process (8–48) vector elements at once. However, our case is not so simple.

Our system is currently limited to the matrix size of 4096×4096. The row with the current pivot is used in n multiply-add-modulo vector operations. Hence, it is quite useful performance-wise to cache it in the SMP shared memory. Current generation NVIDIA GPUs have shared memory size of 48 kB. One matrix row fits the local memory easily ($4096 * 8 = 32758$ B). The processor cores fetch a matrix row from global memory and compute the inverse of the pivot element. Then the processor cores multiply the cached row and multiply-add-modulo other rows in global memory. More details, including the processor core synchronizations and source code samples, are available online[2].

5. RESULTS

After embedding architecture-related optimizations into our *Clock Math* solver and verifying the correctness of our results, we were finally able to benchmark them. There are not many up-to-date solvers like ours currently. We benchmarked primarily against linSolve-0.7.15 [7], a highly optimized solver with performance critical parts written in x86 assembler. The results (Tab. 1) are very satisfying. *Clock Math* solver outperforms linSolve-0.7.15 almost twice for larger matrices.

[2] `http://www.github.com/kubbing/Clock-Math`

6. Conclusion

We have presented the working system for solving ill-conditioned systems of linear equations exactly. We have managed to effectively bypass floating-point rounding error by using modular arithmetics. Most of the entire computation has taken place by solving SLEs within their corresponding modules. This part of the computation can be (and is) solved in parallel on the common-grade GPU. GPU is capable of solving several SLEs at once (8–16, depending on its memory size) and also each system in parallel on its SPE (8–40 cores). Double parallelism has been utilized, so significant speedup is achieved compared to other implementations. We are currently working on advanced kernel optimizations with larger matrix support.

7. Future Work

As we now have a working system, we would like to proceed to:

- add support for both single and double precision for matrix elimination, as the β vector module count and GPU performance may differ;
- add support for larger matrices than 4096×4096 – optimize SMP shared memory usage;
- add support for automatic kernel group size tuning for larger matrices, as the group size lowers the memory access time on different GPUs/architectures;
- examine the modulus operation performance across different GPU architectures and further optimize the *SAXPY*, respectively *DAXPY* functions $(\mathrm{mod}\, m)$;
- utilize OpenMPI library to add cluster support;
- adjust and run the solver on our university STAR cluster to test AMD's OpenCL CPU implementation.

Acknowledgements

This research has been supported by CESNET Development Fund project 390/2010 and by the grant SGS12/097/OHK3/1T/18.

References

[1] Lórencz, R.: *Aplikovaná numerická matematika a kryptografie*. Vydavatelství ČVUT, 2004

[2] Gregory, R.T.: *Error-free computation: why it is needed and methods for doing it*. R. E. Krieger Pub Co, 1980

[3] Lórencz, R., Morháč, M.: *A modular system for solving linear equations exactly*. Computers and Artificial Intelligence, Vol. 12, 1992

[4] Zahradnický, T.: *MOSFET Parameter Extraction Optimization*. Ph.D. thesis, Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague, 2010

[5] Kirk, D.B., Hwu, W.W.: *Programming Massively Parallel Processors, A Hands-on approach*. Morgan Kaufmann Publishers, 2010

[6] *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2008, pages 1-58, 2008

[7] Vondra, L., Lórencz, R.: *System for solving linear equation systems*. Seminar on Numerical Analysis, pages 171-174, Technical University Liberec, 2012