Simultaneous Solving of Linear Programming Problems in GPU

Amit Gurung*
amitgurung@nitm.ac.in

Binayak Das*
binayak89cse@gmail.com

Rajarshi Ray*
raj.ray84@gmail.com

* National Institute of Technology Meghalaya

Abstract. Linear Programs (LPs) appear in a large number of applications. In this paper, we propose a CUDA implementation for solving a large number of LPs simultaneously in GPU. We show that our implementation has a maximum speed-up of 34.5× for a LP of dimension(200 variables, 200 constraints), running 5000 LPs simultaneously in parallel, when compared to sequential solving using the open source GLPK (GNU Linear Programming Kit) library.

1 Introduction

The use of graphics processing units (GPUs) is becoming widespread in scientific applications for accelerating performance. Some of these application domains include medical image processing[8], weather research and forecasting (WRF) model [5], Proteomics (to speed-up peptide spectrum matching [9]) and large scale graph processing [7]. The use of GPUs for general computing problems other than graphics processing is referred to as General Purpose computing on GPUs (GPGPU). A GPU has a massively parallel architecture consisting of thousands of cores designed for handling multiple tasks simultaneously. NVIDIA provides a programming model called Compute Unified Device Architecture (CUDA) to program GPUs. CUDA extends C/C++ and FORTRAN languages with some special keywords and language primitives.

In this paper, we present a CUDA implementation to solve multiple LPs in parallel using the simplex method[2] in a GPU. Our implementation can solve LP problems of dimension up-to 500×500 (500 variables, 500 constraints). We show that beyond a sufficiently large number of LPs to be solved, our implementation shows significant gain in performance compared to solving them sequentially using the GLPK library, which is an open source LP solver.

To the best of our knowledge, prior work has shown parallel implementation of algorithms like simplex and revised simplex to solve an LP in GPU but to solve a large number of LPs simultaneously and efficiently is not addressed.

We refer the reader to an application which requires solving a large number of LPs simultaneous in the area of reachability analysis in [6], which uses the GLPK library for solving multiple LPs of smaller dimension simultaneously.

2 Related Work

A multi-GPU implementation of the simplex algorithm is reported in [4], which shows a speed-up of 2.93× for LP problem of dimension 1000×1000 and an average speed-up of 12.7× for 8000×8000 dimension or higher. A speed-up of 24.5× is shown when two GPUs are used together as compared to a CPU. An implementation of the revised simplex method using in-built graphics library (OpenGL)

is reported in [1]. The use of texture memory in the implementation limited LPs of dimension 2700×2700 with an average speed-up of $18\times$. The reported speed-up were for problems of size 600×600 or above compared to the GLPK library. Almost all the works report speed-up for large size LP problems (typically of dimension 500×500 or above) compared to the sequential CPU implementations.

Rest of the paper is organized as follows. Section 3 describe the simplex method along with our parallel implementation schemes of the simplex method in GPU. In Section 4, we present our experimental results and conclude in 5.

3 Implementation of Simplex Method for Multiple LPs

A *standard linear programming* problem is maximizing an objective function with a given set of constraints, represented as follows:

$$maximize \qquad \sum_{j=1}^{n} c_j x_j \tag{1}$$

subject to the constraints

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad for \quad i = 1, 2, ..., m \tag{2}$$

and

$$x_j \geq 0 \quad for \quad j = 1, 2, ..., n \tag{3}$$

In Expression (1) , $\sum_{j=1}^{n} c_j x_j$ is the objective function to be maximized and inequality (2) is the given m constraints of n variables. The inequality (3) is the nonnegativity constraints over n variables. A standard LP can be converted into a form, called *slack form* by introducing m additional **slack variables**(x_{n+i}) one for each inequality constraints as shown below:

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j, \ for \ i = 1, ..., m \tag{4}$$

along with the nonnegativity constraints $x_{n+i} \geq 0, \ for \ i = 1, ..., m$.

An algorithm that solves the LP problems efficiently in practice is the *simplex method* described in [2]. The variables on the left-hand side of the equation (4) are referred to as **basic variables** and those on the right-hand side are **nonbasic variables**. For some LPs, the *initial basic solution* may not be feasible. For such LPs, the simplex method employs a two-phase algorithm. A new **auxiliary LP** is formed by replacing with a new objective function z, which is the sum of the newly introduced **artificial variables.** The **simplex algorithm** is employed on this auxiliary LP and checks if the optimal solution to z is 0. If so, the original LP has a feasible solution and the simplex method now initiates for Phase II. Therefore, LPs with infeasible initial basic solution takes more time to be solved.

3.1 The Simplex Algorithm

The simplex algorithm is an iterative process of solving a LP problem. Each iteration tries to increase the value of the objective function by replacing one of the basic variable (also know as the **leaving variable**), by a nonbasic variable (called the **entering variable**). The exchange of these

two variables is obtained by a *pivot operation*. The basic simplex algorithm consists of the following main steps:

The algorithm first formulates a **simplex tableau** from the slack form using the coefficients of the given LP. The tableau is of size $(m+1) \times n$ with m as the number of constraints and n as the sum of the number of variables in the given LP, slack variables, artificial variables (provided if the given LP has infeasible initial basic solution) along with three extra columns. The first of the three columns of the table is used to store the coefficient of the basic variables, the second column is for the values b_i (the bounds of the constraints as in the inequality (2)), and finally the last column stores the computed values, which is used to determine the *pivot row* in the algorithm. The last row of the table is used to compute the value which is used to determine the *pivot column* in step 1 below.

Step 1: Determine the entering variable.

At each iteration the algorithm identifies a new *entering variable* (one of the column index of the simplex table) which can improve the existing objective function into an optimal solution, in the algorithm it is referred to as the *pivot column*.

Step 2: Determine the leaving variable.

Once the pivot column is determined the algorithm next identifies the row with the minimum positive value (which is obtained by dividing the column b_i by the corresponding pivot column) for the *leaving variable* which is termed as the *pivot row* in the simplex algorithm.

Step 3: Obtain the new improved value of the objective function.

The algorithm then performs the *pivot operation* which updates the simplex tableau and obtains new improved value for the objective function.

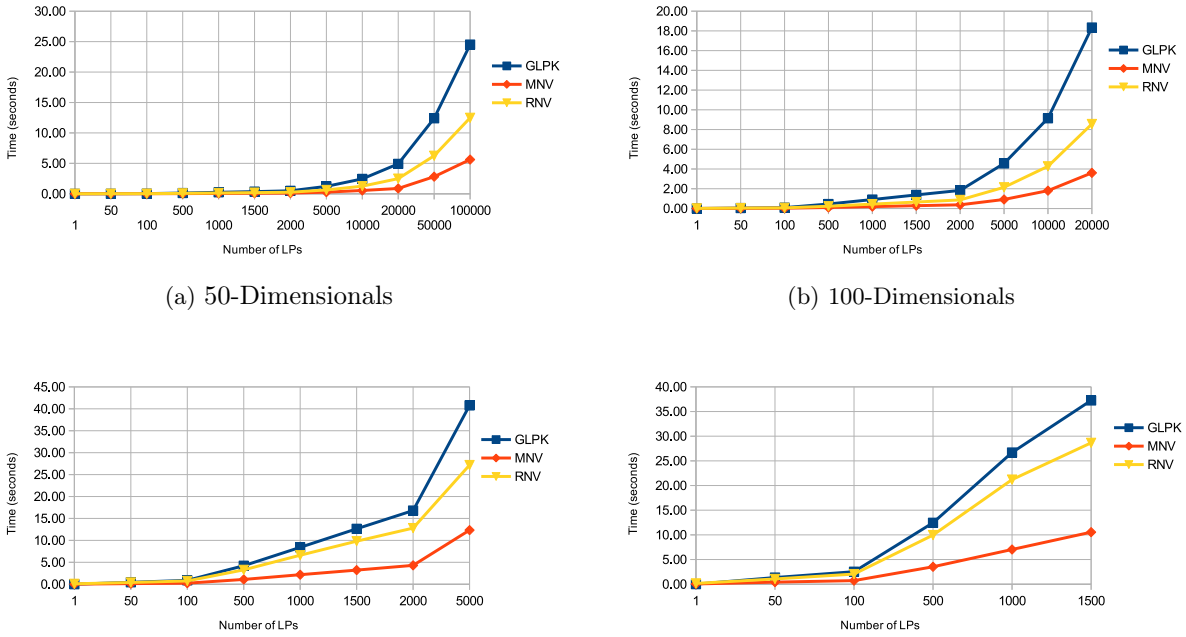
These steps are iterated until the halt condition has reached. The halt condition is met when either the LP is found to be *unbounded* (i.e., no new leaving variable can be computed) or the iterations has found the optimal solution (i.e., no new entering variable can be found).

3.2 Parallel Algorithm for Solving Multiple LPs Simultaneously

We present here our CUDA implementation that solves multiple LPs in parallel on GPU. The simplex tableau described above is populated in the host(CPU) and then it is transferred to the device(GPU). To speed-up populating the table, we use the OpenMP directives. For an efficient kernel launch configuration and to achieve maximum GPU occupancy we map our problem into a two dimensional block with each block ID used to solve an LP. Each block consists of j ($\geq n$, n being the column size of the simplex table) threads, which is a multiple of 32, as instructions in GPU are scheduled and executed as per warp basis where a warp is a collection of 32 threads. We parallelized **step 1** above by utilizing n (out of j)threads in parallel to determine the pivot column using **parallel reduction** described in [3]. A parallel reduction is a technique applied to achieve data parallelism in GPU when a single result is required to be generated out of a large number of data. We also parallelized **step 2** by using parallel reduction technique by using m (out of j)threads in parallel to determine the pivot row. The pivot operation in **step 3** involving only $(m-1)$ rows also employed data parallelism by utilizing $(m-1)$threads (out of j threads) in parallel to perform the required pivot operation.

There are a number of approaches to compute the new **entering variable**, but so far there is no general rule formulated which determines the superiority of one approach over other. In this paper, we have presented only two approaches, to study the effect and the performance of these approaches on different sizes of LPs, when solved simultaneously in parallel. We describe these approaches below:

- **Most Negative Value (MNV):** In this case, we take the minimum negative value computed in the last row of the simplex table in the above algorithm, using parallel reduction technique.
- **Random Negative Value(RNV):** Instead of choosing the minimum negative value as in MNV, here we choose a random negative value from the last row of the simplex table. Which can also be parallelized by using n threads in parallel instead of using parallel reduction technique.



(a) 50-Dimensionals

(b) 100-Dimensionals

(c) 300-Dimensionals

(d) 500-Dimensionals

Figure 1: showing time taken to compute number of LPs for dimension $50, 100, 300$ and 500 respectively for the class of LPs with **initial basic solution as feasible** .

No. Of Lps	No of Streams	5 Dimension			15 Dimension			50 Dimension			100 Dimension			200 Dimension		
		GLPK	MNV	GLPK Vs MNV	GLPK	MNV	GLPK Vs MNV	GLPK	MNV	GLPK Vs MNV	GLPK	MNV	GLPK Vs MNV	GLPK	MNV	GLPK Vs MNV
1	*1*	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.33	0.01	0.00	4.11	0.06	0.01	7.54
50	*1*	0.00	0.00	1.00	0.03	0.00	15.67	0.10	0.01	10.81	0.80	0.03	26.50	3.36	0.11	31.15
100	*1*	0.00	0.00	2.00	0.06	0.00	19.78	0.18	0.02	12.22	1.50	0.05	28.12	6.90	0.21	33.03
500	*10*	0.01	0.01	1.90	0.32	0.01	26.69	0.92	0.07	12.29	7.78	0.25	30.57	33.79	1.04	32.36
1000	*10*	0.02	0.01	2.30	0.61	0.02	25.56	1.84	0.15	12.40	14.98	0.51	29.64	67.61	2.00	33.87
1500	*10*	0.04	0.01	2.50	0.91	0.04	25.69	2.39	0.22	10.89	22.64	0.75	30.02	101.41	2.99	33.90
2000	*10*	0.05	0.02	2.42	1.22	0.05	25.35	3.10	0.29	10.63	30.17	0.99	30.54	135.35	3.98	33.99
5000	*10*	0.12	0.04	2.62	3.04	0.12	25.44	7.73	0.70	10.98	75.44	2.46	30.62	344.68	9.98	34.54
10000	*10*	0.23	0.08	2.84	6.11	0.23	26.72	15.46	1.40	11.06	135.26	4.95	27.34	--	--	--
20000	*10*	0.46	0.17	2.79	12.53	0.43	29.11	30.78	2.79	11.02	--	--	--	--	--	--
50000	*10*	1.16	0.40	2.91	30.56	1.09	28.08	76.51	7.08	10.80	--	--	--	--	--	--
100000	*10*	2.35	0.79	2.97	60.90	2.14	28.46	--	--	--	--	--	--	--	--	--

Table 1: showing comparison between GLPK and GPU implementation for the type of LPs with **initial basic solution as infeasible**.

4 Experiments

We performed our experiment in Intel Q9950, 2.84GHz, 4 Core (without hyper-threading), 8GB RAM with GeForce GTX 670 GPU card for an average of 10 runs. We observed a maximum speed-up of 5.5× using MNV and 2.5× using RNV approach respectively as compared to GLPK for LP problems which has *initial basic solution as feasible* as show in figure 1. We observed that for high dimensional LPs our CUDA implementation performs better even with few (e.g., 50) LPs in parallel. But, for small size LPs, GPU out performs GLPK only after 100 and above LPs.

We gain a maximum speed-up of 34.5× using MNV approach compared to GLPK as shown in table 1 for LP problems with *infeasible initial basic solution.*

5 Conclusions

We presented a parallel implementation to solve multiple LPs simultaneously in GPU. We have also shown the performance gain with two proposed variations in choosing a pivot element in the simplex algorithm. Overall, we have shown significant speed-up when large number of LPs are solved in parallel in GPU compared to sequential solving in GLPK.

References

[1] Jakob Bieling, Patrick Peschlow, and Peter Martini. An efficient GPU implementation of the revised simplex method. *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW 2010*, 2010.

[2] GB Dantzig and MN Thapa. *Linear programming 1: introduction*. Springer Science & Business Media, 2006.

[3] M Harris. Optimizing parallel reduction in CUDA. *NVIDIA Developer Technology*, 2007.

[4] Mohamed Esseghir Lalami, Didier El-Baz, and Vincent Boyer. Multi GPU implementation of the simplex algorithm. *Proc.- 2011 IEEE International Conference on HPCC 2011 - 2011 IEEE International Workshop on FTDCS 2011 -Workshops of the 2011 Int. Conf. on UIC 2011-Workshops of the 2011 Int. Conf. ATC 2011*, pages 179–186, 2011.

[5] John Michalakes and Manish Vachharajani. GPU Acceleration of Numerical Weather Prediction. *Parallel Processing Letter*, pages 1–18, 2008.

[6] Rajarshi Ray and Amit Gurung. Poster : Parallel State Space Exploration of Linear Systems with Inputs using XSpeed. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 285–286. ACM, 2015.

[7] Koichi Shirahata, Hitoshi Sato, Toyotaro Suzumura, and Satoshi Matsuoka. A GPU Implementation of generalized graph processing algorithm GIM-V. *Proceedings - 2012 IEEE International Conference on Cluster Computing Workshops, Cluster Workshops 2012*, pages 207–212, 2012.

[8] Guorui Yan, Jie Tian, Shouping Zhu, Yakang Dai, and Chenghu Qin. Fast cone-beam CT image reconstruction using GPU hardware. *Journal of X-Ray Science and Technology*, 16(4):225–234, 2008.

[9] Jian Zhang, Ian McQuillan, and FangXiang Wu. Speed Improvements of Peptide - Spectrum Matching Using SIMD Instructions. *IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBMW)*, pages 1–22, 2010.