Simultaneous Solving of Batched Linear Programs on a GPU

Amit Gurung
National Institute of Technology Meghalaya
Shillong, Meghalaya
amitgurung@nitm.ac.in

Rajarshi Ray
National Institute of Technology Meghalaya
Shillong, Meghalaya
rajarshi.ray@nitm.ac.in

ABSTRACT

Linear Programs (LPs) appear in a large number of applications. Offloading the LP solving tasks to a GPU is viable to accelerate an application's performance. Existing work on offloading and solving an LP on a GPU shows that performance can be accelerated only for large LPs (typically 500 constraints, 500 variables and above). This paper is motivated from applications having to solve small LPs but many of them. Existing techniques fail to accelerate such applications using GPU. We propose a batched LP solver in CUDA to accelerate such applications and demonstrate its utility in a use case - state-space exploration of models of control systems design. A performance comparison of The batched LP solver against sequential solving in CPU using the open source solver GLPK (GNU Linear Programming Kit) and the CPLEX solver from IBM is also shown. The evaluation on selected LP benchmarks from the Netlib repository displays a maximum speed-up of 95× and 5× with respect to CPLEX and GLPK solver respectively, for a batch of $1e5$ LPs.

KEYWORDS

Linear programming, Batched linear programs, GPU, CUDA, Simplex method

ACM Reference Format:
Amit Gurung and Rajarshi Ray. 2019. Simultaneous Solving of Batched Linear Programs on a GPU. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297663.3310308

1 INTRODUCTION

A linear program (LP) is an optimization problem with a linear cost function subject to a search space defined by a conjunction of linear constraints. The size of a LP is measured by the number of decision variables and the number of linear constraints that it contains. LPs appear in a variety of applications. This work is motivated from applications that require solving a large number of LPs of small size (less than 500 variables and 500 constraints). Traditional CPU computations are now increasingly being carried out on CPU-GPU heterogeneous systems, by offloading data parallel tasks to a GPU for accelerating performance. We propose a hybrid CPU-GPU LP solver which can solve a batch of many LPs simultaneously. Our work assumes the setting that computations begin in a CPU where LPs are created, batched and then offloaded to a GPU for an accelerated solution. The solutions are transferred back to the CPU from the GPU for further processing. As an example, we show an application of our batched LP solver in an algorithm for state-space exploration of models of control systems. In model-based design, state-space exploration is a standard analysis technique. State of the art methods and tools for state-space exploration heavily rely on solving many independent LPs [8, 24]. Moreover, the LPs are generally of small size. We show that using our batched LP solver, the state-space exploration tools can improve their performance significantly. Prior work on solving a LP on a GPU and on multi-GPU architectures are many [3, 16, 17, 23, 26]. The focus of all such works has been on methods to improve the performance of algorithms to solve one LP. Performance gain is reported generally when offloading large LPs of size 500 (500 constraints, 500 variables) and above [3, 17, 26]. It is seen that for small LPs, the time spent in offloading the problems from CPU to GPU memory is more than the time saved with parallel execution in the GPU. Although modern LP solvers like CPLEX [5] and GLPK [19] are very efficient in solving small LPs, solving many of them one by one may consume considerable time. Note that using any of the prior work to solve a LP on GPU will not accelerate our target applications since they perform well only on large LPs. We show that with batched computation, performance acceleration can be achieved even for small LPs (e.g. LPs of size 5) for a considerably large *batch size*, where *batch size* refers to the number of LPs in a batch. We present a CUDA C++ implementation of a solver which implements the simplex method [6], with an effort to keep coalescent memory accesses, efficient CPU-GPU memory transfer and an effective load balancing. To the best of our knowledge, this is the first work in the direction of batched LP solving on a GPU. The solver source can be found at https://bitbucket.org/rajgurung777/simplexprojects. 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 [19], an open source LP solver and the CPLEX solver from IBM. The evaluation on selected LP benchmarks from the Netlib repository displays a maximum speed-up of 95× and 5× with respect to CPLEX and GLPK solver respectively, for a batch of $1e5$ LPs. In addition, we consider a special class of LPs with feasible region as an hyper-rectangle and exploit the fact that these can be solved cheaply without using the simplex algorithm. We implement this special case LP solver as part of the solver.

2 MOTIVATING APPLICATION

In model-based design of control systems, a standard technique of analysis is to compute the state-space of the model using exploration algorithms. Properties of the control system such as safety and stability are analyzed by observing the computed state-space. In this section, we consider two open-source tools that perform state-space exploration of control systems with linear dynamics,

ICPE '19, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6239-9/19/04…$15.00
https://doi.org/10.1145/3297663.3310308

namely SpaceEx [8] and XSpeed [24]. These tools can analyze systems modeled using a mathematical formalism known as *hybrid automaton* [2]. A conservative over-approximation of the exact state space is computed by both the tools. The state of the art state-space exploration algorithm in these tools compute the state-space as a union of convex sets, each having a symbolic representation, known as the support function representation [10]. The algorithm requires a conversion of these convex sets from its symbolic support function representation to concrete convex polytope representation, in order to have certain operations efficient. This conversion involves solving a number of linear programs. Moreover, the precision of the conversion and consequently, the precision of the computed state-space depends on the number of LPs solved. Table 1 shows the number of LPs and its dimension that these tools solve for a fairly accurate state-space computation over a time horizon of just 100 seconds, on some standard control systems benchmarks.

Benchmark	LP Dimension	Total LPs
Fourth Order Filtered Oscillator	6	$7.2e7$
Eight Order Filtered Oscillator	10	$2.0e8$
Helicopter Controller	28	$1.568e9$

Table 1: A large number of LP solving is required for a fairly accurate state-space computation.

We see that the number of LPs to be solved in the above examples is in the order of $1e9$ which cannot be solved in practical time limits even by the fast modern LP solvers like GLPK or CPLEX, when solved sequentially. For instance, to compute the state-space of a Filtered Oscillator model using the tool XSpeed, it requires solving $7.2e7$ LPs [11, 24]. Note that although a solver like CPLEX take approximately 0.003 seconds to solve an LP of dimension 32 in a modern CPU, it will take nearly 60 hours to solve $7.2e7$ LPs sequentially. Therefore, we believe that there is a need to accelerate applications where such bulk LP solving is necessary.

3 LINEAR PROGRAMMING

A linear program in *standard form* is maximizing an objective function under the given set of linear constraints. The objective function is denoted as $\sum_{j=1}^{n} c_j x_j$ and the set of linear constraints is given by $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ and $x_j \geq 0$, for $i = 1, 2, ..., m$ and $j = 1, 2, ..., n$. The inequality $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ is the set of m constraints over n variables and $x_j \geq 0$ is 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:

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

An algorithm that solves LP problems efficiently in practice is the *simplex method* described in [6, 15]. The variables on the left-hand side of the Equation (1) are referred 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 zero. 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. In the first phase, 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 zero. If a zero optimal is found then it implies that the original LP has a feasible solution and the simplex method initiates the second phase. In the second phase, the feasible slack form obtained from the first phase is considered and the original objective function is restored with appropriate substitutions and elimination of the artificial variables. The simplex algorithm is then employed to solve the LP.

Prior to the simplex method, many LP solvers apply preconditioning techniques such as a simple geometric mean scaling in combination with equilibration to reduce the condition number of the constraint matrix in order to decrease the computational effort for solving an LP [7, 18, 22, 27]. In this work, we do not apply any pre-conditioning on the LP for simplicity and use the simplex algorithm described in the following section.

3.1 The Simplex Algorithm

The simplex algorithm is an iterative process of solving a LP. 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* [1]. 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. In our implementation, we consider a tableau of size $p \times q$, where $p = m+1$ and $q = n + sum\ of\ slack\ and\ artificial\ variables + 2$. The $(m+1)$th row stores the best solution to the objective function found until the last iteration, along with the coefficients of the non-basic variables in the objective function.

<table>
<tr><td>Index</td><td>b</td><td>x₁ x₂ ... x_n</td><td>x_{n+1} x_{n+2} ... x_{n+m}</td><td>a₁ a₂ ... a_s</td></tr>
<tr><td>Index of basic variables</td><td>Bounds of the constraints</td><td>Coefficients of non-basic variables</td><td>Coefficients of slack variables</td><td>Coefficients of artificial variables</td></tr>
<tr><td>unused</td><td>Optimal Solution</td><td colspan="3">Coefficients of non-basic variable in objective function (used to determine entering variable)</td></tr>
</table>

Figure 1: Formation of the Simplex Tableau.

There are two auxiliary columns, the first column stores the index of the basic variables and the second stores b_i's of equation (1). Figure 1 shows a schematic of the simplex tableau.

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 ($b_i / -a_{i,e}$), 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 ($b_i / -a_{i,e}$) 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 A GPU

We present our CUDA implementation that solves batched LPs in parallel on a GPU. In this discussion, we shall refer a CPU by *host* and a GPU by *device*. The LP batching is performed on the host and transferred to the device. Our solver implementation assumes that all the LPs in a batch are of the same size. The batch size is adjustable, depending on the device memory size and LP size. Our batching routine considers the maximum batch size that can be accommodated in the device memory.

4.1 Memory Transfer and Load Balancing

First, we allocate device memory (global memory) from the host, that is required for creating a simplex tableau for the LPs in the batch. The maximum number of LPs that can be batched depends on the size of the device global memory in the device. 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 speed-up 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 3). The LP batching routine is shown in Algorithm 1. Lines 2 to 5 computes the basic operations such as obtaining the available size of global memory in GPU, memory requirement of an LP, number of threads for an LP and the number of LPs in a batch. Line 7 determines the number batches required to execute the GPU kernel (line 15). Lines 8 to 14 shows the computation of appropriate indices from the data-structure listLP, so that the device memory *devLP* can load appropriate batches of LP. The GPU kernel modifies the tableau to obtain solutions using the simplex method for every LP in the batch and copies back from the device to the host memory (referred to as D2H-res in Figure 3). We discuss further on our CPU-GPU memory transfer using CUDA streams for efficiency in Section 4.4.

Algorithm 1 Batching Routine: N – the number of LP problems present in the data structure *listLP*. Computed results are returned in R

```
1:  procedure BATCHING(N, listLP, R)
2:      gpuMem ← MemSize()          ▷ GPU's global memory size
3:      lpSize ← LPSize()           ▷ get memory requirement per LP
4:      threadSize ← ThreadSize()   ▷ computes the appropriate thread size based on LP dimension
5:      batchSize = (gpuMem − codeSize) ÷ lpSize   ▷ codeSize is kernel code size
6:      if N > batchSize then
7:          batches = ceil(N ÷ batchSize)
8:          for i = {0, ..., (batches − 1)} do
9:              start = i ∗ batchSize
10:             if i == (batches − 1) then
11:                 end = N − 1
12:             else
13:                 end = start + batchSize − 1
14:             devLP ← copy listLP from index (start to end)
15:             batchKernel(batchSize, threadSize, devLP, R)
16:     else
17:         devLP ← copy listLP from index (1 to N)
18:         batchKernel(batchSize, threadSize, devLP, R)
```

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 tableaux in parallel, introducing another level of parallelism.

4.2 Implementation of the Simplex Algorithm

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 parallelize **Step 1** by utilizing n (out of j) threads in parallel to determine the pivot column using **parallel reduction** described in [13]. 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 from an array of data. We have implemented a parallel reduction by using two auxiliary arrays, one for storing the data and the other for storing the array indices of the corresponding data. The result of a parallel reduction provides us the maximum value in the data array and its corresponding index in the indices array.

We also apply 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). It involves finding a minimum positive value from an array 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 the performance due to warp divergence. Even if we re-size the array to store only the positive values, the kernel still contains conditional statements to check the threads that need to process this smaller size array. To overcome performance degradation with conditional statements, we substituted a large positive number in place of ratios that are negative or undefined. This creates an array 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 in the block).

4.3 Coalescent Memory Access

In this section, we discuss our efforts on keeping a coalescent access to global memory to reduce performance loss due to cache misses. When threads in a warp access contiguous locations in the memory, the access is said to be coalescent. A coalescent memory access results in performance benefits due to an increased cache hit rate.

As discussed earlier, we use global memory to store the simplex tableaux of the LPs in a batch as described in Section 4.1 (Since the global memory in a GPU is of the maximum size in the memory hierarchy, it can accommodate many tableaux). We store the simplex tableau in memory as a two-dimensional array. High level languages like C and C++ use the row-major order by default for representing a two-dimensional array in the memory. CUDA is an extension to C/C++ and also use 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 elements of all rows from a specific column are accesses 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 access elements from the memory separated by the size equal to the column-width of the array. When elements of a specific row are accessed simultaneously by threads 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 access data from contiguous locations in the memory.

We now show 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 ensure more coalesced memory access in comparison to having a row-major storage.

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 from the last row. This requires a row-operation and as mentioned in Section 4.2, we use parallel reduction using two auxiliary arrays, *Data* and *Indices*. 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 use the technique of *Parallel Reduction: Sequential Addressing* in [13], a technique that ensures coalesced memory access.

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 $(b_i/-a_{i,e})$, as described in Section 3.1. This requires two column-operations involving the access to all elements from columns b and x_e as shown in Figure 2. To compute the row index with the minimum positive ratio, we use parallel reduction as described above in Section 4.2. Our tableau being stored in a column-major order, access to columns b and x_e 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 [13] for coalesced memory access. **Step 3** performs the pivot operation that updates

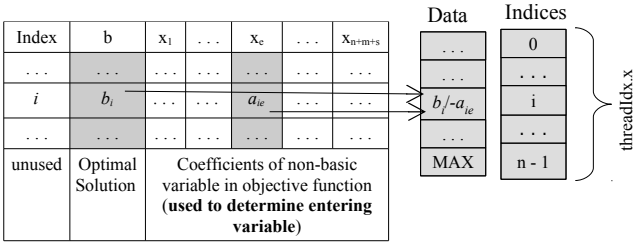

Figure 2: Simplex Tableau along with two separate arrays, *Data* to store the positive ratio and *Indices* to keep track of the indices of the corresponding values in the *Data* array. Ratios that reduces to negative or undefined are replaced by a large value denoted by MAX.

the elements of the simplex tableau and is the most expensive of the three steps. It first involves a non-coalescent 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 at 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 $NewRow_{ij} = OldRow_{ij} - PivotCol_{ie} * NewPivotRow_{\ell j}$. 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 parallelize the outer for-loop that maps the rows of the simplex tableau. Our data being represented in a column-major order, parallel access to all rows for each element in the j^{th} column of the inner for-loop is coalesced.

To verify the performance gain due to coalesced memory access, we experiment with **Step 3** which is the most expensive of the three steps in the simplex algorithm, by modifying it to have non-coalesced memory accesses. We interchange the inner for-loop with the outer loop (loop interchange, a common technique to improve cache performance [14]). This loop interchanging converts the Step 3 to have non-coalesced memory access since our simplex

tableau is represented in a column-major order. Table 2 presents the experimental results to show the gain in performance when the accesses to memory are coalesced, as compared to having non-coalesced memory accesses. The results show a significant gain in performance on a Tesla K40c GPU, for LPs with the initial basic solution as feasible.

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

Table 2: Performance due to coalesced/non-coalesced memory access on a GPU, LPs with feasible initial basic solution.

We observe that Step 1 has a row-operation, Step 2 has two column-operations and Step 3 has a row and a column operation each with a nested for-loop which can be expressed both in row as well as column operations. We see that there are more column operations than row operations.

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 **nvprof** [20] to profile time for memory transfer and kernel operation of our implementation discussed above in Section 4. The result of profiling in a Tesla K40c GPU, for LPs with an initial basic solution as feasible, is reported in Table 3. We observe that, for a small batch size (e.g. 10 in the Table 3), the memory copy operation takes a maximum of 5.75% of the execution time, whereas for larger batch size (rows in gray colour), the memory copy operation takes 6% to 15% of the execution time.

LP Dimension	Batch-size	Time %			
		Kernel	MemCpy		Total %
			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

Table 3: Profiling report obtained using nvprof tool for LPs with an initial basic solution as feasible. H2D - stands for host to device and D2H indicates device to host memory copies respectively.

A standard technique to improve the performance in CPU-GPU heterogeneous applications is by using CUDA streams. CUDA streams allow overlapping memory copy operation with kernel execution. A stream in CUDA consists of a sequence of operations executed on the device in the order in which they are issued by the host procedure. These operations can not only be executed in an interleaved manner, but also be executed concurrently in order to gain performance [12].

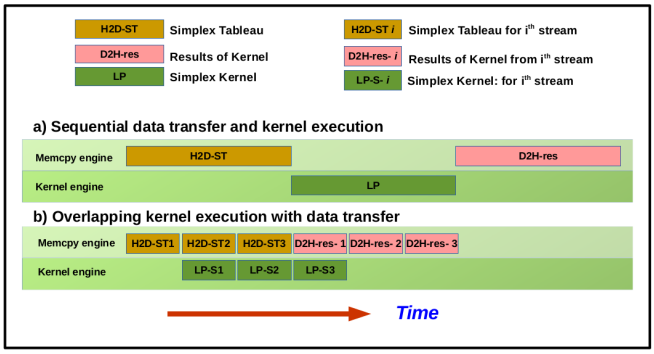

Figure 3: Performance gain due to overlapping kernel execution with data transfer compared to sequential data transfer and kernel execution. The time required for host-to-device (H2D), device-to-host (D2H) and kernel execution are assumed to be the same.

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 are performed by the copy engine. However, some GPUs have two copy engines, one each for copying data from host to device and from device to host, for better performance. Figure 3 illustrates the overlapping of kernel executions with memory copy operation, when the GPU has one kernel and one copy engine. Streaming by batching similar operations causes more overlap of copies with kernel executions, as depicted in the figure. Adding all host-to-device copy to the CUDA streams followed by all kernel launches and device-to-host data copies, can result in a significant overlap of memory copy operations with kernel executions, resulting in a performance gain. When there are two 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 may result in a better performance than the former method. For GPUs with compute capability 3.5 and above, both the methods result in the same performance due to the Hyper-Q [4] feature.

Although a large number of CUDA streams achieves more concurrency and interleaving among operations, it incurs 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 the batch size and the LP dimension, the optimal number of streams also varies. In this paper, we report results with 10 streams for batch size more than 100 LPs. We use a single stream when the batch size is less than 100 (for LPs of any dimension).

4.5 Limitations of the Implementation

As the current limit on threads per block is 1024 in CUDA, our implementation limits the size of LPs having *initial basic solution as feasible* to 511×511. The size limit for LPs having *initial basic solution as infeasible* is 340×340. This limit is derived from (2):

$$(var + slack + arti + 2) \leq 1024 \tag{2}$$

where var is the number of variables (dimension of the LP problem), $slack$ is the number of slack variables and $arti$ is the number of artificial variables in the given LP. The number 2 indicates the use of two auxillary columns described in Section 3.1.

4.6 Solving a Special Case of LP

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 (3) shows that maximizing the objective function is the sum of the dot products of n terms.

$$\underset{x \in \mathcal{B}}{\text{maximize}}(\ell.x) = \sum_{i=1}^{n} \ell_i.h_i, \text{where } h_i = \begin{cases} a_i & \text{if } \ell_i < 0 \\ b_i & \text{otherwise} \end{cases} \tag{3}$$

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

An implementation for solving this special case of LPs is incorporated in our solver. 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. A preliminary introduction to this technique is introduced in the paper [24].

5 EVALUATION

We evaluate our solver on two set of LPs. The first is a set of randomly constructed LPs. The LPs in this set are constructed by randomly selecting the coefficients of the constraint matrix A from a range of [1…1000], the bounds of the constraints b from a range of [1…1000] and the coefficients of the objective functions c from the range of [1…500] respectively. The second set of LPs are selected from the Netlib repository. On both the set of LPs, we evaluate the performance of our batched solver on varying batch sizes. In the text that follows, we refer to our solver on a GPU as BLPG, abbreviating Batched LP solver on a GPU. The solver using CUDA streams is referred as BLPG-SM. We perform our experiment in Intel Xeon E5-2670 v3 CPU, 2.30 GHz, 12 Core (without hyper-threading), 62 GB RAM with Nvidia's Tesla K40c GPU. The reported performance is an average over 10 runs. A performance comparison of our solver to GLPK is shown in Figure 4, for the first set of randomly generated LPs of various sizes and various batch sizes. We observe a maximum speed-up of 16× for LPs having the *initial basic solution as feasible*, for a batch of 2K LPs of dimension 100. For the same type of LPs, we see a maximum speed-up of 18×, for a batch of $5K$ LPs of dimension 100, using BLPG-SM. We observe that for LPs of large size, BLPG performs better even with a few LPs in parallel (e.g., batch size=50 for a 500 dimensional LP). However, for small size LPs, BLPG out-performs GLPK only for larger batch sizes (e.g. a batch size of 100 for a 5 dimensional LP).

Batch-size	Time (Sec)			Speed-up w.r.t. GLPK	
	GLPK	BLPG	BLPG-SM	Vs BLPG	Vs BLPG-SM
1	0.000	0.000	0.000	0.00	0.00
50	0.001	0.000	0.000	0.00	5.00
100	0.001	0.000	0.001	12.00	1.09
500	0.009	0.001	0.002	9.40	4.48
1000	0.016	0.003	0.003	5.43	4.94
1500	0.031	0.009	0.004	3.44	7.75
2000	0.041	0.005	0.006	8.20	6.83
5000	0.104	0.013	0.014	8.00	7.43
10000	0.166	0.025	0.025	6.64	6.64
20000	0.317	0.049	0.048	6.47	6.60
50000	0.775	0.122	0.120	6.35	6.46
100000	1.663	0.242	0.239	6.87	6.96

(a) 5-Dimension

Batch-size	Time (Sec)			Speed-up w.r.t. GLPK	
	GLPK	BLPG	BLPG-SM	Vs BLPG	Vs BLPG-SM
1	0.002	0.003	0.004	0.67	0.50
50	0.099	0.013	0.040	7.62	2.48
100	0.178	0.018	0.018	9.89	9.89
500	0.681	0.070	0.050	9.73	13.62
1000	1.628	0.105	0.095	15.50	17.14
1500	2.400	0.153	0.153	15.69	15.69
2000	3.283	0.200	0.184	16.42	17.84
5000	7.900	0.486	0.435	16.26	18.16
10000	15.695	0.956	0.860	16.42	18.25
20000	31.283	1.904	1.714	16.43	18.25
50000	78.280	4.778	4.277	16.38	18.30

(b) 100-Dimension

Batch-size	Time (Sec)			Speed-up w.r.t. GLPK	
	GLPK	BLPG	BLPG-SM	Vs BLPG	Vs BLPG-SM
1	0.045	0.074	0.046	0.61	0.98
50	2.133	0.234	0.172	9.12	12.40
100	4.242	0.339	0.274	12.51	15.48
500	21.395	1.480	1.309	14.46	16.34
1000	43.084	2.844	2.572	15.15	16.75
1500	62.819	4.295	3.846	14.63	16.33
2000	75.288	5.730	5.143	13.14	14.64

(c) 500-Dimension

Figure 4: Performance of batched LPs of type initial basic solution as feasible.

Figure 5 shows a performance comparison of BLPG with GLPK on LPs having the initial basic solution as infeasible. In spite of the fact that BLPG executes the kernel twice due to the two-phase simplex algorithm as discussed in Section 3 (an extra overhead of data exchange between the two kernels), we observe a better performance. We gain a maximum speed-up of nearly 12×, for a batch of 10K LPs of dimension 200.

On profiling BLPG-SM, we observe that for small sized LPs, the processing time of the kernel is much larger than the data transfer time. As a result, the gain in performance due to overlapping data transfer with kernel execution is negligible. This is evident from the results in Figure 4a. As the LP size increases, the volume of data transfer also significantly increases. 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 streams. This results in performance gain up to 2% to 3% for LPs of large sizes, as evident from Figures 4b and 4c.

Batch-size - ->			1	5	10	100	500	1000	5000	7000	10000	50000	70000	100000
Sl No	Benchmarks	BLPG Vs	Speed-up											
1	ADLITTLE	IBM CPLEX	0.06	0.31	0.60	4.29	4.58	5.07	5.26	5.68	5.82	5.45	5.48	5.51
2	AFIRO		0.91	4.93	10.76	37.76	73.43	74.84	87.51	89.72	96.12	98.90	93.25	95.21
3	BLEND		CPLEX could not read the input file (SIF format) available in the Netlib repository											
4	ISRAEL		0.09	0.36	0.64	1.74	2.14	2.17	2.21	2.26	2.23	2.17	2.17	2.17
5	SC105		0.16	0.94	1.18	4.52	5.53	5.66	5.87	5.79	5.82	5.58	5.58	5.55
6	SC205		0.04	0.19	0.38	0.67	0.78	0.77	0.85	0.78	1.07	1.07	1.08	1.08
7	SC50A		0.64	3.10	6.56	37.16	46.88	45.28	50.83	52.25	49.45	50.41	50.73	50.75
8	SC50B		0.51	2.71	5.53	29.06	46.23	37.92	40.23	42.86	42.35	40.74	41.01	41.44
1	ADLITTLE	GLPK	0.04	0.17	0.28	1.36	1.40	1.52	1.61	1.61	1.62	1.64	1.65	1.64
2	AFIRO		0.00	0.33	0.67	1.63	2.45	2.14	2.68	2.69	2.93	2.78	2.81	2.89
3	BLEND		0.05	0.10	0.20	0.89	1.34	1.35	1.42	1.42	1.42	1.43	1.40	1.39
4	ISRAEL		0.09	0.31	0.52	1.27	1.55	1.59	1.61	1.63	1.60	1.56	1.57	1.56
5	SC105		0.20	0.67	0.79	2.30	2.51	2.57	2.62	2.63	2.63	2.48	2.49	2.49
6	SC205		0.07	0.24	0.43	0.89	0.92	1.01	1.02	1.02	1.23	1.22	1.22	1.21
7	SC50A		0.20	1.00	0.80	5.00	5.43	5.62	5.75	5.74	5.82	5.82	5.75	5.81
8	SC50B		0.17	0.83	0.67	4.36	4.69	4.78	4.97	4.97	5.05	5.00	5.00	4.97

Table 5: Performance evaluation on selected benchmarks from Netlib

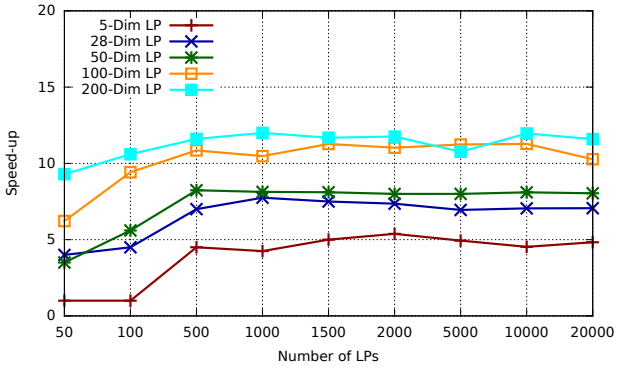

Figure 5: Performance comparison between GLPK and BLPG for LPs with initial basic solution as infeasible

Batch-size -->			1	10	100	1000	10000	100000
Benchmark	Rows	Cols	Gflop/s					
ADLITTLE	71	97	0.09	0.92	7.51	8.27	8.81	8.84
AFIRO	35	32	0.10	0.98	7.64	10.01	11.19	11.27
BLEND	117	83	0.15	1.34	6.17	9.90	10.47	10.48
ISRAEL	174	142	0.57	4.46	12.46	15.35	15.61	15.61
SC105	150	103	0.48	3.73	13.87	15.48	15.72	15.74
SC205	296	203	0.95	7.18	13.67	15.11	16.94	16.93
SC50A	70	48	0.22	2.20	13.14	14.69	15.52	15.56
SC50B	70	48	0.22	2.17	13.04	14.52	15.40	15.47

Table 4: Performance of batched LP solver on a GPU

Table 5 shows a comparative evaluation of our solver to CPLEX and GLPK, on a set of selected LPs from the Netlib repository. The experimental platform is a 4 Core Intel Xeon CPU E5-1607 v4, 3.10 GHz, 63 GB RAM with Nvidia's Tesla K40m GPU. The LP benchmarks in the Netlib repository are present in MPS format. We use the MATLAB's built-in function *mpsread* to read the benchmarks and then convert them into the standard form. The converted sizes of the benchmarks is shown in Table 4 with column heading "Rows" indicating the number of converted constraints and "Cols" indicating the size of the benchmark. The table also shows the number of floating point operations per second (in Giga flops), giving an estimation of the floating point computations in the Simplex algorithm and the utilization of GPU by our proposed batched LP solver. We use the visual profiler (nvvp) available in the CUDA Toolkit [21]. The batched LP solver gives a maximum of 16.93 Gflops/s for a batch size of 100K LPs on a Nvidia Tesla K40m GPU that has a theoretical peak of 1.43 Tflops/s, for double precision arithmetic.

Due to the LP size limitation of BLPG discussed earlier in Section 4.5, we choose benchmarks that satisfy this limitation. Table 5 shows the performance comparison of the two LP solvers with that of BLPG, on the Netlib benchmarks. Note that Netlib benchmarks are highly sparse in nature and LP solvers such as IBM CPLEX and GLPK are optimized for sparse LPs. Our implementation is the original Simplex method proposed by Dantzig and does not have any optimization for sparse LPs. We observe that in some benchmarks such as **SC205** in Table 5 (and some others, not included in the table) CPLEX performs better than GLPK, whereas in other benchmark instances, GLPK outperforms CPLEX. Our proposed BLPG achieves a maximum of 95× and nearly 6× speed-up on the Netlib benchmarks w.r.t. CPLEX and GLPK respectively.

5.1 Motivational Application

In this section, we demonstrate the performance enhancement of state-space exploration of models of control systems, using our batched LP solver. As discussed in Section 2, the state-space exploration routines in the state of the art tools requires solving a large number of LPs. We consider two benchmarks, the Helicopter controller and a Five dimensional dynamical system for its state-space computation using the tools SpaceEx-LGG and XSpeed. The Helicopter controller benchmark is a model of a twin-engined multi-purpose military helicopter with 8 continuous variable modeling the motion and 20 controller variables that governs the various controlling actions of the helicopter [8, 25]. The Five dimensional dynamical system benchmark is a model of a five dimensional linear continuous system as defined in [9]. We direct the reader to the paper [9] for details on the dynamics of the model.

In order to show the impact of our solver BLPG, we evaluate the performance of the tools by solving the resulting LPs generated from the state-space exploration routines on these benchmarks. The LPs are sequentially solved using GLPK and in parallel using BLPG. The performance comparison is shown in Table 6, in an experimental setup of Intel Q9950 CPU, 2.84 Ghz, 4 Core (no hyper-threading), 8 GB RAM with a GeForce GTX 670 GPU. We observe a maximum speed-up of 12× and 9× in the tools performance with parallel solving of the LPs in BLPG w.r.t. sequential solving in GLPK. When compared to the tool SpaceEx-LGG, we observe a maximum of 54× and 39× speed-up in XSpeed using our solver. Note that the LPs generated by the tool on these benchmarks have the property that their feasible region is an hyper-rectangle. Therefore, BLPG solves these using the technique mentioned in Section 4.6.

Benchmark	Nos. of LPs	Time (in Secs)			Speed-up w.r.t. XSpeed (GPU)	
		XSpeed (Seq)	SpaceEx	XSpeed (GPU)	Vs XSpeed (Seq)	Vs SpaceEx
Five Dimensional System	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 6: Speed-up in XSpeed using Hyperbox LP Solver

5.2 Future Work

As the current limit on threads per block is 1024 in CUDA, our implementation limits the size of LPs having *initial basic solution as feasible* to 511×511 and 340×340 for LPs having *initial basic solution as infeasible*. We intend to address this limitation in future work. Another limitation of the solver is the LPs in the batch have to be the same size. Although this limitation is not a concern for our motivational application in particular, but the solver will be useful in more general applications without this limitation.

6 CONCLUSION

Solving a linear program on a GPU for an accelerated performance on a CPU-GPU heterogeneous platform has been extensively studied. To the best of our knowledge, all such work report a performance gain only on linear programs of large size. We present a solver implemented in CUDA that can accelerate applications having to solve small to medium size LPs, but a large number of them. Our solver batches the LPs in an application and solves them in parallel on a GPU using the simplex algorithm. We report significant performance gain on benchmarks in comparison to solving them in CPU using GLPK and CPLEX solvers. We show the utility of our solver in an application of state-space exploration of models of control systems which involves solving many small to medium size LPs, by showing significant performance improvement.

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. The authors thank Santibrata Parida, for his help in experimental evaluations.

REFERENCES

[1] Ilan Adler, Christos Papadimitriou, and Aviad Rubinstein. 2014. On simplex pivoting rules and complexity theory. In *International Conference on Integer Programming and Combinatorial Optimization*. Springer, 13–24.
[2] Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. 1993. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*. Springer, 209–229.
[3] Jakob Bieling, Patrick Peschlow, and Peter Martini. 2010. An efficient GPU implementation of the revised simplex method. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 1–8.
[4] Thomas Bradley. 2012. Hyper-Q Example. http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf
[5] IBM ILOG CPLEX. 2009. V12. 1: UserâĂŹs Manual for CPLEX. *International Business Machines Corporation* 46, 53 (2009), 157.
[6] George B. Dantzig and Mukund N. Thapa. 1997. *Linear Programming 1: Introduction*. Springer.
[7] Joseph M Elble and Nikolaos V Sahinidis. 2012. Scaling linear optimization problems prior to application of the simplex method. *Computational Optimization and Applications* 52, 2 (2012), 345–371.
[8] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable Verification of Hybrid Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV) (LNCS)*, Shaz Qadeer Ganesh Gopalakrishnan (Ed.). Springer.
[9] Antoine Girard. 2005. Reachability of Uncertain Linear Systems Using Zonotopes. In *HSCC (Lecture Notes in Computer Science)*, Manfred Morari and Lothar Thiele (Eds.), Vol. 3414. Springer, 291–305.
[10] Antoine Girard and Colas Le Guernic. 2008. Efficient Reachability Analysis for Linear Systems using Support Functions. In *Proc. IFAC World Congress*.
[11] Amit Gurung, Rajarshi Ray, Ezio Bartocci, Sergiy Bogomolov, and Radu Grosu. 2018. Parallel reachability analysis of hybrid systems in XSpeed. *International Journal on Software Tools for Technology Transfer* (2018), 1–23.
[12] Mark Harris. [n. d.]. How to Overlap Data Transfers in CUDA C/C++. http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/
[13] Mark Harris. 2007. Optimizing parallel reduction in CUDA. *NVIDIA Developer Technology* (2007). http://vuduc.org/teaching/cse6230-hpcta-fa12/slides/cse6230-fa12--05b-reduction-notes.pdf
[14] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.
[15] Victor Klee and George J Minty. 1970. *How good is the simplex algorithm*. Technical Report. WASHINGTON UNIV SEATTLE DEPT OF MATHEMATICS.
[16] Mohamed Esseghir Lalami, Vincent Boyer, and Didier El-Baz. 2011. Efficient implementation of the simplex method on a CPU-GPU system. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 1999–2006.
[17] Mohamed Esseghir Lalami, Didier El-Baz, and Vincent Boyer. 2011. Multi GPU implementation of the simplex algorithm. In *2011 IEEE International Conference on High Performance Computing and Communications*.
[18] T Larsson. 1993. On scaling linear programs–Some experimental results. *Optimization* 27, 4 (1993), 355–373.
[19] Andrew Makhorin. 2009. GNU Linear Programming Kit, v.4.37. http://www.gnu.org/software/glpk.
[20] Nvidia. 2015. CUDA C Programming Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
[21] Nvidia. 2017. CUDA Toolkit Documentation v8.0 (2017). *Nvidia Corporation* (2017).
[22] Nikolaos Ploskas and Nikolaos Samaras. 2015. A computational comparison of scaling techniques for linear optimization problems on a graphical processing unit. *International Journal of Computer Mathematics* 92, 2 (2015), 319–336.
[23] Nikolaos Ploskas and Nikolaos Samaras. 2015. Efficient GPU-based implementations of simplex type algorithms. *Appl. Math. Comput.* 250 (2015), 552–570.
[24] Rajarshi Ray, Amit Gurung, Binayak Das, Ezio Bartocci, Sergiy Bogomolov, and Radu Grosu. 2015. XSpeed: Accelerating Reachability Analysis on Multi-core Processors. In *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings (Lecture Notes in Computer Science)*, Nir Piterman (Ed.), Vol. 9434. Springer, 3–18. https://doi.org/10.1007/978-3-319-26287-1_1
[25] Sigurd Skogestad and Ian Postlethwaite. 2005. *Multivariable Feedback Control: Analysis and Design*. John Wiley & Sons.
[26] Daniele G. Spampinato and Anne C. Elster. 2009. Linear optimization on modern GPUs. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 1–8. https://doi.org/10.1109/IPDPS.2009.5161106
[27] John A Tomlin. 1975. On scaling linear programming problems. *Computational practice in mathematical programming* (1975), 146–166.