Solver for Systems of Linear Equations with Infinite Precision on a GPU Cluster

Jiří Khun

Department of Computer Systems, Faculty of Information Technologies,
Czech Technical University in Prague, Prague, Czech Republic

jiri.khun@fit.cvut.cz

Abstract. *In this paper, we would like to introduce an accelerated solver for systems of linear equations with an infinite precision designed for GPU clusters. The infinite precision means that the system can provide a precise solution without any rounding error. These errors usually come from limited precision of floating point values within their natural computer representation. In a simplified description, the system is using modular arithmetic for transforming an original SLE into dozens of integer SLEs that are solved in parallel via a GPU cluster. In the final step, partial results are used for a calculation of the final solution. The usage of GPUs plays a key role in terms of performance because the whole process is computationally very intensive but also well scalable. An overall performance of the solver directly depends on the cluster's configuration but it can offer the performance far beyond a single computing node.*

Keywords

AMD; Beowulf cluster; dense matrix-vector multiplication; dense matrix representation; GPU; GPGPU; GPU cluster; high performance numerical linear algebra; modular arithmetic; OpenCL; OpenMP; MPI; parallel execution; system of linear equations;

1. Introduction

There are many scientific tasks that need to solve large systems of linear equations as a part of their solutions. We can found such tasks in many areas of modern science like mathematics, physics, chemistry, biology, economics, and many others.

Tasks taken from the real world usually work with floating-point numbers and that can be a problem. Representations of floating-point values stored in computers have typically only limited precision that leads to limited precision of calculations and consequently their results. That can be a disaster if we are working with ill-conditioned tasks where only small difference in input values causes enormous difference in results. Therefore several methods have been developed in order to minimize such issues and provide as precise solution as possible.

1.1. Rounding-free arithmetic

In our solver we are using a method [1] that incorporates modular arithmetic in order to provide solution without any rounding error i.e. with the infinite precision. The method is relatively simple and consists of a few steps. At the beginning input values are transformed (scaled up) into integer values. Than the integer SLE is solved in sufficient amount of different modular arithmetic. At the end all results from individual solutions are put together for a calculation of final solution via Mixed-radix sort algorithm. Despite the simplicity the whole process is computationally very intensive because the method requires to solve many SLEs in different modular arithmetic. Therefore we utilized a GPU cluster as an important source of computational power.

1.2. CPU/GPU

Todays GPUs are able to provide general-purpose computations (GPGPU) with a peak performance far beyond todays CPUs. Their power lies in a massive parallelism because they usually consist of hundreds or thousands of small and simple computation cores. The GPGPU approach requires very good planning in terms how the task utilize a GPU device and its parts. Only with an appropriate program model and fine-tuning it is possible to unlock full potential of a GPU.

During the process we need to solve dozens of individual SLEs and that can be processed in parallel. Even the solving process itself consists of steps that can be done in parallel. Therefore the usage of GPU is an obvious choice.

1.3. GPU cluster

Even high performance of a single GPU can be often insufficient for such demanding task like solving thousands of large SLEs. Therefore our latest version of the solver involves computing via cluster of GPUs.

A GPU cluster consists of n computing nodes, each equipped with GPGPU device (next to a necessary CPU and RAM). Every node is connected with others by some kind of high-speed bus and can provide the same functions like standalone computer. It is an example of distributed memory multiprocessor system.

In academical environment it is typical that researchers are working with relatively cheap commodity-grade computers. These personal computers or even laptops can be equipped with appropriate GPU and networked into a small local area network. This is usually called a Beowulf (GPU) cluster and our research is also using such type of system.

2. State of the art

The topic of this article is an intersection of two large research areas. The first is computation with floating-point values and resulted difficulties like limited precision, rounding, accumulation of errors, etc. The second is computation intensiveness rising from necessity to solve large systems of linear equations which is a task with a time complexity equal to $O(n^3)$. Science made a significant progress in both of these areas in past decades.

2.1. Avoiding limited precision of floating-point representation

The main difficulty during usage of standard floating-point representation, according to the technical standard IEEE 754 [12], within a computer program is the limited precision resulting into rounding errors and their following accumulation during the computation. The IEEE 754 standard also defines a non-uniform distribution of all possible values. It can lead to another limitation in terms of precision.

Therefore several other approaches including appropriate arithmetic operations have been developed for floating-point representation e.g. logarithmic, p-adic, arithmetic with a continued fraction, modular or interval arithmetic [6].

Another possible approach how to avoid the problems with the standard floating-point representation mentioned above is using special libraries allowing user-defined length of floating-points variables. The libraries, e.g. GMP (GNU Multiple Precision Arithmetic Library) [7] or its fork MPIR (Multiple Precision Integers and Rationals) [8], can bypass the limitation of standard floating-point representation, with a help of complex data structures based on basic data types, and bring almost unlimited precision of calculations. A significant drawback is a high memory and computational demands that do not allow to use them effectively for computational intensive calculations like solving large systems of linear equations.

2.2. Acceleration by vector processing

As already mentioned above the solving of SLEs is a difficult task that requires a lot of computational power. Time complexity of solving a single SLE with a Gauss-Jordan elimination is $O(n^3)$, where n is a number of unknowns within the system. Therefore several approaches have been developed allowing solving of larger SLE. First effective approach for solving large SLE had been represented by vector computers. These devices equipped with a special processing units called vector processor had been capable to accelerate calculations with vectors and consequently whole solving process. Among first computers that fully utilized vector technique was for example famous Cray-1 in 1976 [13]. Than many other types followed its path and vector processing became a standard and necessary approach how to achieve the highest performance.

Experience learned from vector computing became an inspiration for SIMD (Single instruction, multiple data) technique that influenced design of standard CPUs during 90s. The main idea was simple. One instruction is applied in parallel on several operands. This is very useful for solving large SLEs where dozens of very same operations must be processed on different data. SIMD instructions are still one of the main trends in terms of modern CPUs development and enhancing standard instruction set with vector operations. There are several representatives of the SIMD technology today, for example MMX, SSE and AVX developed by Intel, 3DNow! (AMD) or NEON (ARM).

A very modern way how to handle vector operations and consequently solve large SLEs is using GPU for the general-purpose computing. First unified GPU in the world: NVidia's G80, capable to handle GPGPU operations, was introduced in 2006 [14] and since them all major vendors offer GPUs that can provide the general-purpose computations. As mentioned above the key aspect of GPU computational performance lies within the cooperation between many relatively simple computational elements - processors. Therefore a modern GPUs internal architecture is quite similar to a supercomputer with many computational nodes where every of them is processing only small part of the given task. This approach is logically leading to increased demands in terms of synchronization and utilization of computational resources. We have chosen the GPGPU approach because it can offer far more computational power than presents CPUs and due to the inherited parallelism it seems to be a logical choice for our task where it is necessary to solve large amount of independent SLEs.

2.3. Related work

There are few other papers dealing with the similar topic like our work. All these works took the mathematical background from [1] and tried to solve the dozens of SLEs with a different approach. First work [2] is proposing special hardware architecture for accelerating the whole process. Second work [3] is using OpenMPI technology for parallelization of the computing process among many independent computing nodes (without involving GPUs). Third article [5] is describing acceleration via GPU but the work is using only single GPU and different technology and also partially different mathematical approach than our work.

3. Mathematical background [4][5]

Let us start with a simple system of linear equations:

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

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

3.1. Matrix Scaling

At the beginning of the whole process it is necessary to scale up the whole input matrix with appropriate constant in order to avoid a computation with floating-point values. That means that the matrix is transformed into its integer representation. It must be done carefully without loosing a single bit of precision. We can achieve this condition by using appropriate scaling constant that will be 2^n, where $n \subset \mathbb{N}$.

The approach how to determine the scaling constant is following. At the start we need to find the smallest element in the absolute form of each row. Then we have to extract the absolute value of its exponent and multiply this value by 2^{53}. This constant comes from the simple fact that a significant mantissa bit size in the technical standard IEEE 754 for double precision floating point number format is 53 bits long. Scaling constant s is computed as follows:

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

where exp represents a function that extracts and returns the exponent (as an integer - power of 2) and min_{row} is the row's element closes to zero (the smallest one in the absolute value). More details can be found in [1].

3.2. System of Linear Equations Solution

After the scaling we have the original system of linear equations scaled up into big integer values:

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

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

M has to have integer with big enough positive value in order to avoid rounding errors during the computation. For estimation of the M value we are using Hadamard's determinant D calculated from matrix A.

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

The highest value of the M that could appear during the computation can be estimated in following way:

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

where

$$i, j = 1, 2, ..., n$$

and

$$gcd(M, D) = 1$$

We also need to keep following conditions for vector $\beta = (m_1, m_2, ..., m_r)$ and module M:

- $\prod_i^r m_i = M$
- $m_1 < m_2 < ... < m_r$
- $m_1, m_2, ...m_r$ are primes

The following condition for the SLE (Eg. (3)) determinant is satisfied when:

$$|D|_{m_i} \neq 0, i = 1, 2, ..., r \tag{6}$$

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

$$|\mathbf{Ax}|_{m_i} = |\mathbf{b}|_{m_i} \tag{7}$$

or, for individual modules m_i of vector β:

$$|\mathbf{Ax}|_{m_i} = |\mathbf{b}|_{m_i}, i = 1, 2, ..., r \tag{8}$$

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

$$\left|\mathbf{AA^{-1}}\right|_\beta = \left|\mathbf{A^{-1}A}\right|_\beta = \mathbf{E} \tag{9}$$

and

$$|\mathbf{x}|_\beta = \left|\mathbf{A^{-1}b}\right|_\beta \tag{10}$$

where $\mathbf{E}$ is the identity matrix. We will use Gauss-Jordan elimination algorithm with the *non-zero pivotization* for solving the SLEs from Eg. (8) within the specific modular arithmetics. Despite the original Gauss-Jordan elimination we are using the modular arithmetics for all algorithm's steps. There is also a simplification within the process. We do not need to find the greatest element. In the elimination steps every non-zero value is appropriate due to usage of the modular arithmetic.

The G-J elimination algorithm is relatively simple. We have a matrix $\mathbf{W}$ of dimension $n \times (n + 1)$ consisting of matrix $\mathbf{A}$ and vector $\mathbf{b}$:

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

The goal of the algorithm is to eliminate all $\mathbf{W}$ elements one by one to get the resulting $\mathbf{x}$ vector to Eg. (10) after $\approx n^3$ elimination steps.

3.3. Inverse Transformation

After completing two previous steps we have a set of vectors $\mathbf{x}$ within each module from $\beta = (m_1, m_2, ..., m_r)$. They represent sub-solutions for the specific $(\mathbb{Z}_M, +, \cdot)$ arithmetic. We will use this sub-solutions within inverse transformation back to $(\mathbb{Z}_\beta, \oplus, \odot)$. The mandatory condition for this process is a non-zero determinant from Eq.(6).

We are using Mixed-radix algorithm [2] for this inverse transformation. The final result of the calculation is the vector $\mathbf{x}$ in form of fractions that represents the solution of the input SLE (Eq. (3)).

4. Overall architecture

This section is focused on the overall architecture from the upper point of view. Processes within individual nodes will be described in the next section.

Our solver is based on simple but well scalable master-slave architecture. A single master node is preparing work and receiving results from various number of slave nodes. At the end all partial results from slave nodes are used for calculation of a final result back in master node.

There is no difference between individual nodes in terms of program code. Also the master node can perform the function of slave and do the same work in case of inactivity.

4.1. The exact sequence of the algorithm steps

1. An input SLE (matrix) is scaled up from floating point to integer values within master node.
2. The scaled up matrix is distributed from master to all slaves.
3. The master calculates all modulus (numbers) necessary for the calculation.
4. Based on number of slaves (and their computational performance) master divides the modulus into groups of various size and sends them to individual slaves. One group keeps for itself.
5. All nodes perform G-J elimination on the scaled up SLE within the specific modular arithmetic for all assigned module numbers.
6. All nodes send their partial results back to master node.
7. Master node calculates final result from the partial ones.

4.2. Used technology

The background technology allowing the communication among nodes is Message Passing Interface (MPI). The framework can be used in small clusters with only few computers as well as in largest supercomputers with hundreds of thousands nodes. It is widely known and reliable technology. We are using open source implementation OpenMPI [16].

5. GPGPU programming

As mentioned above, the GPU is usually formed by hundreds or thousands of small computation cores. Compared to the modern CPUs' cores they are very simple and small. Therefore one GPU chip can contain so many of them.

A theoretical peak performance of modern GPUs is often more than magnitude higher than the computation performance provided by modern CPUs. But unlike development for today's modern CPUs is GPGPU programming very sensitive for careful design of the program, utilization of computing sources and synchronization.

5.1. Massively parallel architecture

In our research we have focused on GPGPU provided by AMD [15]. Unlike more widespread Nvidia's CUDA technology is general-purpose computing in AMD based on OpenCL framework [10] that represents really universal solution supported by many other vendors including Intel, Apple, Qualcomm, Xilinx and many others. The key advantage of OpenCL technology is its open design available for everybody. It is a general standard for high performance computing that can be used not only in GPGPU area. Applications based on this technology can therefore operate on different devices provided by different vendors.

On figure 5.1 we can see a base block of modern AMD GPU a Compute Unit (CU). Compute unit is a symmetrical multiprocessor containing 64 processing elements (i.e. computing cores) divided into 4 independent SIMD units. There are also other blocks like for example scalar unit, branch predictor or special load / store units. One CU can process 4 independent blocks of compute threads at the time while many others can wait for data store or data load from the main GPU memory.

Latest AMD GPUs contain up to 64 CUs. Our testing GPU (Radeon HD 8750M) was a mobile type equipped with 6 CUs (384 processing elements). It represents bottom line of available GPUs.

5.2. Parallel design of our solver (single node)

As mentioned before the key point during GPGPU development is an optimal handling with available GPU sources. Tasks that are processed on a GPU are called kernels and they are dispatched from CPU according a host application plan. There are many aspects that have to be satisfied in order to get a significant part of potential performance like data hazards or synchronization. Also it is necessary to keep the GPU device as much as possible occupied by computing because latency between main GPU memory and individual processing elements is extensive. Otherwise the whole solution can be even slower than on common CPU.

Logically not all tasks are suitable for GPGPU processing which is optimal for tasks that can be processed massively in parallel. The solving of SLEs is such type of task and that is why we chose the GPGPU approach for its acceleration. On figure 5.2 we can see a heart of our application - the elimination kernel which represents the key function that process the Guass-Jordan elimination on individual SLEs.

The kernel is providing several steps that are necessary for complete G-J elimination. At the beginning a bunch (64 - 256) of compute threads (called work-items in OpenCL terminology) is generated and creating a work-group. Workgroup is a term from OpenCL terminology and represents synchronized group of work-items with a common base like a shared memory, synchronization and other aspects.

Then the work-items are assigned to individual input matrix's columns. The matrix represents one SLE that must be processed by the G-J elimination. Every work-item has its own column or even more if the matrix's dimension is higher than number of work-items. In general we incorporated a column based parallelism.

Then comes the first step within the elimination process - a line swapping. If it’s necessary to swap two lines within the matrix every work item will swap own values within its column.

Next step is a calculation of modular inversion where every work-item calculates the inversion for the input value from its column.

In the next step is this value used (by the work-item that calculated it) for an elimination row calculation. This row is after that used for the forward elimination steps when the processing is heading from upper part of matrix down to the bottom of the matrix. It is a standard part of G-J elimination process.

Then kernel load the appropriate elimination row from the global memory into local memory (with low latency) and the elimination step will take a part in opposite direction from bottom to upper parts of the matrix.

All these steps are performed in the described order until the G-J elimination is finished.

A single CU can process four independent work-groups at a time due to the independent SIMD units. That represents a solving of four independent SLEs at once. Each of this work-groups is also working in parallel supported by 16 processing elements within the SIMD unit. It is important to mention that a GPU usually contains more than one CU so it can run concurrently more independent kernels or more instances of the same kernel. Even our relatively weak testing GPU were able to process 6 kernels at once. This is a triple parallelism within single GPU!

6. Results

Final version of the program was not ready during the writing of this paper therefore we are showing results only for a single-node testing that is mentioned below. The preliminary results of multi-node solution are very promising and tending to a linear scalability for small number of nodes

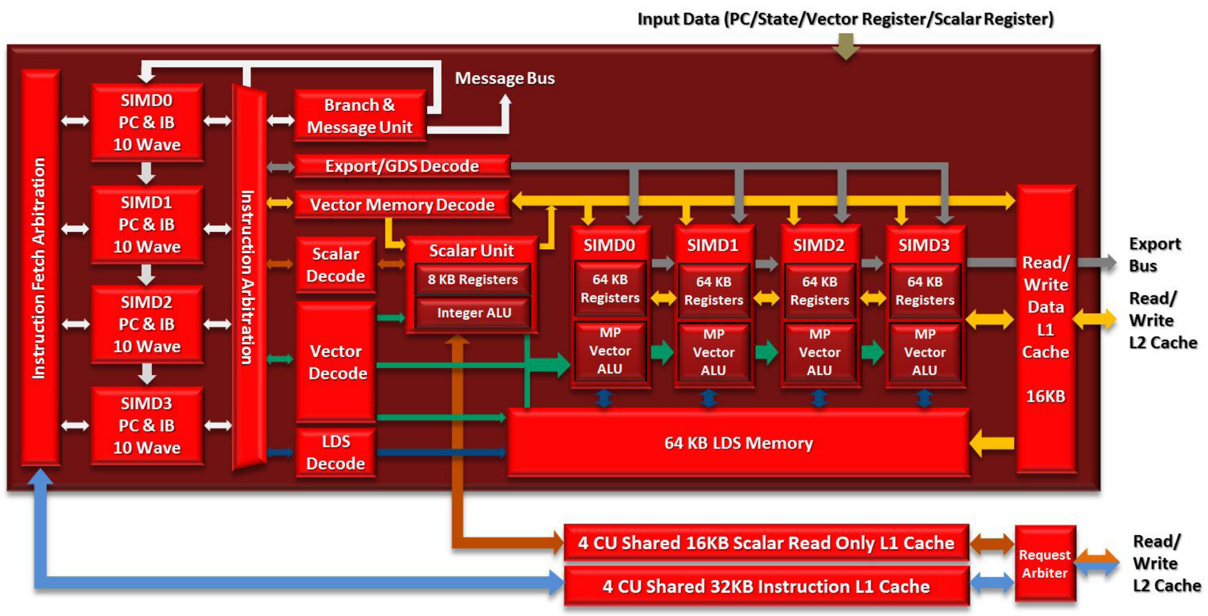

Fig.1. Schematic representation of a Compute Unit (CU) that represents an independent SMP engine containing 64 computing elements capable to handle hunderds of computing threads [9]

but there are still some minor issues that must be optimized. On the other side the single-node results demonstrate well the potential of the multi-node solution and allow us to easily estimate final results.

6.1. Single-node testing

We were testing our solution on Hilbert matrix with wide range of dimensions. It represents ideal testing platform for the precise solving because its ill-conditioned values. Right side of the matrix (the vector **b**, see section 3 for a reference) was appropriately generated in order to have a nonsingular matrix with an existing solution. We have also developed a system that can prove whether the provided solution is absolutely (infinitely) precise. That is very important especially for testing tasks with a higher dimension.

6.2. Testing platform

Like a testing platform we have used a common notebook HP Probook 450 with a following configuration:

- CPU: Intel Core i5 4200M (Haswell), 2× physical computing core, 4× logical computing core, 2,5 - 3,1 GHz (turbo)
- GPU: AMD Radeon 8750M (384 processing elements, 6 Compute Units, 825 MHz), 2GB DDR3 (900 MHz) Memory
- RAM: 8GB DDR3 1600 MHz

We chose these common testing platform on purpose because we wanted to prove whether it is possible to process such computationally intensive task in reasonable time on a standard hardware or even on a notebook that is equipped with appropriate GPU.

6.3. Experiments and results

For testing purposes and comparison we have prepared 3 versions of the program in terms of acceleration of the G-J elimination that is the computational most intensive part. First version is completely sequential and uses only one compute thread running on a CPU. Second version is using OpenMP technology [11] for utilizing all available CPU sources (all cores) via multi-thread execution. This version is thus also parallel like the third one but does not use GPU. Third version is the leading one using OpenCL technology a utilizing GPU.

We gradually tested solving of Hilbert matrices with rising dimensions. We are intentionally showing only results of the G-J elimination itself because that was the only part of the algorithm that differed among the versions and represents the significantly largest time fraction from the whole computation and therefore we can omit the rest of the calculation from the comparison. All mentioned results within table I are calculated like an arithmetic mean from 5 independent experiments. It is important to highlight that for example solving of Hilbert matrix with dimension 1000 usually requires to solve a few thousands of individual SLEs within different modular arithmetic. The elimination time is a time for solving all of them.

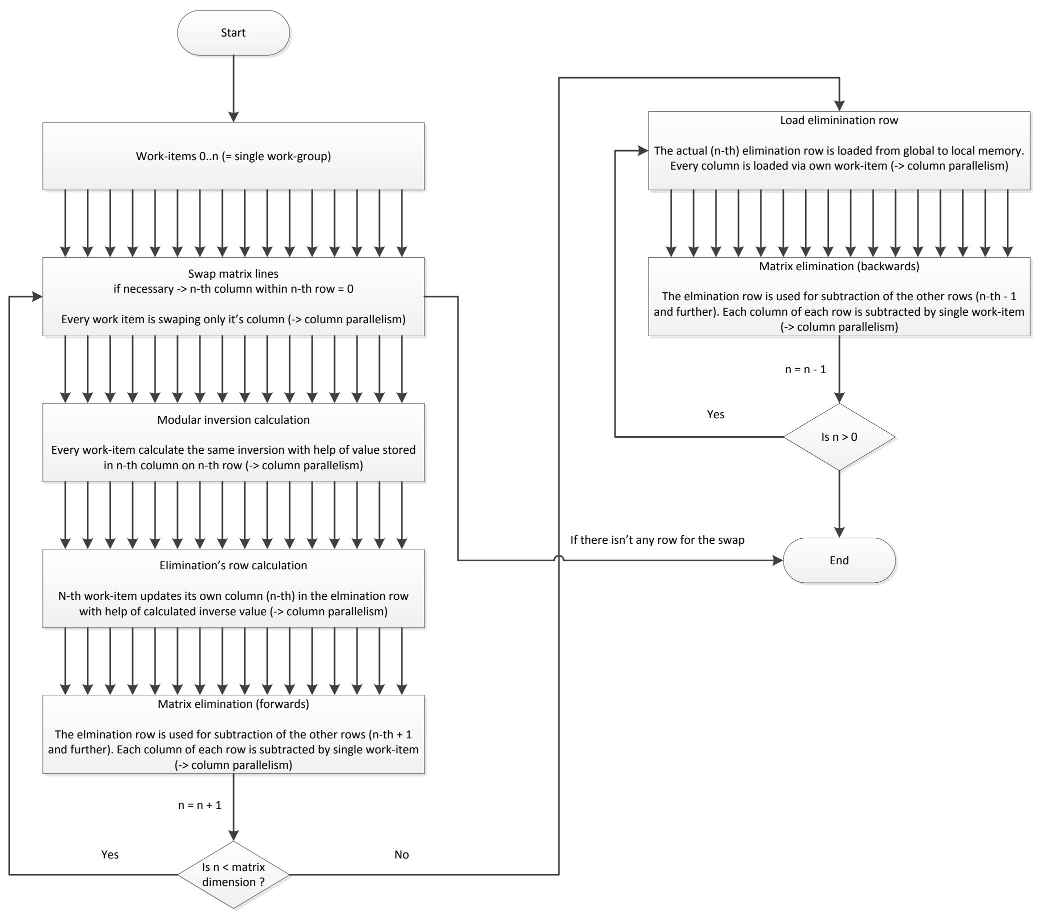
Start
Work-items 0..n (= single work-group)
Swap matrix lines
if necessary -> n-th column within n-th row = 0
Every work item is swaping only it's column (-> column parallelism)
Modular inversion calculation
Every work-item calculate the same inversion with help of value stored in n-th column on n-th row (-> column parallelism)
Elimination's row calculation
N-th work-item updates its own column (n-th) in the elmination row with help of calculated inverse value (-> column parallelism)
Matrix elimination (forwards)
The elmination row is used for subtraction of the other rows (n-th + 1 and further). Each column of each row is subtracted by single work-item (-> column parallelism)
n = n + 1
Yes
Is n < matrix dimension ?
No
Load eliminination row
The actual (n-th) elimination row is loaded from global to local memory. Every column is loaded via own work-item (-> column parallelism)
Matrix elimination (backwards)
The elmination row is used for subtraction of the other rows (n-th - 1 and further). Each column of each row is subtracted by single work-item (-> column parallelism)
n = n - 1
Yes
Is n > 0
If there isn't any row for the swap
End

Fig.2. This diagram represents a key part in the SLEs solver - function that are processing Gauss-Jordan elimination on individual SLEs in a GPU

The table contains calculations of achieved acceleration that provided OpenCL (GPU) version of the program against multi-threaded OpenMP (CPU) version. We can see that for matrices with dimensions equal to 300 and more is acceleration achieved by GPU very significant and at least 4 times higher than by fully utilized CPU (OpenMP). That is a very good result in situation when we are actually using a low-end GPU.

A little degradation of GPU results for matrices with dimensions equal to 750 or 1000 is caused by small amount of GPU memory when data must be transfered gradually into a GPU and back. It is delaying the whole calculation because data transfer via PCI-Express bus is significantly slower then the rest of the subsystems.

Best performance was achieved for matrices with dimensions equal to 500 when GPU processing provided even 6 times better performance than the OpenMP version.

Dim. (n)	Seq. (s)	OpMP (s)	OpCL (s)	Spdup
10	0.01	0.01	0.85	0.01
50	0.21	0.17	0.91	0.19
100	2.61	1.65	1.45	1.14
300	176.82	98.60	20.80	4.74
500	1254.12	671.47	111.71	6.01
750	6150.62	2995.96	535.02	5.60
1000	15120.26	7395.07	1596.31	4.63

Tab. 1. Time of G-J elimination and achieved acceleration on single computing node (1 GPU)

7. Conclusion

Our work is presenting a non-standard and effective way how to obtain infinitely precise solution for ill-conditioned floating-point SLEs. The modular arithmetic's approach can be finally used on a common and cheap HW even for large SLEs. With only a single bottom line GPU we were able to achieve about the magnitude higher performance than on today's modern CPU.

The usage of a GPU cluster is pushing the whole solution into higher performance level. Even preliminary results show us that the scalability of the algorithm is almost linear for an adequate number of computing nodes. Large SLEs can be solved with the infinite precision in several seconds.

There are still performance bottlenecks within this approach like for example necessity to use GMP/MPIR library for final calculations on CPU side or an overhead necessary for GPU processing. But it is obvious that the GPU acceleration worths it.

Acknowledgements

This research has been supported by CTU internal grant SGS No. SGS16/122/OHK3/1T/18.

References

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

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

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

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

[5] Hladík, J., Lórencz, R., Šimeček, I.: *Clock Math - System for Solving SLEs Exactly*. Acta Polytechnica, Vol. 53, No. 2, pages 70-74, 2013

[6] Hickey, T., Ju, Q., van Emden, M.H.: *Interval Arithmetic: from Principles to Implementation*. http://fab.cba.mit.edu/classes/S62.12/docs/Hickey_interval.pdf

[7] The GNU Multiple Precision Arithmetic Library (GMP), https://gmplib.org/

[8] Multiple Precision Integers and Rationals (MPIR), http://mpir.org/

[9] AMD Graphics Core Next (GCN) Architecture - White Paper, https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

[10] Khronos Group - OpenCL, https://www.khronos.org/opencl/

[11] The OpenMP API specification for parallel programming, http://openmp.org/wp/openmp-specifications/

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

[13] SCD Supercomputer Gallery: CRAY 1-A: 1977 - 1989, https://www.cisl.ucar.edu/computers/gallery/cray/cray1.jsp

[14] *NVIDIA GeForce 8800 GPU Architecture Overview*. Technical Brief, 2006, http://www.nvidia.com/page/8800_tech_briefs.html

[15] AMD Developer Central: OpenCL^{TM} Zone, http://developer.amd.com/tools-and-sdks/opencl-zone/

[16] Open MPI: Open Source High Performance Computing, http://www.open-mpi.org/