Implementation and performance analysis of the Simplex algorithm adapted to run on commodity OpenCL enabled graphics processors

Adis Hamzić
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
ah15028@etf.unsa.ba

Alvin Huseinović
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
ahuseinovic@etf.unsa.ba

Novica Nosović
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
nnosovic@etf.unsa.ba

***Abstract*—The Simplex algorithm is commonly used for solving Linear Optimization problems. Linear Optimization methods are used to solve problems in areas such as Economics, Business, Planning and Engineering. Developments of hardware platforms have allowed the use of Linear Optimization methods on problems that presented serious computational challenges in the past. However, solving large optimization problems can be time consuming, which has to be taken into consideration for time-critical applications. With the invention of the GPU assisted computing the situation in this field has progressed. In this paper, implementation and performance analysis of the Simplex algorithm adapted to take the advantage of modern graphics processors versus traditional CPU adapted implementation is presented.**

Linear optimization; simplex; GPU; OpenCL; linear programming;

I. Introduction

Linear Optimization problems can be found in many areas of commerce, business, engineering etc. Some of the problems encountered can have thousands of variables and require large computational resources to solve. This can present a challenge to the application of adequate solving techniques in these areas. In this paper we present a method of finding solutions for problems of Linear Optimization using Graphical Processing Units—massively parallel processors that can be found most modern computers. The method presented allows the computation of solutions for models with several thousands of variables while achieving large speedups compared to the same CPU solving methods.

A. GPU

Graphical Processing Units (GPUs) are devices designed to render and display 2D and 3D graphics on computer screens. Through market demand, those devices were continually improved by the manufacturers. The users wanted to see richer, more realistic worlds, and the manufacturers strived to achieve this goal. They managed to achieve it by developing new hardware designs and techniques and always trying to out beat themselves and each other. The result of this "arms race" in technology is what we know today as the modern GPU.

The modern GPU devices contain a large set of processors that can work in parallel. These processors can execute thousands of threads at the same time earning the modern GPU the title of a massively parallel processor.

Up until recently the only way to interact with these devices was through graphics libraries such as OpenGL and Direct3D. When developers realized the power of parallel computing that lies within the GPUs, they started using these libraries to do general computing such as cracking codes or running complex algorithms. This gave rise to the term General Purpose GPU (GPGPU).

Manufacturers of the devices saw an opportunity to introduce a better programming model to the community that would allow the programmers easier, less restricted access to the GPU. The first such framework was NVidia's CUDA framework released in 2007. The CUDA framework allowed *C* and *C++* programmers to access the GPU in an easier manner. Rather than writing image processing code for graphics libraries, they could now write a special subset of *C* code that would get executed on the GPU. Shortly thereafter ATI developed a similar framework called ATI Stream.

In 2008 Apple Inc. created a portable API that could access both NVidia, as well as ATI GPU devices. The API would also act as an abstraction layer and could even run GPU programs on the CPU if no GPU was available. The API was named OpenCL and was later taken into custody by the Khronos Group, the maintainers of the OpenGL API. This paper uses the OpenCL API to implement the proposed methods as it is an open specification that is not bound to a single implementation or hardware manufacturer.

II. Related Work

There have been several previous attempts to use the computational power of GPU hardware for Linear Optimization. As noted, before the introduction of GPGPU frameworks, programmers had to use graphics libraries to access the GPU. Therefore, the earliest attempts used these libraries to implement the algorithms. The procedures were written as image processing programs and images were used as inputs, buffers and outputs. One of the first such methods [1] uses a revised Simplex method as the solver, while the

978-1-4577-0746-9/11/$26.00 ©2011 IEEE

other [2] uses an Interior point method. The problem with these implementations is that graphics libraries don't allow nearly as much control over the GPU as GPGPU frameworks. Therefore, arbitrary limitations that make sense in the graphics context plague the implementations (such as the severely limited input size). Also, as the graphics libraries are specially tuned for graphical processing, additional overhead is observed during calculation and programming [3].

Later, a method for using CUDA to solve LCP problems for use in physics simulations was published by NVidia Corporation [4]. While this method does use CUDA, it is limited to solving LCP problems which are a special case of quadratic programming.

A method based on [1] that used the CUDA framework was presented in [5]. The method also uses a revised Simplex method, but implements it without the limitations of a graphics library. Still, the speedup achieved is only marginal, reaching at most 2.5 for a model with 2000 variables.

Another method of interest uses FPGA hardware to implement the Interior point method. On the FPGA board, dedicated parallel hardware is constructed to implement parts of the algorithm.

III. Linear Optimization

Linear Optimization is a technique used to find the optimal (minimal or maximal extreme) value of a linear polynomial subject to linear constraints. Linear Optimization is favored as it is relatively easy to create linear models that can be realistic enough to provide an accurate view of real-life problems. As such, Linear Optimization has found applications in many areas not limited to engineering, management, logistics, statistics, pattern recognition, etc. It has also found uses in many software problems such as GUI layout [6].

A linear model in Linear Optimization is represented by a linear polynomial whose value is optimized (the goal function, objective function), and a set of linear equations serving as constraints on the variables that compose the objective function. Linear Optimization falls within Convex Optimization Theory, as linear model constraints represent a convex multi-dimensional polytope, whereas the objective function represents a plane in the problem dimension. The goal of Linear Optimization is to find the extreme point (or points) where the plane and the polytope intersect when moving the plane along its normal.

A linear model can be expressed in canonical form like this:

$$\begin{aligned} &\boldsymbol{maximize} && c^T \times x \\ &\boldsymbol{subject\ to} && A \times x \leq b \\ &\boldsymbol{with} && x \geq 0 \end{aligned}$$

where x is a vector of variables forming the model, c is the vector of coefficients of the objective function, A is the matrix of coefficients forming the constraints and b is the vector of the constant terms of the constraints. The canonical form mandates that all variables be non-negative.

Any other variation of a linear model can easily be turned into the canon form by multiple means, such as, but not limited to:

- Multiplying the objective function of a minimization model with -1
- Adding slack variables to the problem
- Adding surplus variables to the problem
- Splitting variables into a positive and a negative part

A. Simplex Method

Simplex is one of the Linear Optimization methods used to optimize linear models. The method was formulated by George Dantzig [7] in 1947, and up until recently it was the method of choice due to its simplicity, robustness and efficiency. In later years, two additional methods were proposed, the first being developed by Khachiyan [8] in 1972, and the second by Karmarkar [9] in 1984. Only the second method has proven itself as being superior to the Simplex method in robustness and efficiency, albeit greatly sacrificing simplicity. This method was later to be known as the first of the so-called Interior point methods. Through the years many more were developed, but they all follow the same basic principle.

The Simplex method operates on linear models expressed in the so-called standard form:

$$\begin{aligned} &\boldsymbol{maximize} && c^T \times x \\ &\boldsymbol{subject\ to} && A \times x = b \\ &\boldsymbol{with} && x \geq 0,\ b \geq 0 \end{aligned}$$

Just as with the canonical linear model form, it is easy to convert other models into the standard form using the mentioned means. To convert a model from the canonical form to the standard form it is enough to:

- Multiply every constraint with a negative right-hand side with -1
- Convert inequalities to equalities by adding or subtracting additional variables

Given a standard form model, it is then possible to construct a tableau that will be used by the method. The

x_b	b	x	x_s
	b	A	
		C	

Fig. 1. Initial Simplex tableau of a linear model. In the tableau, the values below x_s form an identity matrix.

Simplex tableau is a matrix containing the coefficients of the linear model objective function and constraints. An example of a tableau can be seen on Fig. 1. The x_b column contains the current base variables, that is, variables for which the solution has been found. All variables not in the base are equal to zero.

When forming the initial tableau, the base is filled with slack variables. As slack variables don't contribute to the objective function, the initial solution for the problem is always zero.

- ***find*** *the entering variable index* i
  - ***find*** $i = max_{index}(c) \mid c_i > 0$
  - ***if*** *no* $c_i > 0$ *we are done*
- ***find*** *the exiting variable index* j
  - ***compute*** $t = \frac{b}{a_{column\ i}}$
  - ***find*** $j = min_{index}(t) \mid t_j \geq 0$
  - ***if*** *no* $t_j \geq 0$, *there is no solution*
- ***update*** *the tableau*
  - ***for*** *every element* m *in the tableau*
    - $m_{kl} = m_{kl} - \frac{m_{jl}\, m_{ki}}{m_{ji}} \mid k \neq j$
    - $m_{jk} = \frac{m_{jk}}{m_{ji}}$

Fig. 2. Pseudo-code implementation of the Simplex method.

The simplex method operates by improving the current solution. It does that by removing low-contributing variables from the base and adding higher-contributing ones. By doing this, the solution essentially jumps from one vertex of the domain polytope to the next, much like a hill-climbing algorithm. The method stops when the optimal solution is found. The pseudo-code for the method is shown in Fig. 2.

B. Dual Method

To understand what the Dual Simplex method is one must first understand the duality theory of Linear Optimization [10].

The constraints of a linear model are a set of equations defining a feasible region. If the constraints are inconsistent so that the feasible region is empty, the model is called infeasible. If the region is not empty, the problem is called feasible. In the case of a feasible problem, there are two possibilities: the objective function is either bounded or unbounded in the feasible region. The duality theory states that every Linear Optimization model, referred to as the primal, has a counterpart model called the dual. If one of the models is bounded, then the other is too. The solutions of both are identical.

The duality theory allows us to convert a dual feasible model to a primal feasible problem, and solve it using the standard Simplex method.

The standard Simplex method requires all right hand sides (the constant terms of the constraints) to be non-negative. When some of the right hand elements are negative, the model is primal infeasible. Such a model can be converted to the standard form by adding constraints or variables. The dual theory, however, allows the conversion of the model to a primal feasible one without adding new variables or constraints, thereby allowing it to be solved easier. This is done by using the dual Simplex method.

The dual Simplex method pushes out all negative right hand sides until the problem is primal feasible. After all the right hand sides are non-negative, the standard Simplex method can be used to compute the solution.

The pseudo-code for the method is shown in Fig. 3.

- ***find*** *the exiting variable index* i
  - ***find*** $i = min_{index}(b) \mid b_i < 0$
  - ***if*** *no* $b_i < 0$ *we are done*
- ***find*** *the entering variable index* j
  - ***compute*** $t = \left|\frac{c}{a_{row\ i}}\right|$
  - ***find*** $j = min_{index}(t) \mid a_{ij} \leq 0$
  - ***if*** *no* j *satisfies, there is no solution*
- ***update*** *the tableau*
  - ***for*** *every element* m *in the tableau*
    - $m_{kl} = m_{kl} - \frac{m_{il}\, m_{kj}}{m_{ij}} \mid k \neq i$
    - $m_{ik} = \frac{m_{ik}}{m_{ij}}$

Fig. 3. Pseudo-code implementation of the dual Simplex method.

IV. CPU IMPLEMENTATION

As the results of the proposed GPU method need to be analyzed, a reference implementation that uses a similar method is needed. For that purpose, an implementation for the CPU is created. The implementation is written in *C++* and implements both the dual Simplex method, as well as the standard Simplex method.

Both algorithms use the same tableau for the calculations performed. The tableau is a large array of floating point values that is used like a matrix by calculating the index stride. The size of the array is $s = (c + 1)(v + 1) - 1$, where c is the number of constraints, and v is the number of variables of the model. This layout is very space-efficient as no values are wasted to padding.

The base variable column is not placed inside the matrix as the nature of floating point values is not well suited for this application. Instead, an additional array is used containing indices of base variables. The size of the array is

- ***while*** *not* $dualDone()$
  - ***if*** *not* $performDualIteration()$
    - ***return*** *no solution*
- ***while*** *not* $standardDone()$
  - ***if*** *not* $performStandardIteration()$
    - ***return*** *no solution*
- ***return*** *found solution*

Fig. 4. Pseudo-code implementation of the linear model solver.

equal to the number of constraints. The total number of memory used for representing the model is therefore $m = 4(v + c(2 + v))$ bytes.

The implementation has been thoroughly tuned to give an accurate depiction of the performance of the algorithm on the platform. One example of an optimization is the iteration of the tableau. To reduce the number of operations needed to access an element inside the tableau array, a series of iterators was created. These iterators allow sequential access to general rows and columns, as well as the well-known rows and columns of the tableau. By utilizing the iterator to the previous element in the sequence, index calculation was replaced by simple pointer arithmetic.

The procedure for finding a solution of a linear model is to first check if the dual Simplex method can be applied to the model to make it primal feasible. If this is possible, iterations of the method are performed until the either model becomes primal feasible, or an infeasibility condition is reached. After the dual method can no longer be applied, and the model is still feasible, the standard Simplex method iterations can be applied. The procedure is finished after a solution is found, or it is established that none can be found.

The pseudo-code of the procedure can be seen in Fig. 4.

V. GPU Implementation

To be able to compare the GPU to the CPU implementation, the GPU version will use the same basic algorithms, but will implement them differently. When writing an implementation of an algorithm for a massively parallel processor, one must dissect the algorithm and try to spot every chance to take advantage of the parallel nature of the platform. In other words, one must try to eliminate as much of the sequential nature of an algorithm as possible. This is important as sequential code is not only not faster on a parallel platform, it is often times much slower.

If one analyzes the described standard Simplex and dual Simplex methods, one can notice that most of the steps fall in one of these two categories:

- Finding the index of the minimum or maximum element or composite element of a row or a column
- Updating the tableau given the index of the entering and the leaving variable

Both of these problems can be implemented in a parallel manner. The process by which an aggregate value of a set is found in parallel is called reduction. Reduction and the parallel implementation of the tableau update are explained in the next two subchapters.

The first part of the workflow of a GPU algorithm is the initialization of data in the GPU memory. While the data transfer speed between the GPU memory and the system memory is high, it can only be taken advantage of in large sequential reads and writes. Therefore this step should be done at the beginning of the algorithm, and communication should be kept to the minimum. It is because of this that this implementation allocates all the buffers that will be used during the solving process, at the very beginning of the process, and reuses them all through the runtime.

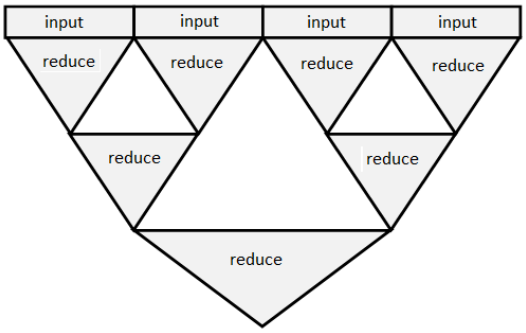

Fig. 5. Visual representation of parallel reduction.

A. Reduction

As mentioned, reduction is the process by which an aggregate value of a set is computed in a parallel manner. The CPU implementation uses sequential search when finding extreme values of sets, as the sets are not sorted. It is not possible to make the CPU sequential code parallel without changing the algorithm, because each iteration depends on the one that happened before it. The best way to make a parallel version of the search is to apply the divide-and-conquer strategy. Instead of computing the aggregate value of the entire set, the set is divided in smaller sets. It is then necessary to compute the aggregate value of each of these sets, combine them into a new set and repeat the process until one single aggregate value is calculated. A visual representation of this process is shown in Fig. 5.

For it to be possible to use reduction with an aggregation relation, the relation must be associative and commutative. Examples of associative and commutative aggregation relations include addition, subtraction, minimal element, maximal element etc.

The actual implementation of parallel reduction in OpenCL has been greatly optimized with the help of [11]. The final version is a two-step process. In the first step a variable-length buffer is aggregated into a fixed length buffer using reduction. The size of the buffer corresponds to the number of processing blocks used. Essentially the only reason for this second-level buffer is to avoid the need for inter-block memory synchronization, as it is generally very

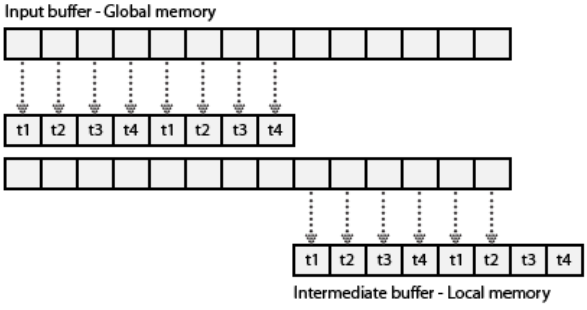

Fig. 6. Sliding aggregation window showing a single processing block. Every thread aggregates two values to the local memory.

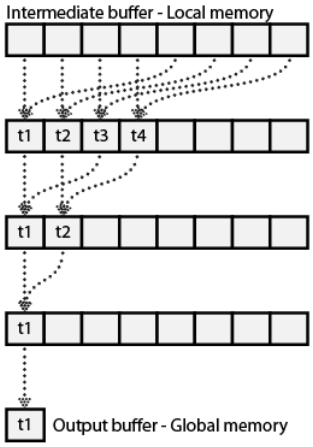

Fig. 7. Reduction of local memory buffers. Only one thread writes the result to the output buffer.

slow. Every block gets its own space in the buffer where it is safe to write without conflicting with other threads.

On the first level a sliding aggregation window is implemented [Fig. 6]. Every thread aggregates two values from the buffer under the current window and then moves the window. This is done till the input buffer is exhausted. The immediate values are held inside the local memory which is visible to one processing block. Once all the values from the input buffer have been aggregated to the local memory, the local memory buffer is itself reduced [Fig 7]. The final value is written to the output buffer.

From the new buffer the final value is extracted using sequential search. The use of the sequential search here makes sense as the buffer is typically so small that using more threads would only incur overhead.

This setup presents us, however, with one problem: when searching for the index of the minimal or maximal element, how can re-reading the global memory on every comparison be prevented. The solution to this problem is to simply keep two buffers in local memory, one for the values and one for the indices. By doing this, the performance difference between searching for the minimal element and searching for the index of the minimal element becomes negligible.

B. Tableau Update

On the first look, the process for updating the tableau looks as if it could be made parallel without any changes. But on closer inspection one can see that that would cause synchronization issues as one thread could write to elements of the tableau before another thread reads them. The best way to solve this is to use two copies of the tableau buffer, a write buffer and a read buffer. After an iteration is done, buffers can be swapped out so the results that were written by the previous iteration become available as the read buffer of the next. Swapping out buffers is really fast as all one has to do is swap out the handles to them.

The algorithm can now be made parallel without the fear of synchronization issues. The algorithm is implemented with a moving update window much like in the reduce algorithm. As the window moves across the read buffer, the relevant indices are calculated and the numbers are fetched from the global memory. The calculation is performed and the result is written to the write buffer.

C. Space Efficiency

As the tableau update step needs two buffers for the tableau storage, the space efficiency of the GPU implementation is already twice as low as that of the CPU implementation. Additionally, the GPU implementation needs reduction buffers for value and index storage, as well as a buffer for the final result retrieval. Fortunately, these buffers are very small.

The final amount of GPU memory needed by this method is $s = 4(2b + v + 2(1 + c)(1 + v))$ bytes, where c is the number of constraints, v is the number of variables and b is the number of blocks used by the reduce algorithm.

VI. Performance Analysis

To measure the speedup of the method and its base parts, three kinds of tests are performed. The first performance test tracks the speed of GPU parallel reduction versus the CPU sequential search algorithms on large sets of data. The second test tracks the speed of the GPU tableau update algorithm versus the CPU equivalent. The final test compares the overall speed of the Simplex solvers.

As the interest of this paper is in the performance benefits of one method versus the other, it is necessary to make sure that both of them are doing the same amount of work. To accomplish this, random data sets are used instead of real linear models, and feasibility checks are removed from the algorithm. Also, the entering and the leaving variable are always calculated, but the calculations are ignored. This is done to make sure that the performed calculations are always the same. As the extreme selection methods use different ordering, different variables can be selected in case of multiple, equal, extreme values.

The tests were run 5 times and the result was calculated as the mean average of the medium three result times, while the best and the worst time were discarded. The tests were run on a machine with an Intel E8400 64 bit processor running at 3.0 GHz with 833MHz DDR2 memory and using an NVidia 9600GT graphics card.

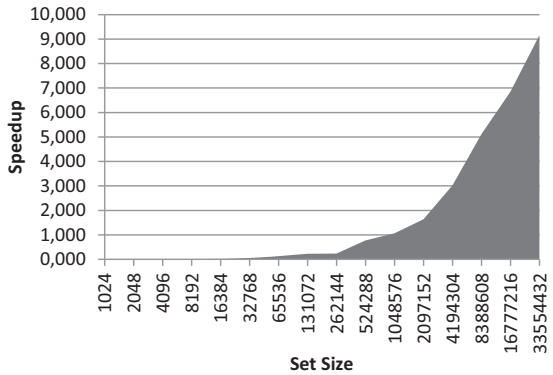

Fig. 8. Parallel reduction time vs. CPU sequential search time.

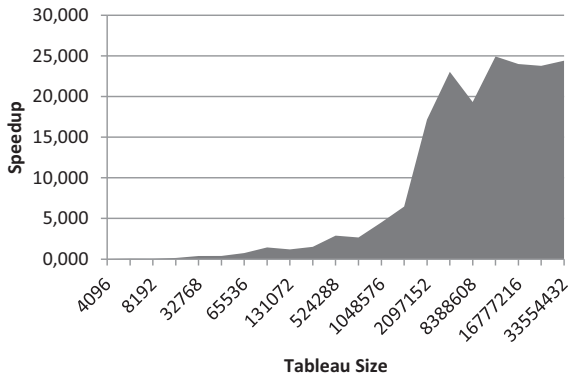

Fig. 9. GPU tableau update speedup in relation to the CPU times.

A. Reduction Performance

The performance of the parallel reduction is compared to the CPU sequential search using sets of variable length. The length starts at 2^{10} and goes to 2^{25}. The result of the comparison can be seen in Fig. 8. The maximum speedup is 9.1 for a set with 2^{25} numbers. For the smaller sets the speedup is also smaller, while for all sets below 2^{20} numbers, there is in fact a slowdown. This is due to the overhead of performing a GPU call (which amounts to about $1ms$ in this case) versus the amount of processing the card does on the data.

Tableau Size	Gpu Time (ms)	Cpu Time (ms)	Speedup
33554432	1907,26	49806,70	26,114
33554432	1908,84	52099,40	27,294
16777216	962,12	25333,10	26,330
8388608	493,44	14275,90	28,931
8388608	487,41	11089,00	22,751
4194304	288,42	5221,17	18,103
2097152	157,89	2710,06	17,165
2097152	144,75	1017,99	7,033
1048576	82,49	383,59	4,650
524288	51,95	137,24	2,642
524288	52,45	198,42	3,783
262144	36,42	70,68	1,941
131072	27,30	29,37	1,076
131072	28,05	38,41	1,369
65536	22,91	15,09	0,659
32768	20,75	6,18	0,298
32768	21,65	6,92	0,320
16384	20,01	1,35	0,068
8192	18,79	1,44	0,077
8192	20,49	1,37	0,07
4096	23,23	0,67	0,03

Fig. 10. Measurement times for the overall performance test.

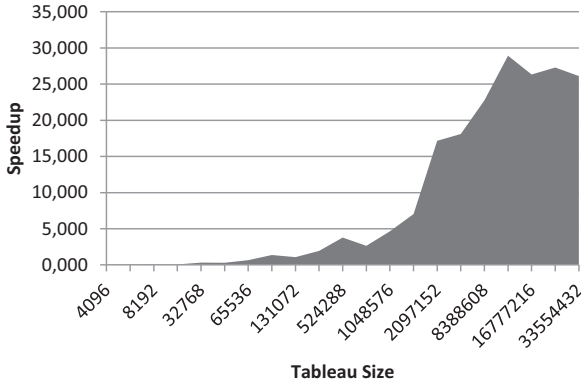

Fig. 11. GPU overall speedup in relation to the CPU times.

B. Tableau Update Performance

The tableau update tests are performed on a random tableau with a size that starts at 2^{12} and goes to 2^{25} elements. The speedup can be seen on Fig. 9. The maximum speedup is close to 25 times while for tableau size below 2^{16} a slowdown is achieved, which can again be ascribed to overhead.

C. Overall Performance

The final test measured the overall speed increase of the GPU algorithm in relation to the CPU algorithm. As the tableau update step is the most time consuming step of the procedures, one can expect that to have most impact on the results. The results were achieved by running 10 iterations of the dual Simplex method and 10 iterations of the standard simplex method. The tableau size was varied, and the maximum tested number of variables was 8192 with 4096 constraints. The opposite case was also tested, with 4096 variables and with 8192 constraints. The results are displayed in Fig. 10 and Fig. 11. The maximum speedup is 28.9 while below the tableau size of 2^{16} a slowdown is achieved.

VII. Conclusion and Future Work

In this paper a method for solving Linear Optimization problems on massively parallel processors was presented. As it was shown, a large performance increase can be achieved by tuning the algorithms to the specific platform and taking advantage of its parallel computation capabilities. The method was able to work on problems with as much as 8192 variables and achieve speedups of up to almost 29 times.

The limiting factor for this method had become the memory usage of a tableau. In the future, this might be resolved by using sparse matrices or multiple matrices of the same size.

Also, while the methods here were carefully tuned, there is no doubt that higher speedups could be achieved if enough effort was given. It remains to be seen how much the current solution can be improved.

REFERENCES

[1] G. Greeff, "The revised simplex algorithm on a GPU," Univ. of Stellenbosch, Tech. Rep., Feb. 2005.

[2] J. H. Jung and D. P. O'Leary, "Implementing an interior point method for linear programs on a CPU-GPU system," Electronic Transaction on Numerical Analysis, vol. 28, pp. 174–189, 2008.

[3] Tien-Tsin Wong, "Shader Programming vs CUDA," June 2008, CIGPU, WCCI 2008, available: http://www.cs.ucl.ac.uk/staff/W.Langdon/cigpu2008/

[4] P. Kipfer, "LCP algorithms for collision detection using CUDA," in GPU Gems 3, H. Nguyen, Ed. Addison-Wesley, 2007, pp. 723–740, http://http.developer.nvidia.com/GPUGems3/gpugems3_ch33.html

[5] D. G. Spampinato and A. C. Elster, "Linear optimization on modern GPUs," in IPDPS. IEEE, 2009, pp. 1–8. [Online], available: http://www.inf.fu-berlin.de/lehre/SS10/SP-Par/download/lp.pdf

[6] Christof Lutteroth, Robert Strandh, Gerald Weber. Optimal GUI Layout as a Problem of Linear Programming. Technical Report UoA-SE-2007-6, The University of Auckland, August 2007.

[7] G.B. Dantzig. Linear Programming and Extensions. Princeton Univ. Press, Princeton, New Jersey, 1963.

[8] L.G. Khachiyan. A polynomial algorithm in linear programming. Doklady Akademiia Nauk SSSR, 244:1093-1096, 1979. Translated into English in Soviet Mathematics Doklady 20, 191-194.

[9] N.K. Karmarkar. A new polynomial-time algorithm for linear programming. Combinatorica, 4:373-395, 1984.

[10] C. Roos, T. Terlaky, J.-Ph. Vial. Interior Point Methods for Linear Optimization. Springer, September 2005, pp. 15

[11] Mark Harris, NVIDIA Developer Technology, Optimizing Parallel Reductiion in CUDA, November 2007, available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf