Modular Arithmetic for Solving Linear Equations on the GPU

J. Hladík, I. Šimeček

Faculty of Information Technology, Prague
Czech Technical University in Prague

1 Introduction

The linear algebraic equations solution is quite a frequent task within numerical mathematics. One might often find problems while solving problems of the ill-conditioned matrix. The solution stability cannot be ensured for large dense sets of linear equations. Rounding error during the numerical computation cannot be tolerated. There are methods developed that minimize the influence of rounding errors on the solution.

The one method I am using relies on modular arithmetic [2] to solve dense systems of linear equations precisely. The idea behind is sounds quite simple – bypass floating point rounding limitations using integer arithmetic. It consist of three parts – converting floating point numbers into integers, solving system of linear equations and finally converting back.

In this paper I propose a GPU-running system that solves systems of linear equations on the GPU. Modern GPU hardware is capable of accelerating data-parallel algorithms [5] so we can expect a huge speedup compared to CPU implementations.

2 Mathematical Background

Let us have a system of linear equations to solve:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1}$$

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 desired vector – solution of our system of linear equations.

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

The question now is how to determine the scaling constant. First, by finding the smallest absolute value element (closest to zero) of the row. Then extracting the absolute value of its exponent and multiplying it by the 2^{24} constant[1]. Finally the scaling constant s is then computed:

$$s = 2^{24} * 2^{|\texttt{exp}(min_{row})|}, \tag{2}$$

where `exp` is a function that extracts an exponent (as an integer, power of 2) and min_{row} is the row's element closes to zero. An approach I used is explained with more details (and with alternatives) in [2].

[1]24 bits – a significant mantissa precision in the IEEE 754 floating point number format [9]

System of Linear Equations Solution Here I am using an approach described with details in [1]. The *Gauss-Jordan* elimination. It is an algorithm for getting matrices in reduced row echelon form using elementary row operations. It is a variation of a *Gaussian* elimination. *Gauss-Jordan* goes further by placing zeros below and above each pivot – these matrices are said to be in reduced row echelon form.

I am using library functions [10] here to perform parts (elementary row operations) of the *Gauss-Jordan* elimination on the matrix of integers. More details in section 3.2.

3 GPU Implementation and Optimization

GPGPU[2] is the area of my research, hence I will be optimizing computations on PC graphical hardware. There are several platforms to develop on. NVIDIA CUDA [6], but that is a proprietary one with usage limited to NVIDIA hardware. Microsoft DirectX DirectCompute [8] is bound to Microsoft platform only. I use OpenCL technology, that is an open standard and works well across platforms (GNU/Linux, Microsoft Windows and Apple Mac) and on all latest graphical hardware (NVIDIA, AMD and Intel GPUs).

3.1 Matrix Scaling

The goal here is to find the absolute value of the smallest (closest to zero) vector element. Than extract it's exponent to compute the scaling coefficient and scale up.

1. Each streaming multiprocessor (SM) is assigned to load a single row of our system of linear equations. All SM cores then find smallest element in their corresponding vector parts.
2. Parallel prefix sum (PPS) algorithm will then be used get smallest element of one SM to the first core that will compute the row scaling coefficient.
3. Scale ratio is computed using the following formula:

$$\texttt{scale} = \texttt{ldexp}(1, 24 + e_{min}), \tag{3}$$

   where scale is the desired scale factor and e_{min} is exponent of the smallest row element extracted using C math function `T frexp(T x, int *e)` from the floating point representation. Constant 24 is the number of significant bits used to store mantissa within an IEEE 754 floating point standard. `T ldexp(T x, int n)` is a C math function that does $x * 2^n$ computation.
4. Last step is to scale row up to integers using a vector-scalar multiplication.

3.2 System of Linear Equations Solution

This is the crucial part of solution. Integer-scaled linear system has to be solved on the GPU using the *Gauss-Jordan* algorithm. The algorithm itself has to be split into trivial algebraic routines such as vector-scalar multiplication and vector addition.

I am relying here on the ViennaCL [10] framework that provides not only an above-OpenCL abstraction layer, but also generates hardware-optimized OpenCL kernels to be executed in

[2]GPGPU – General-Purpose computation on Graphics Processing Units

parallel on the GPU device. This framework does feature BLAS[3] 1-3 level support for GPU-accelerated programming in a friendly way.

4 Conclusion

In this paper I briefly described a system for solving systems of linear equations using modular arithmetic capable of running on the GPU. Modular arithmetic is an elegant way of bypassing floating point rounding errors during computations. GPGPGU is the way to go for accelerating data-parallel algorithms using massively parallel processors.

5 Future Work

First thing to do is to finish GPU-running implementation with basic optimizations I proposed in this paper. And do benchmark against the CPU-running implementation. As the next step I will apply more GPU optimizations, especially on the *Gauss-Jorgan* algorithm. Later then I would like to apply multi modular arithmetic to solve real world problems.

Acknowledgement: This research has been supported by following:

- CESNET Development Fund project 390/2010
- SGS SGS10/276/OHK3/3T/13 CTU Prague

Reference

[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, 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] NVIDIA CUDA – NVIDIA Corporation:
`http://developer.nvidia.com/category/zone/cuda-zone`. 2011

[7] AMD APP – AMD Inc.:
`http://developer.amd.com/pages/default.aspx`. 2011

[8] DirectX 11 DirectCompute – Microsoft Corporation:
`http://www.microsoft.com/download/en/details.aspx`. 2011

[3]BLAS – Basic Linear Algebra Subprograms, application interface programming standard

[9] IEEE Standard for Floating-Point Arithmetic (IEEE 754):
`http://en.wikipedia.org/wiki/IEEE_754-2008`. 2008

[10] ViennaCL – Institute for Microelectronics, TU Wien:
`http://viennacl.sourceforge.net/`. 2011