

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/308646528>

Solving Batched Linear Programs on GPU and Multicore CPU

Article · September 2016

CITATION

1

READS

494

2 authors:



[Amit Gurung](#)

Martin Luther Christian University

18 PUBLICATIONS 85 CITATIONS

[SEE PROFILE](#)



[Rajarshi Ray](#)

Indian Association for the Cultivation of Science

31 PUBLICATIONS 1,070 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Digital Image Processing [View project](#)



Formal Methods for Safety-Critical Systems [View project](#)

Solving Batched Linear Programs on GPU and Multicore CPU

Amit Gurung*, Rajarshi Ray

*Department of Computer Science & Engineering, National Institute of Technology
Meghalaya, Shillong - 793003, India*

Abstract

Linear Programs (LPs) appear in a large number of applications and offloading them to the GPU is viable to gain performance. Existing work on offloading and solving an LP on GPU suggests that performance is gained from large sized LPs (typically 500 constraints, 500 variables and above). In order to gain performance from GPU for applications involving small to medium sized LPs, we propose batched solving of a large number of LPs in parallel. In this paper, we present the design and CUDA implementation of our batched LP solver library, keeping memory coalescent access, reduced CPU-GPU memory transfer latency and load balancing as the goals. The performance of the batched LP solver is compared against sequential solving in the CPU using an open source solver GLPK (GNU Linear Programming Kit). The performance is evaluated for three types of LPs. The first type is with the initial basic solution as feasible, the second type is with the initial basic solution as infeasible and the third type is with the feasible region as a Hyperbox. For the first type, we show a maximum speedup of $18.3\times$ when running a batch of $50k$ LPs of size 100 (100 variables, 100 constraints). For the second type, a maximum speedup of $12\times$ is obtained with a batch of $10k$ LPs of size 200. For the third type, we show a significant speedup of $63\times$ in solving a batch of nearly 4 million LPs of size 5 and $34\times$ in solving 6 million LPs of size 28. In addition, we show that the

*Corresponding author

Email addresses: `amitgurung@nitm.ac.in` (Amit Gurung*), `rajarshi.ray@nitm.ac.in` (Rajarshi Ray)

open source library for solving linear programs-GLPK, can be easily extended to solve many LPs in parallel with multi-threading. The thread parallel GLPK implementation runs $9.6\times$ faster in solving a batch of $1e5$ LPs of size 100, on a 12-core Intel Xeon processor. We demonstrate the application of our batched LP solver in the domain of state-space exploration of mathematical models of control systems design.

Keywords: Linear programming, Batched linear programs, GPU, Simplex method, Pivot selection rules, GLPK library

1. Introduction

Computations which were traditionally purely carried out in the CPU are increasingly being computed with CPU and GPU in heterogeneity by offloaded expensive data parallel tasks to a GPU for accelerating performance. Some of the application domains where GPU has been used to accelerate performance include medical image processing [1, 2], weather research and forecasting (WRF) [3], Proteomics (to speed-up peptide spectrum matching [4]), signal processing for radio astronomy[5], simulation of various physical and mechanical systems (using variants of Monte Carlo algorithm)[6, 7] and large scale graph processing [8]. However, gaining performance from a GPU requires insights on its architecture in order to have an effective load balancing, efficient memory access and an effective mapping of computations in the SIMD paradigm of computing.

Linear Programming is a method of maximizing or minimizing a linear objective function subject to a set of linear constraints. Linear programs (LPs) appear extensively in a large number of application domains such as business process modeling to maximize profit, economics to design optimized demand-supply model (for example Leontief Input-Output model [9]), optimal cost and transport assignment in transportation problem [10], optimal job scheduling [11] and optimize packets routing in computer networks, to name just some.

In this work, our focus is on CPU-GPU heterogeneous computations that, in particular, requires solving a large number of LPs. Our work is on the setting

that computations begin in a CPU where LPs are created and then offloaded to a GPU for an accelerated solution. The solutions are transferred back to the CPU from the GPU for further processing. There has been prior work in this direction with parallel implementation of algorithms to solve LPs on a GPU, like the simplex and revised simplex algorithm [12, 13, 14]. However, the performance gain is reported only when offloading large LPs of size 500 (500 constraints, 500 variables) and above. Prior works state that for small size LPs, the time spent in offloading the LPs from CPU to GPU memory is more than the time gained with parallel solution in the GPU. Therefore, how can applications requiring to solve small to medium size LPs exploit the power of a GPU, remains a research challenge.

Our work in this paper target application that involves solving small to medium size LPs, but many of them. The existing work of offloading LPs to GPU does not provide acceleration in such applications due to small-medium size LPs. We therefore propose to use GPUs to solve not a single LP at a time, but to batch them and solve them simultaneously. We show that with batched computation, the performance gain with parallelism is more than the performance loss in transferring LP tasks from CPU-GPU memory, even for small size LPs (e.g. LPs of size 5). We present a CUDA C/C++ implementation of our library which implements the simplex method [15], with an effort to keep coalescent memory accesses, efficient CPU-GPU memory transfer and effective load balancing. To the best of our knowledge, this is the first work in the direction of batched LP solving in the GPU. Batched computations in GPU to draw performance is, however emerging as a technique in general [16, 17]. The library source and necessary instructions for repeatability evaluation can be found at <https://bitbucket.org/rajgurung777/simplexprojects>.

We report solutions of LPs of dimension up to 511×511 (511 variables, 511 constraints) with our library. We show that beyond a sufficiently large batch size, our implementation shows significant gain in performance compared to solving them sequentially in the CPU using the GLPK library [18], an open source LP solver. We also report our observations on two pivot selection rules

in the simplex method implemented in the library. In addition, we present a technique to solve a special class of LP when the feasible region is a hyper-rectangle and show that these can be solved cheaply without using the simplex algorithm. We implement this special case LP solver as part of the library.

Finally, we attempt to address the problem that GLPK implementation is not *thread safe*. By *not thread safe*, we mean that multiple threads running local instances of the GLPK object is not safe. As a solution, we show the necessary changes to make it thread safe and report performance gain with multi-threaded solving of many LPs in a multi-core architecture.

The rest of the paper is organized as follows. Related works are discussed in Section 2. In Section 3, we discuss the simplex method that is needed to appreciate the rest of the paper. Section 4 illustrates our CUDA implementation for solving batched LPs on GPU, with memory coalescence, effective load balancing and efficient GPU-CPU memory transfer using CUDA streams to gain performance. In Section 5, we present the implementation and experimental results of a thread safe GLPK for solving multiple LPs using multi-threading. Section 6 shows the performance of our CUDA implementation for solving a special class of LP problems in batches in comparison to solving the same LPs sequentially with GLPK. In Section 7, we show an application of our batched LP solver GPU library in the domain of model based analysis of control systems design.

2. Related Work

A multi-GPU implementation of the simplex algorithm in [12] reports a speedup of $2.93\times$ on LP problems of dimension 1000×1000 . An average speedup of $12.7\times$ has been reported for the larger problems of dimension 8000×8000 or higher on a single GPU. An implementation of the revised simplex method using inbuilt graphics library (OpenGL) is reported in [13]. An average speedup of $18\times$ has been reported, compared to the GLPK library, for problems of size 600×600 or higher. A GPU implementation of the revised simplex algorithm

is also reported in [14] with a speedup of $2\times$ to $2.5\times$ in comparison to a serial ATLAS-based CPU implementation for LPs of dimension 1400 up to 2000. Automatically Tuned Linear Algebra Software (ATLAS[19]) is a software library for linear algebra providing an implementation of the BLAS (Basic Linear Algebra Subprograms) APIs for C and Fortran. BLAS[20] is a specification that prescribes routines for basic vector and matrix operations. BLAS implementation is optimized for performance on a specific architecture. We observed that almost all the works report speedup only for large size LP problems (typically of dimension 500×500 or above) compared to the sequential CPU implementations.

3. Linear Programming

A linear program in *standard form* is maximizing an objective function under the given set of linear constraints, represented as follows:

$$\text{maximize} \quad \sum_{j=1}^n c_j x_j \quad (1)$$

subject to the constraints

$$\sum_{j=1}^n a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \dots, m \quad (2)$$

and

$$x_j \geq 0 \quad \text{for } j = 1, 2, \dots, n \quad (3)$$

In Expression (1), $\sum_{j=1}^n c_j x_j$ is the objective function to be maximized and Inequality (2) shows the m constraints over n variables. Inequality (3) shows the non-negativity constraints over n variables. An LP in *standard form* can be converted into *slack form* by introducing m additional **slack variables** (x_{n+i}), one for each inequality constraint, to convert it into an equality constraint, as shown below:

$$x_{n+i} = b_i - \sum_{j=1}^n a_{ij} x_j, \quad \text{for } i = 1, \dots, m \quad (4)$$

An algorithm that solves LP problems efficiently in practice is the *simplex method* described in [15]. 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 **non-basic variables**. The *initial basic solution* of an LP is obtained by assigning its non-basic variables to 0. The *initial basic solution* may not be always feasible (when one or more of the b_i s are negative, resulting in the violation of the non-negativity constraint). For such LPs, the simplex method employs a two-phase algorithm. A new **auxiliary LP** is formed by having 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 it is checked if the optimal solution to the objective function is 0. If a zero optimal is found, then the original LP has feasible solution and the simplex method 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 an LP problem. Each iteration of the simplex algorithm attempts to increase the value of the objective function by replacing one of the basic variables (also known as the **leaving variable**), by a non-basic variable (called the **entering variable**). The exchange of these two variables is obtained by a *pivot operation*. The index of the leaving and the entering variables are called the pivot row and pivot column respectively. The simplex algorithm iterates on a tabular representation of the LP, called the **simplex tableau**. The simplex tableau stores the coefficients of the non-basic, slack and artificial variables in its rows. It contains auxiliary columns for storing intermediate computations. For our discussion, we consider a tableau of size $p \times q$, where $p = m + 1$ and $q = n + \text{sum of slack and artificial variables} + 2$. The $(m + 1)$ the row stores, the best solution to the objective function found so far, along with the coefficients of the non-basic variables in the objective function.

There are two auxiliary columns, the first column stores the index of the basic variables and the second stores b_i 's of inequality (2). A schematic of the

Index	b_i	$x_1 \ x_2 \ \dots \ x_n$	$x_{n+1} \ x_{n+2} \ \dots \ x_{m+n}$	$a_1 \ a_2 \ \dots \ a_s$
Index of basic variables	Bound value of the constraints	Coefficients of non-basic variable	Coefficients of slack variable	Coefficients of artificial variable
unused	Optimal Solution	Coefficients of non-basic variable in objective function (used to determine entering variable)		

Figure 1: Formation of the Simplex Tableau.

simplex tableau is shown in Figure 1.

Step 1: Determine the entering variable.

At each iteration, the algorithm identifies a new *entering variable* from the non-basic variables. It is called an entering variable since it enters the set of basic variables. The choice of the entering variable is with the goal that increasing its value from 0 increases the objective function value. The index of the entering variable is referred to as the *pivot column*. The most common rule for selecting an entering variable is by choosing the index e of the maximum in the last row of the simplex tableau (excluding the current optimal solution).

Step 2: Determine the leaving variable.

Once the pivot column is determined (say e), the algorithm identifies the row index with the minimum positive ratio $(b_i / -a_{e,i})$, say ℓ , called the *pivot row*. The variable x_ℓ is called the leaving variable because it leaves the set of basic variables. This ratio represents the extend to which the entering variable x_e (in step 1) can be increased without violating the constraints.

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

The algorithm then performs the *pivot operation* which updates the simplex tableau such that the new set of basic variables are expressed as a linear combination of the non-basic ones, using substitution and rewriting. An improved value for the objective function is found after the pivot operation.

The above steps are iterated until the halt condition is reached. The halt condition is met when either the LP is found to be *unbounded* or the **optimal**

solution is found. An LP is unbounded when no new leaving variable can be computed, i.e. when the ratio $(b_i / -a_{e,i})$ in step 2 is either negative or undefined for all i . An optimal solution is obtained when no new entering variable can be found, i.e., the coefficients of the non-basic variables in the last row of the tableau are all negative values

4. Simultaneous Solving of Batched LPs on GPU

We present our CUDA implementation that solves fixed size batched LPs in parallel on a GPU. In this discussion, we shall refer a CPU by *host* and a GPU by *device*.

4.1. CPU-GPU Memory Transfer and Load balancing

First, we allocate device memory (global memory) from the host, that is required for creating a simplex tableau for every LP in the batch. The maximum number of LPs that can be batched depends on the size of the device global memory. The tableau for every LP in the batch is populated with all the coefficients and indices of the variables in the host side, before transferring to the device. To speedup populating the tableau in the host, we initialize the tableau in parallel using OpenMP threads. Once initialized, the simplex tableaux are copied from the host to the device memory (referred to as H2D-ST in Figure 7). The GPU kernel modifies the tableaux to obtain solution using the simplex method and the results for every LP in the batch is copied back from the device to the host memory (referred to as D2H-res in Figure 7). We discuss further on our CPU-GPU memory transfer using CUDA streams for efficiency in Section 4.4.

Load Balancing. We assign a CUDA block of threads to solve an LP in the batch. Since blocks are scheduled to Streaming Multiprocessors (SMs), this ensures that all SMs are busy when there are sufficiently large number of LPs to be solved in the batch. As CUDA blocks execute asynchronously, such a task division emulates solving many LPs independently in parallel. Moreover, each

block is made to consist of j ($\geq q$) threads, which is a multiple of 32, as threads in GPU are scheduled and executed as warps. The block of threads is utilized in manipulating the simplex tableau in parallel, introducing another level of parallelism. In Figure 2, we show a block diagram of our parallel implementation on the GPU.

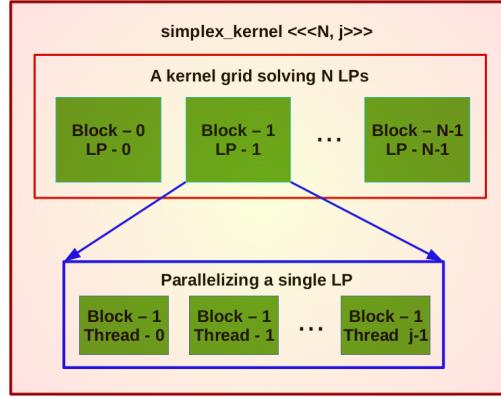


Figure 2: Visualization of how threads are mapped to solve N LPs in GPU. Each block is mapped to an LP and j threads are assigned to parallelize a single LP.

4.2. Simplex Algorithm Implementation

Finding the pivot column in **step 1** of the simplex algorithm above requires to determine the index of the maximum value from the last row of the tableau. We have parallelized **step 1** by utilizing n (out of j) threads in parallel to determine the pivot column using **parallel reduction** described in [21]. A parallel reduction is a technique applied to achieve data parallelism in GPU when a single result (e.g. min, max) is to be computed out of a large number of data. We have implemented a parallel reduction by using two auxiliary arrays, one for storing the data and the other for storing the indices of the corresponding data. The result of a parallel reduction algorithm provides us the maximum value in the first array and its corresponding index in the other array.

We also applied parallel reduction in **step 2** by utilizing m (out of j) threads

in parallel to determine the pivot row (m being the row-size of the simplex tableau). Using parallel reduction in step 2 requires other modifications. It involves finding a minimum positive value from a vector of ratios (as described in Step 2 above) and therefore ratios which are not positive needs to be excluded from the minimum computation. This leads to a conditional statement in the parallel reduction algorithm and degrades performance due to warp divergence. Even if we re-size the vector to store only the positive values, the kernel will still require conditional statements to check the thread IDs that need to process this smaller size vector. To overcome performance degradation with conditional statements, we substituted a large positive number in place of ratios that are negative or undefined. This creates a vector that is suitable for parallel reduction in our kernel implementation.

Data parallelism is also employed in the pivot operation in **step 3**, involving substitution and re-writing, using the $(m - 1)$ threads (out of j threads on the block).

There are a number of pivot selection rules that could be applied in step 1. In this work, we have experimented with two pivot selection rules, to study its effect on the performance of simplex algorithm in the GPU. We describe these pivot selection rules below:

Largest Positive Coefficient (LPC):. We take the index of the maximum positive coefficient in the last row of the simplex tableau (step 1 in the above algorithm).

Random Positive Coefficient (RPC):. Instead of choosing the index of the maximum positive coefficient as in LPC, we choose a random index having a positive coefficient from the last row of the simplex tableau. Although this rule is generally not efficient since it may result in more iterations in the algorithm, our purpose is to see its effect in the context of a GPU implementation since it requires no overhead of parallel reduction unlike the LPC rule. The choice of this pivot selection rule may be appropriate in the context of the GPU as a warp of 32 threads can read simultaneously 32 values in only one cycle and can assign only the index containing the positive value to a shared variable, as the pivot

column. Therefore, RPC rule seems to incur less overhead in simplex iterations in GPU compared to the LPC rule. However, it remains to be experimented if this gain dominates the loss of performance due to the possible extra iterations. Our observations on the performance using the above mentioned pivot selection rules are illustrated in Section. 4.6.

4.3. Memory Coalescent Access

In this section, we discuss our efforts of keeping a coalescent access to global memory to reduce performance loss due to cache misses. When all threads of a warp access contiguous region of the memory, it is coalesced and this ensures improved performance due to high cache hit rate. However, if the access to memory is not coalesced, then the memory controller undergoes cache block replacements that incur delays and degrades the performance in GPU.

As discussed earlier, we use global memory to store the simplex tableaux of the LPs in a batch as described in Section 3.1 (Global memory being the largest can accommodate many tableaux). We store the simplex tableau in memory as a two-dimensional array. High level languages like C and C++ uses the row-major order by default for representing a 2-dimensional array in the memory. CUDA is an extension to C/C++ and also uses the row-major order. The choice of row or column major order representation of two-dimensional arrays plays an important role in deciding the efficiency of the implementation, depending on whether the threads in a warp access the adjacent rows or adjacent columns of the array and what is the offset between the consecutive rows and columns.

We use the term *column-operation*, when element of all rows from a specific column is accessed simultaneously by each thread in a warp. If the array is in a row-major order, then this operation is not a coalesced memory access, as each thread accesses elements from the memory separated by the size equal to the column-width of the two dimensional array. When elements of a specific row are accessed simultaneously by each thread of a warp, we called this a *row-operation*. Note that for a two dimensional array stored in row-major order, a row-operation is coalesced since each thread accesses data from contiguous

region in the memory.

We show below that in the simplex algorithm described above, there are more column-operations than row operations and thus, storing our data (i.e. simplex tableau) in a column-major order would ensure higher coalesced memory accesses.

Step 1 of the simplex algorithm determines the entering variable (also known as the pivot column), which requires finding the index of the maximum positive coefficient (in case of LPC rule) from the last row. This requires a row-operation and as mentioned in Section 4, we use parallel reduction using two auxiliary arrays, *Data* and *Indices* as shown in Figure 3. Although accessing from the last row of the simplex tableau is not coalesced (due to our column-major ordering) but copying into the *Data* (and *Indices*) array is coalesced and so is the parallel reduction algorithm on the *Data* (and *Indices*) array. We used the technique of *Parallel Reduction: Sequential Addressing* in [21], a technique that ensures coalesced memory access.

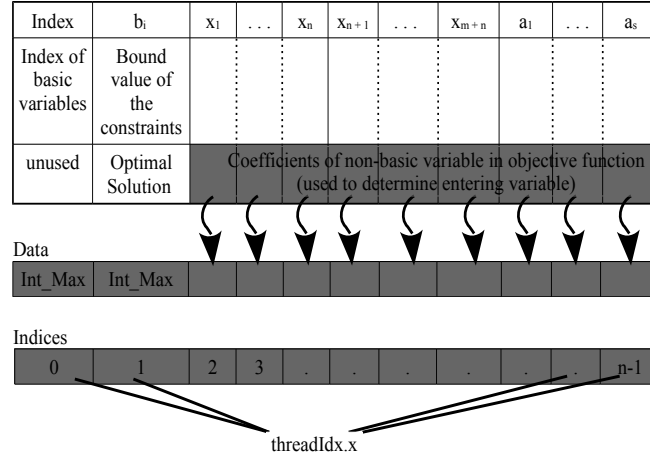


Figure 3: Showing the Simplex Tableau along with two separate arrays, *Data* to store the coefficients of the objective function and *Indices* to keep track of the indices of the corresponding values in the *Data* array.

Step 2 of the simplex algorithm determines the leaving variable (also called the pivot row) by computing the row index with the minimum positive ratio

$(b_i / -a_{e,i})$, as described in Section 3.1. This requires two column-operations involving the access to all elements from columns b_i and $a_{e,i}$ as shown in Figure 4. To compute the row index with the minimum positive ratio, we use parallel reduction as described above in Section 4. Our tableau being stored in a column-major order, access to columns b_i and $a_{e,i}$ are both coalesced. The ratio and its corresponding indices (represented by the thread ID) are stored in the auxiliary arrays, *Data* and *Indices* which is also coalesced. Like in Step 1, we use the same technique of *Parallel Reduction: Sequential Addressing* in [21] for coalesced memory access.

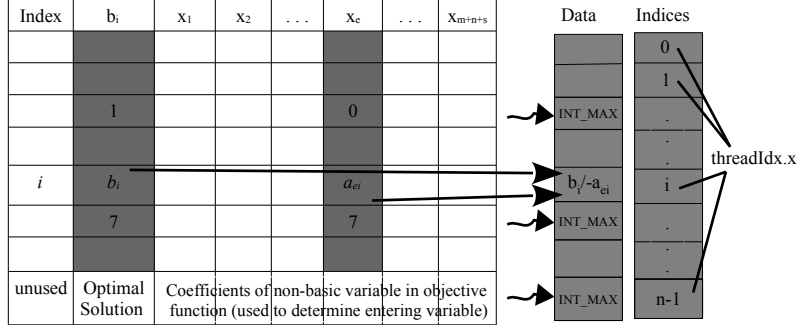


Figure 4: Showing the Simplex Tableau along with two separate arrays, *Data* to store the positive ratio and *Indices* to keep track of the indices of the corresponding values in the *Data* array. Ratios that reduces to negative or undefined are replaced by a large value denoted by INT.MAX.

Step 3 performs the pivot operation that updates the elements of the simplex tableau and is the most expensive of the three steps. It first involves a non-coalesce row-operation which computes the new modified pivot row (denoted by the index ℓ) as $\{NewPivotRow_\ell = OldPivotRow_\ell \div PE\}$, where PE is the element in cell in the intersection of the pivot row and the pivot column for that iteration, known as the pivot element. The modified row ($NewPivotRow_\ell$) is then substituted to update each element of all the rows of the simplex tableau, using the formula $NewRow_{ij} = OldRow_{ij} - PivotCol_{ie} * NewPivotRow_{\ell j}$ (see the code Listing 1 below). The elements of the pivot column are first stored in an array named *PivotCol* which is a column-operation, and so is coalesced,

due to the column-major representation of the tableau. The crucial operation is updating each j^{th} element for every i^{th} row (except the pivot row ℓ) of the simplex tableau, which requires a nested for-loop operation. We have parallelized the outer for-loop that maps the rows of the simplex tableau. Our data being represented in a column-major order, so parallel access to all rows for each element in the j^{th} column of the inner for-loop is coalesced.

Listing 1: Showing code fragment for step 3 that updates the simplex tableau.

```

for (int i=0;i<rows;i++) { //Parallelized outer loop to
    ↪ map each i with the thread ID
    for (int j=0;j<cols;j++) {
        NewRow[i][j] = OldRow[i][j] - PivotCol[i] *
            ↪ NewPivotRow[l][j]; //l index of pivot row
    }
}

```

To verify the performance gained due to coalesced memory access, we have experimented with **Step 3** which is the most expensive of all steps in the simplex algorithm, by modifying it to have non-coalesced memory access. In the code Listing 1, we interchange the inner for-loop with the outer loop (loop interchange, a common technique to improve cache performance[22]). This loop interchanging converts the Step 3 to have non-coalesced memory access since our simplex tableau is represented in a column-major order. Figure 5 presents the experimental results to show the gain in performance when the access to memory is coalesced as compared to non-coalesced access. Clearly, the result has shown a significant gain in performance on a Tesla K40c card, implementing the LPC pivot selection rule for LPs with initial basic solution as feasible.

We observed that step 1 has a row-operation, step 2 has two column-operations and step 3 has a row and a column operation each along with a nested for-loop which can be expressed both row as well as column operations. Clearly, there are more column-operations than rows. However, the size of column is more than

LP Dim	Batch-size	Non-coalesced Access Time (seconds)	Coalesced Access Time (seconds)	Speed-up
10	1000	0.193	0.016	12.06
50	1000	0.286	0.033	8.67
100	1000	0.947	0.105	9.02
200	1000	4.739	0.397	11.94
300	1000	14.482	0.921	15.72
400	1000	30.320	2.109	14.38
500	1000	43.416	2.844	15.27

Figure 5: Showing the time taken to solve batched LP due to coalesced and non-coalesced memory access on GPU, in LPC implementation for LPs with initial basic solution as feasible.

row of our simplex tableau, therefore, one can experiment on the row-major layout of the tableau, to determine if this representation has higher coalesced memory accesses.

4.4. Overlapping data transfer with kernel operations using CUDA Streams

The memory bandwidth of host-device data copy is a major bottleneck in CUDA applications. We use Nvidia’s profiling tool **nvprof** [23] to profile time for memory transfer and kernel operation for our implementation discussed above in Section 4. The result of profiling in a Tesla K40c card, implementing the LPC pivot selection rule for LPs with an initial basic solution as feasible, is reported in Figure 6. We observed that, for a small batch-size problem (e.g. 10 in the Figure 6), the memory copy operation is a maximum of 5.75%, whereas for bigger batch-size problem the memory copy operation is in the range of 10 – 15% and above. Although, the value is not substantial for significant performance tuning, but it cannot be ignored either.

A standard technique to improve performance in CUDA applications is by using CUDA streams which allow overlapping memory copies with kernel execution. A stream in CUDA consists of a sequence of operations, which is executed on the device in the order in which they are issued by the host procedure. These different sequence of operations not only can be interleaved, but can also be executed concurrently in order to gain higher performance as described in [24].

LP Dimension	Batch-size	Time %			
		Kernel	MemCpy		Total Time
			H2D	D2H	
10	10	98.79	0.63	0.59	100
10	90000	93.93	6.06	0.01	100
50	10	99.22	0.71	0.07	100
50	90000	84.74	15.26	0.00	100
200	10	98.40	1.59	0.01	100
200	9000	84.93	15.07	0.00	100
500	10	94.25	5.75	0.00	100
500	900	86.04	13.96	0.00	100

Figure 6: Showing the profile report obtained using **nvprof** tool in our LPC implementation for LPs with an initial basic solution as feasible. H2D - stands for host to device and D2H indicates device to host memory copies respectively.

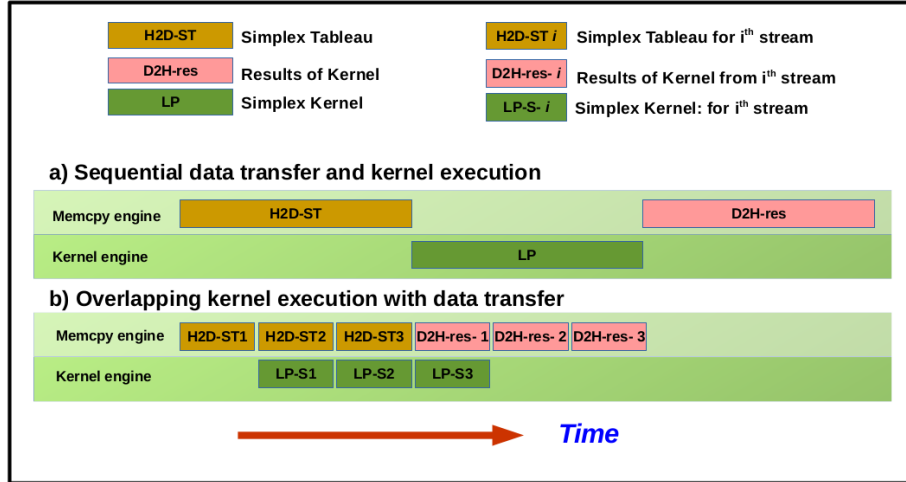


Figure 7: Showing the gain in time due to overlapping kernel execution with data transfer compared to sequential data transfer and kernel execution. The time required for host-to-device(H2D), device-to-host(D2H) and kernel execution are assumed to be same.

A GPU in general, has a separate kernel and a copy engine. All kernel operations are executed using the kernel engine and memory copy operations to and from the device is performed by the same copy engine. However, some GPU cards have two copy engines, one each for copying data to and from the device, to achieve higher performance on the GPU. Figure 7, illustrates the

overlapping of kernel executions with data copy, when the GPU has only one kernel and copy engine each. To obtain maximum performance in such GPU configuration, streaming by batching similar operations provides more overlap of copies with kernel executions. This is done by adding all host-to-device copy to the different streams followed by all kernel launches and device-to-host data copies. When there is to copy engines, looping the operations in the order of a host-to-device copy followed by kernel launch and device-to-host copy, for all streams would yield higher performance than the former method. However, for all devices with compute capability 3.5 and above, both the methods yield same performance, due to the Hyper-Q [25] feature enabled in them.

Higher number of CUDA streams achieves higher concurrency and interleaving among operations, but it involves stream creation overhead. The number of CUDA streams that gives optimal performance is found by experimentation. From our experimental observations, we conclude that with varying batch size and LP dimension to be solved, the optimal number of streams also varies. In this paper, we have reported the results with 10 streams for batch size higher than 100 LPs and only 1 stream when the batch size is less than 100 (for LPs of any dimension).

4.5. Limitations of the Implementation

The memory required for an LP (i.e., a tableau) in our implementation can be approximately computed as:

$$Y = \{(m + 1) \times cols \times dataSize + x\} \quad (5)$$

$$cols = (var + slack + arti + 2)$$

$$dataSize = sizeof(DType)$$

$$x = 2 \times (cols \times dataSize)$$

where $(m + 1)$ and $cols$ are the sizes of rows and columns of the simplex tableau respectively. Thus, the size of each LP is Y bytes, where $DType$ is the data type being used and x being the size of array used for performing parallel reduction operation, the number 2 in the equation $x = 2 \times (cols \times dataSize)$

signify the use of two auxiliary arrays. The size of the *cols* is described in Subsection 3.1. Thus, if S is the size of total global memory (in bytes) available in the GPU, then our threshold limit or the number of LPs that can be solved at a time is determined by the equation $N = \lfloor \frac{S}{V} \rfloor$. As the current limit on threads per block is 1024 for GPU, thus, our implementation limits the size of an LP problem to 511×511 for LP problems whose *initial basic solution is feasible* and up to 340×340 for the class of LP problems with *initial basic solution as infeasible*. This limit is defined by the inequality (6)

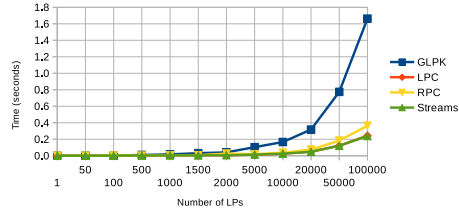
$$(var + slack + arti + 2) \leq 1024 \quad (6)$$

where *var* is the number of variables (dimension of the LP problem), *slack* is the number of slack variables (or constraints) and *arti* is the number of artificial variables (if any) of the given LP as in the equation (5). This limitation can be overcome either by mapping a single thread to work on more than one data-instruction at a time or by mapping an LP problem with more than one thread blocks.

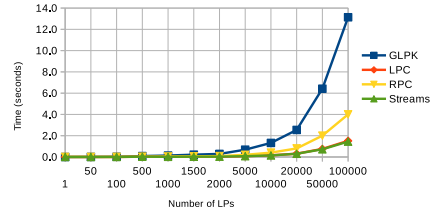
4.6. Performance Analysis of Solving Batched LPs on GPU

We performed our experiment in Intel Xeon E5-2670 v3 CPU, 2.30GHz, 12 Core (without hyper-threading), 62GB RAM with Nvidia’s Tesla K40c GPU card. The reported running time is an average over 10 runs. We observed a maximum speedup of $16.43\times$ for 100-dimensional LP runs 20k LPs, using the LPC rule of pivot selection and a speedup of $6.74\times$ running 50k LPs of 100-dimension using the RPC rule of pivot selection, as compared to GLPK for LP problems which has *initial basic solution as feasible*. A maximum speedup of $18.30\times$ is observed on 50k LPs of size 100 using streams with LPC rule, as compared to GLPK for LP problems which has *initial basic solution as feasible*, as shown in Figure 8. We observed that for LPs of large size, our CUDA implementation performs better even with few LPs in parallel (e.g., batch-size= 50 for 500 dimensional LP). However, for small size LPs, our CUDA implementation out-performs GLPK only for larger batch-size (e.g. 100 and above for 5

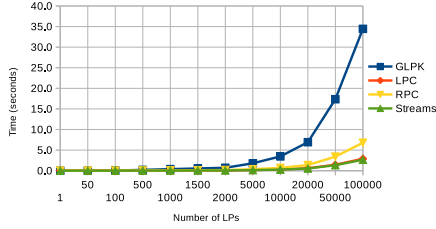
dimensional LP). We also observe that the LPC pivot selection rule shows better performance than the RPC rule, although LPC involves the extra overhead of computing the maximum in each simplex iteration using parallel reduction. It is known that in most cases, the LPC rule converges to the optimum in less number of simplex iterations compared to the RPC rule. Therefore, we can deduce that the time taken in computing the extra iterations that are required using the RPC rule overshoots the performance gain by avoiding the maximum computation at each iteration.



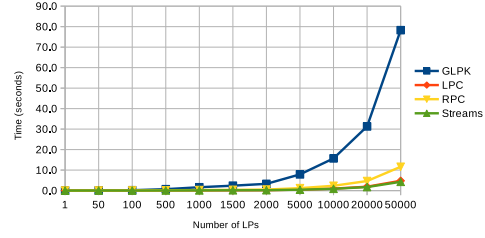
(a) 5-Dimension



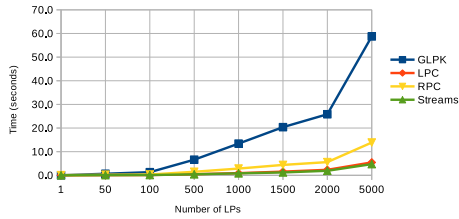
(b) 28-Dimension



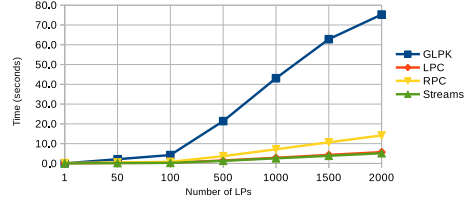
(c) 50-Dimension



(d) 100-Dimension



(e) 300-Dimension



(f) 500-Dimension

Figure 8: Showing time taken to compute a batch of LPs for dimensions 5, 28, 50, 100, 300 and 500 respectively for the type of LPs with **initial basic solution as feasible**.

For LP problems with *infeasible initial basic solution*, though our implementation had to execute the kernel twice due to the two-phase simplex algorithm as described above in Section 3 (an extra overhead of data exchange between the two kernels), but we still observed that our implementation performed better than the GLPK library. We gain a maximum speedup of $11.96\times$ for $10k$ LPs of size 200 using the LPC pivot selection rule compared to GLPK as shown in Figure 9.

Batch-Size	5 Dimension			28 Dimension			50 Dimension			100 Dimension			200 Dimension		
	Time(Sec)		Speed-up	Time(Sec)		Speed-up	Time(Sec)		Speed-up	Time(Sec)		Speed-up	Time(Sec)		Speed-up
	GLPK	LPC	LPC Vs GLPK	GLPK	LPC	LPC Vs GLPK	GLPK	LPC	LPC Vs GLPK	GLPK	LPC	LPC Vs GLPK	GLPK	LPC	LPC Vs GLPK
50	0.001	0.001	1.00	0.004	0.001	4.00	0.014	0.004	3.50	0.056	0.009	6.22	0.195	0.021	9.29
100	0.001	0.001	1.00	0.009	0.002	4.50	0.028	0.005	5.60	0.113	0.012	9.42	0.371	0.035	10.60
500	0.009	0.002	4.50	0.049	0.007	7.00	0.140	0.017	8.24	0.499	0.046	10.85	1.786	0.154	11.60
1000	0.017	0.004	4.25	0.093	0.012	7.75	0.260	0.032	8.13	0.975	0.093	10.48	3.540	0.295	12.00
1500	0.030	0.006	5.00	0.135	0.018	7.50	0.381	0.047	8.11	1.465	0.130	11.27	5.303	0.454	11.68
2000	0.043	0.008	5.38	0.184	0.025	7.36	0.504	0.063	8.00	1.942	0.176	11.03	7.092	0.603	11.76
5000	0.089	0.018	4.94	0.410	0.059	6.95	1.224	0.153	8.00	4.812	0.428	11.24	17.638	1.639	10.76
10000	0.163	0.036	4.53	0.783	0.111	7.05	2.454	0.303	8.10	9.689	0.859	11.28	35.498	2.969	11.96
20000	0.333	0.069	4.83	1.575	0.223	7.06	4.899	0.609	8.04	19.225	1.872	10.27	70.805	6.110	11.59
50000	0.749	0.169	4.43	4.131	0.575	7.18	12.233	1.498	8.17	47.825	4.742	10.09			
100000	1.470	0.339	4.34	7.984	1.121	7.12	24.478	3.142	7.79	96.05	8.76	10.96			

Figure 9: Showing comparison between GLPK and GPU implementation for the type of LPs with **initial basic solution as infeasible**

While profiling the CUDA streams, we observed that for small sized LPs, the processing time of the kernel is much larger than the data transfer time as in Figure 6 and so, the gain in performance of overlapping data transfer with kernel is also negligible as shown in 10a, 10b and 10c. But as the LP size increases (e.g., 500) the size of data transfers are also significantly larger as in Figure 6. Hence, the operation of data transfer for all the streams (except the first) can be overlapped while the first kernel is in execution, thereby saving the time for data transfer in the rest of the stream. Thus, an extra 2 – 3% gain in performance for LPs of larger dimensions is observed in our experiments, as shown in Figure 8, due to the overlapping of data transfer with the kernel’s execution using the CUDA streaming technique.

5. Implementation of Thread Safe GLPK

GLPK (GNU Linear Programming Kit) is a commonly used open source and optimized linear programming solver library. However, the GLPK library is not thread safe, i.e., multiple threads running independent instances of GLPK objects may produce inconsistent results. In order to solve multiple LPs in parallel using GLPK, one may have a multi-process implementation which is not efficient since each process own its own memory space (when large numbers of GLPK objects are forked, large memory is going to be used). Experimentally, we observed that a multi-process GLPK implementation consumes double the size of memory as compared to its multi-threaded implementation. We identified that GLPK has a shared data which results in race condition on a multi-threaded setting. We modified the shared data (a pointer variable named *tls* in the source file, “glpenv02.c”) to thread-local in the source and we could ensure thread safety, enabling us to make parallel calls to GLPK in order to solve multiple LPs using multi-threading. Each independent thread, still solve a single LP sequentially. We use the OpenMP directives to create multiple execution threads, each making a call to an LP solver.

We performed multi-threading experiments with the thread safe GLPK on a 12 cores Intel Xeon CPU E5-2670, 2.30GHz with 62.8 GB RAM, for an average of 10 runs. We observed a maximum of $9.6\times$ speedup for $1e5$ LPs of dimension 100, using the thread parallel GLPK as compared to sequential solving using GLPK. We have also recorded the overhead in memory (in Megabytes (MB)) incurred due to threading multiple GLPK objects in parallel as shown in Figure 10 by the column labeled “Extra mem. (Par - Seq)”. The table column “CPU util. Gain (%)”, is the difference of CPU utilization in parallel with that of the Sequential executions. We observed that in our experimental setup, the thread parallel GLPK out performs the sequential GLPK only for LPs of dimension 16 and above. For LPs with smaller dimensions, the penalty of thread creation and context switching is more than the the gain with parallelization.

Clearly, for large size LPs we observed the gain in speedup but at the cost

No. of LPs	Time (in Secs)		Speed-up	Extra Mem. (Par - Seq)	CPU util. Gain (%)
	SEQ	PAR			
10	0.000	0.000	0.0	0.61	3.3
100	0.003	0.002	1.7	0.74	5.3
1000	0.024	0.023	1.1	0.71	89.9
5000	0.121	0.030	4.0	0.79	86.3
10000	0.240	0.273	0.9	0.78	59.4
50000	1.165	0.658	1.8	1.12	43.4
100000	2.362	3.950	0.6	1.46	27.4
200000	4.715	7.662	0.6	1.33	26.4
300000	7.102	11.422	0.6	2.67	25.9
400000	11.240	15.930	0.7	3.68	26.4
500000	11.730	19.938	0.6	4.27	25.1
1000000	23.454	39.631	0.6	7.04	25.5

(a) 10-Dimension

No. of LPs	Time (in Secs)		Speed-up	Extra Mem. (Par - Seq)	CPU util. Gain (%)
	SEQ	PAR			
1000	0.320	0.047	6.8	1.94	90.2
5000	1.597	0.200	8.0	1.71	89.5
10000	3.159	0.386	8.2	1.83	89.2
50000	15.981	1.947	8.2	2.05	84.8
100000	31.975	3.862	8.3	2.43	83.9
200000	63.662	7.737	8.2	2.15	82.5
300000	94.244	10.395	9.1	2.34	89.2
400000	127.274	15.437	8.2	3.98	81.9

(b) 50-Dimension

No. of LPs	Time (in Secs)		Speed-up	Extra Mem. (Par - Seq)	CPU util. Gain (%)
	SEQ	PAR			
1000	1.424	0.153	9.3	2.96	89.4
5000	7.288	0.945	7.7	3.10	82.1
10000	14.588	1.521	9.6	5.01	88.1
50000	72.378	8.116	8.9	3.35	80.4
100000	141.838	14.905	9.6	3.89	86.6
200000	280.984	29.930	9.4	2.97	86.0
300000	422.110	44.971	9.4	5.23	85.7
400000	569.272	61.246	9.3	4.17	84.2

(c) 100-Dimension

No. of LPs	Time (in Secs)		Speed-up	Extra Mem. (Par - Seq)	CPU util. Gain (%)
	SEQ	PAR			
1000	5.329	0.872	6.1	8.98	81.3
5000	26.113	3.818	6.8	8.97	84.8
10000	52.282	7.652	6.8	8.98	83.6
50000	258.696	38.079	6.8	9.12	83.0
100000	521.494	89.326	5.8	9.07	84.7
200000	1036.652	159.880	6.5	10.01	85.9
300000	1533.904	229.137	6.7	10.05	82.6
400000	2088.620	308.442	6.8	8.97	81.7

(d) 200-Dimension

Figure 10: Showing comparison between multiple LPs solved in parallel using thread safe and sequential GLPK implementation on a 12-Core Intel Xeon processor

of memory overheads due to threading multiple GLPK objects.

6. CUDA Implementation of Special-Case LPs

The feasible region of an LP given by its constraints defines a convex polytope. We observe that when the feasible region is a hyper-rectangle, which is a special case of a convex polytope, the LP can be solved cheaply. Equation (7) shows that maximizing the objective function is the sum of the results on n dot products.

$$\underset{x \in \mathcal{B}}{\text{maximize}}(\ell.x) = \sum_{i=1}^n \ell_i.h_i, \text{ where } h_i = \begin{cases} a_i & \text{if } \ell_i < 0 \\ b_i & \text{otherwise} \end{cases} \quad (7)$$

where $\ell \in \mathbb{R}^n$ is the sampling directions over the given hyperbox $\mathcal{B} = \{x \in \mathbb{R}^n | x \in [a_1, b_1] \times \dots \times [a_n, b_n]\}$.

The performance results of our GPU implementation of the hyperbox LP solver are presented in Table 1. In order to solve many LPs in parallel, we organize CUDA threads in a one-dimensional block of threads with each block used to solve an LP. Each block is made to consist of only 32 threads, the warp size. Within each block, we used only a single thread to perform the operations of the kernel. The operation $\sum_{i=1}^n l_i \cdot h_i$, which can be performed using parallel reduction is expensive than computing sequentially, due to the overheads in implementing the parallel reduction technique. A preliminary introduction about this technique is introduced in the paper[26].

Model	Nos. of LPs	Time (in Secs)		Speed-up vs. GLPK
		GLPK	GPU	
Five Dim. Model	20010	0.120	0.006	20.0
	100050	0.643	0.014	45.9
	1000500	6.385	0.116	55.0
	2000500	12.771	0.225	56.8
	2001000	12.927	0.214	60.4
	4001000	25.780	0.406	63.5
Helicopter Controller	56056	0.746	0.027	27.6
	112056	1.487	0.054	27.5
	1569568	20.931	0.636	32.9
	2002000	27.617	0.799	34.6
	3003000	40.500	1.224	33.1
	6003000	81.476	2.388	34.1

Table 1: Comparing GLPK with Hyperbox LP Solver in GPU

We present the performance gained by our GPU implementation for hyperbox LPs in Table 1, as compared to solving sequential using GLPK. In general, our hyperbox LP solver can simultaneously solve a batch of independent LPs (i.e. Each LP with a different set of constraints), but to keep the experimental setup same as in Table 2 we consider the same LPs (i.e. All LPs with the same set of constraints) of five and 28 dimensions with large number of different objective functions. This setup enables an efficient sequential GLPK implementation. The column labeled “No. of LPs” in Table 1 indicate the total number

of objective functions required to be solved for the same given LP problem.

We performed our experiment in Intel Q9950, 2.84Ghz, 4 Core (no hyper-threading), 8GB RAM with GeForce GTX 670 GPU card for an average of 10 runs. We observed a $63.5\times$ speedup for 4001000 LPs of 5-dimension and a $34.12\times$ speedup for 6003000 LPs of 28-dimension, using our hyperbox LP solver in GPU as compared to the GLPK solver.

Model	Nos. of LPs	Time (in Secs)			Speed-up	
		Seq	SpaceEx	Par (GPU)	vs. Seq	vs. SpaceEx
Five Dim. Model	20010	0.133	0.345	0.018	7.4	19.2
	100050	0.717	1.399	0.060	12.0	23.3
	1000500	6.695	24.171	0.576	11.6	42.0
	2001000	13.128	59.996	1.121	11.7	53.5
Helicopter Controller	56056	1.400	4.399	0.172	8.1	25.6
	1569568	39.089	123.794	4.246	9.2	29.2
	2002000	50.367	187.825	5.397	9.3	34.8
	3003000	75.087	311.652	8.055	9.3	38.7

Table 2: Performance Speed-up in XSpeed using Hyperbox LP Solver

7. Application of Parallel LP Solving in GPU

In the design of control systems, a standard technique of analysis is by mathematically modeling the control system design and computing the state-space of the model using exploration algorithms. Properties of the control system such as safety and stability can be analyzed with the computed state-space. In this section, we discuss two open-source tools that perform state-space exploration of linear systems with continuous dynamics. First is the tool SpaceEx [27] and the second is XSpeed [26]. These tools can analyze systems modeled as ordinary differential equations with uncertainty ($\dot{x} = A.x(t) + u$, $u \in \mathcal{U}$, $x \in \mathcal{X}_0$ at $t = 0$), where the set \mathcal{U} models the set of all possible control inputs. A conservative over-approximation of the exact reachable state space is computed by both these tools. A common state space computation algorithm in these

tools compute the reachable state space as a union of convex sets, each having a symbolic representation in memory, known as the support function representation [28]. However, the algorithm requires to convert the convex sets from its support function representation to convex polytope representation, for certain operations to be efficient and for the visualization of the state space. Such a conversion to convex polytope representation loses precision since they provide an over-approximation of the original convex set. A support function of a convex set $\Omega \subset \mathbb{R}^d$ is a function that takes a vector $\ell \in \mathbb{R}^d$ as an argument and produces the real number $r = \max_{x \in \Omega} \ell \cdot x$. The conversion from a support function representation to a polytope representation involves sampling the support function in a finite number of arguments ℓ . The precision of the conversion depends on the number of function samples taken. It can be easily seen that sampling the support function of convex sets which are convex polytopes, is a linear programming problem. It is common in applications to have these convex sets Ω as convex polytopes. Therefore, these conversions results in many LPs to be solved. Figure 11 shows the computed reachable state space of the model of a five dimensional system with different precisions, approximated by a union of convex polytopes using the tool XSpeed. It can be seen that the precision of the state space shown in green is better than the one shown in red. The former requires solving 1e6 LPs of size five whereas the later requires 1e5 LPs of same dimension to be solved. Therefore, we see that reachability analysis tools requires a large number of LP solving. We now briefly discuss two continuous systems of small dimension and illustrate the performance speedup obtained by using our library in the XSpeed tool with batched LP solving.

7.1. Helicopter Controller

It is a model of a twin-engined multi-purpose military helicopter with 8 continuous variables modeling the motion and 20 controller variables that govern the various controlling actions of the helicopter [29, 27]. We consider the initial input set X_0 to be an hyperbox and the non-deterministic input sets \mathcal{U} as a point set. The fact that the given input set is an hyperbox enables us to perform all

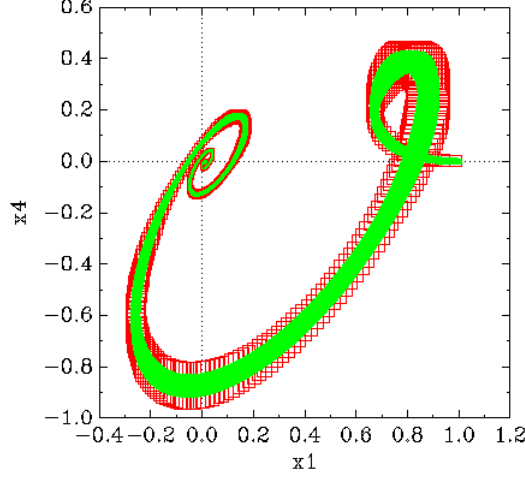


Figure 11: Reachable State Space of a Five Dimensional Model with Different Precisions.

computation of support function samplings using our special-case LP solver in GPU as described earlier.

7.2. Five Dimensional System

It is a model of a five dimensional linear continuous system as defined in [30]. We consider the initial input set X_0 as an hyperbox centered at $(1,0,0,0)$ with sides of length 0.02 units. We take the non-deterministic input to be a point set as $(0.01,0.01,0.01,0.01,0.01)$. We direct the reader to the paper [30] for details about the dynamics of the model.

We performed our experiment in a system with Intel Q9950, 2.84Ghz, 4 Core (no hyper-threading), 8GB RAM with a GeForce GTX 670 card. The performance reported averages over 10 runs. In comparison to the sequential solving of LPs using the GLPK library call, we observed a maximum of $12\times$ and $9\times$ speedup with parallel LP solving in the GPU, for a five dimensional system and a helicopter controller benchmark respectively. When compared to the tool SpaceEx, we observed a maximum of $54\times$ and $39\times$ speedup in XSpeed using our CUDA implementation, for a five dimensional system and a helicopter controller model respectively, as shown in Table 2 (a section of this result has

also been published in the paper [26]).

8. Conclusion

We present a CUDA implementation to solve multiple LPs of small to medium size simultaneously in a GPU. We explain the implementation choices to have a coalescent memory access, efficient load balancing and efficient CPU-GPU memory copy operation using CUDA streams. We show the techniques that have been used to implement the simplex algorithm, like parallel reductions and loop interchange. We experiment with two pivot selection rules in the simplex algorithm to observe its performance in GPU. We deduce with experiments that the LPC rule shows better performance than the RPC rule in GPU even though LPC involves the extra overhead of parallel reduction. We present a thread safe GLPK implementation and its experimental evaluation in solving many LPs with multi-threading in a multi-core CPU. We demonstrate significant performance speedup with the multi-threaded implementation. It is observed that LPs can be solved very cheaply when its feasible region that given by its constraints is a hyperbox, i.e., a subclass of general convex polytopes. We implement the solution of such special case LPs in CUDA and report significant performance improvement when compared to solving using GLPK. Lastly, We illustrate an application which involves many LP solving of small to moderate size and we show the performance speedup of the application with our batched LP solver library.

Acknowledgments

This work was supported by the National Institute of Technology Meghalaya, India and by the the DST-SERB, GoI under project grant No. YSS/2014/000623.

References

- [1] M. Birk, R. Dapp, N. V. Ruiter, J. Becker, Gpu-based iterative transmission reconstruction in 3d ultrasound computer tomography, *Journal of Parallel and Distributed Computing* 74 (1) (2014) 1730–1743.
- [2] G. Yan, J. Tian, S. Zhu, Y. Dai, C. Qin, Fast cone-beam CT image reconstruction using GPU hardware, *Journal of X-Ray Science and Technology* 16 (4) (2008) 225–234.
URL <http://iospress.metapress.com/content/17Q3P507TG723144>
- [3] J. Michalakes, M. Vachharajani, GPU Acceleration of Numerical Weather Prediction, *Parallel Processing Letter* (2008) 1–18.
- [4] J. Zhang, I. McQuillan, F. Wu, Speed Improvements of Peptide - Spectrum Matching Using SIMD Instructions, *IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBMW)* (2010) 1–22.
- [5] M. A. Clark, P. La Plante, L. J. Greenhill, Accelerating radio astronomy cross-correlation with graphics processing units, *International Journal of High Performance Computing Applications* (2012) 1094342012444794.
- [6] K. Karimi, N. Dickson, F. Hamze, High-performance physics simulations using multi-core cpus and gpgpus in a volunteer computing context, *International Journal of High Performance Computing Applications* 25 (1) (2011) 61–69.
- [7] R. K. Lim, J. W. Pro, M. R. Begley, M. Utz, L. R. Petzold, High-performance simulation of fracture in idealized brick and mortarcomposites using adaptive monte carlo minimization on the gpu, *International Journal of High Performance Computing Applications* (2015) 1094342015593395.
- [8] K. Shirahata, H. Sato, T. Suzumura, S. Matsuoka, A GPU Implementation of generalized graph processing algorithm GIM-V, *Proceedings - 2012 IEEE International Conference on Cluster Computing Workshops, Cluster Workshops 2012* (2012) 207–212doi:10.1109/ClusterW.2012.34.

- [9] J. Chandra, S. O. Schall, Economic justification of flexible manufacturing systems using the leontief input-output model, *The Engineering Economist* 34 (1) (1988) 27–50.
- [10] J. Munkres, Algorithms for the assignment and transportation problems, *Journal of the Society for Industrial and Applied Mathematics* 5 (1) (1957) 32–38.
- [11] M. Drozdowski, M. Lawenda, F. Guinand, Scheduling multiple divisible loads, *International Journal of High Performance Computing Applications* 20 (1) (2006) 19–30.
- [12] M. E. Lalami, D. El-Baz, V. Boyer, Multi gpu implementation of the simplex algorithm, in: *2011 IEEE International Conference on High Performance Computing and Communications*, 2011.
- [13] J. Bieling, P. Peschlow, P. Martini, An efficient gpu implementation of the revised simplex method, in: *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010 IEEE International Symposium on, IEEE, 2010, pp. 1–8.
- [14] D. G. Spampinato, A. C. Elster, Linear optimization on modern gpus, in: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, IEEE, 2009, pp. 1–8. doi:10.1109/IPDPS.2009.5161106. URL <http://dx.doi.org/10.1109/IPDPS.2009.5161106>
- [15] G. B. Dantzig, M. N. Thapa, *Linear Programming 1: Introduction*, Springer, 1997.
- [16] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Performance tuning and optimization techniques of fixed and variable size batched cholesky factorization on gpus, *Procedia Computer Science* 80 (2016) 119–130.

- [17] C. Jhurani, P. Mullenney, A gemm interface and implementation on nvidia gpus for multiple small matrices, *Journal of Parallel and Distributed Computing* 75 (2015) 133–140.
- [18] A. Makhorin, GNU Linear Programming Kit, v.4.37, <http://www.gnu.org/software/glpk> (2009).
- [19] R. C. Whaley, Atlas (automatically tuned linear algebra software), in: *Encyclopedia of Parallel Computing*, Springer, 2011, pp. 95–101.
- [20] Blas–basic linear algebra subprograms (2016).
URL <http://www.netlib.org/blas/>
- [21] M. Harris, Optimizing parallel reduction in CUDA, NVIDIA Developer Technology.
URL <http://vuduc.org/teaching/cse6230-hpcta-fa12/slides/cse6230-fa12--05b-reduction-notes.pdf>
- [22] J. L. Hennessy, D. A. Patterson, *Computer architecture: a quantitative approach*, Elsevier, 2011.
- [23] Nvidia, Cuda c programming guide (2015).
URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
- [24] M. Harris, How to Overlap Data Transfers in CUDA C/C++.
URL <http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/>
- [25] T. Bradley, Hyper-q example (2012).
URL http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf
- [26] R. Ray, A. Gurung, B. Das, E. Bartocci, S. Bogomolov, R. Grosu, Xspeed: Accelerating reachability analysis on multi-core processors, in: N. Piterman (Ed.), *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November*

17-19, 2015, Proceedings, Vol. 9434 of Lecture Notes in Computer Science, Springer, 2015, pp. 3–18. doi:10.1007/978-3-319-26287-1_1.
URL http://dx.doi.org/10.1007/978-3-319-26287-1_1

- [27] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, O. Maler, Spaceex: Scalable verification of hybrid systems, in: S. Q. Ganesh Gopalakrishnan (Ed.), Proc. 23rd International Conference on Computer Aided Verification (CAV), LNCS, Springer, 2011.
- [28] A. Girard, C. Le Guernic, Efficient reachability analysis for linear systems using support functions, in: Proc. IFAC World Congress, 2008.
- [29] S. Skogestad, I. Postlethwaite, Multivariable Feedback Control: Analysis and Design, John Wiley & Sons, 2005.
- [30] A. Girard, Reachability of uncertain linear systems using zonotopes, in: M. Morari, L. Thiele (Eds.), HSCC, Vol. 3414 of Lecture Notes in Computer Science, Springer, 2005, pp. 291–305.