Dionysia: Accelerating the Simplex Algorithm

Ben Ewing and Kyle Finke, *Duke University*

***Abstract*—Linear programming is an approach used to optimize linear objective functions with linear constraints. While existing software solutions are quite advanced, the use of linear programming in many latency sensitive industries demonstrates the need for research into hardware accelerated solutions. We focus on a specific algorithm for solving linear programs called Simplex, which represents optimization problems as matrices, and searches for solutions with a series of row and column transformations. To translate this algorithm to hardware we designed Dionysia: a parametrizable architecture inspired by the actor concurrency model, which takes advantage of the structure of the Simplex algorithm to solve linear programs with a high degree of parallelism and minimal data movement.**

***Index Terms*—Simplex, Hardware Accelerator, Field-Programmable Gate Array (FPGA), Architecture, Chisel, CPLEX, Spatial, Linear Programming, Amazon Web Services (AWS)**

I. Introduction

LINEAR programming (LP) has applications in a wide range of industries, from archaeological simulations to high voltage electricity transmission [1], [7]. As general purpose hardware has grown faster and memory has grown larger, linear programs have grown to the scale of tens of thousands of variables and tens thousands of constraints (and beyond). While free and open source solutions exist, commercial solvers such as IBM's CPLEX and Gurobi's Gurobi Optimizer are required to handle larger LPs efficiently. Unfortunately these command annual license fees from $2000 to $12000 [8], and to truly take advantage of these software packages the user will also need to invest in high-end hardware.

These high costs provided the initial motivation necessary to explore a potential alternative hardware accelerator solution for LP solving. A hardware approach was chosen as there appears to be very little research into hardware acceleration of LP solvers. While most software solutions make use of a variety of optimization algorithms we will focus on the Simplex algorithm, which provides good average runtime and can be largely parallelized. Additionally, we use AWS EC2 F1 platform to build and deploy our compiled Simplex hardware for use on an FPGA. This paper will proceed as follows: Section II will outline the Simplex Algorithm and our motivations for choosing it, Section III will outline Dionysia's architecture, Section IV will provide simulated benchmarks, and Section V will outline next steps (there are many).

II. Linear Programming and the Simplex Algorithm

At first glance the requirement that linear programs use both linear constraints and a linear objective function may seem quite restrictive, however, researchers in many fields have found that even within these constraints, a large number of problems can be modeled. As an example, consider the problem of a student who must allocate hours during a semester between two classes to maximize their GPA, while maintaining their marriage (i.e. not spending every single hour working). One of these classes may be difficult (i.e. it will require many hours), and the other easy. This problem can be expressed as a system of equations:

$$\begin{aligned} \text{Maximize} \quad & 0.5x_1 + 0.5x_2 \\ \text{subject to} \quad & 0.67x_1 + 0.9x_2 \leq 100 \\ & 0x_1 + 0.32x_2 \leq 30 \\ & x_1 \geq 0 \\ & x_2 \geq 0. \end{aligned} \tag{1}$$

A problem in this form will typically be summarized in matrix form as:

$$\begin{aligned} \text{Maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0. \end{aligned} \tag{2}$$

In this example $\mathbf{c} = [0.5, 0.5]$, $\mathbf{x} = [x_1, x_2]$, $\mathbf{A} = \left[\begin{smallmatrix} 0.67 & 0.9 \\ 0 & 0.32 \end{smallmatrix}\right]$, and $\mathbf{b} = [100, 30]$.

The Simplex algorithm, developed by George Dantzig in the late 1940s, is the first algorithm for solving this type of constrained optimization [@dantzig]. While faster algorithms now exist, Simplex offers an average polynomial runtime ([3]; though problems with exponential runtimes do exist) along with properties that map well to hardware. Simplex first adds additional variables, called slack variables, to allow the inequality constraints to become equality constraints, and then maps the optimization into a matrix, called the tableau. The tableau has the following form (though there are many variations):

$$\begin{bmatrix} \mathbf{A} & \mathbf{I} & \mathbf{b} \\ \mathbf{c} & \mathbf{0} & 0 \end{bmatrix}$$

where $\mathbf{I}$ represents the identity matrix. The example in (2) has the following tableau form:

$$\begin{bmatrix} 0.67 & 0.9 & 1 & 0 & 100 \\ 0 & 0.32 & 0 & 1 & 30 \\ 0.5 & 0.5 & 0 & 0 & 0 \end{bmatrix}$$

.

After building the tableau, the Simplex algorithm proceeds by carrying out a number of simple row-wise operations until stopping conditions are detected: (1) Searching for a pivot column, (2) Searching for a pivot row, (3) Pivoting along the pivot row, (4) Pivoting the remaining rows. These steps repeat until either a pivot row or a column can not be found. It is sometimes possible for the algorithm to stall and get stuck at a sub-optimal solution. Notably, these operations do not change the shape of the tableau, and the row operations and row data are mostly independent of each other, providing many parallelism opportunities.

III. Dionysia: Translating Simplex to Hardware

Dionysia is our hardware implementation of the Simplex algorithm. It is built with Chisel 3.2, a hardware design language with a focus on parametrizable design and the generation of optimized circuits, and targets the AWS EC2 F1 FPGA platform.

In the context of ancient Greece, Dionysia was a festival largely centered around theater. In this spirit our accelerator, Dionysia, is likewise centered around actors, but in the context of the actor-concurrency model. As such, Dionysia consists of four types of modules: the actor and the director carry out the Simplex algorithm, and the core and subcore manage the FPGAs DRAM and communication with the host. A sketch of the director and actor modules can be seen in Figure 1.

The core and subcore modules are responsible for receiveing communcation with the host (including loading data into the FPGA DRAM), loading data into the director module, sending the start signal, and reporting the results of the algorithm back to the host. Future versions of Dionysia, which make better use of FPGA DRAM, will expand the role of the subcore module.

The primary control module in Dionysia is the director. This module is responsible for loading data into the actors, issuing instructions to the actors, and monitoring the state of the Simplex algorithm. The director is loosely-coupled with the subcore: it receives data from the subcore and allocates it to each actor, and then waits for a start signal.

To carry out the Simplex algorithm the director module uses a nite state machine which cycles through the steps algorithm until a stopping condition is detected. During step (1) the director sets the pivot column to be the index of the minimum positive value for the actor that represents the objective function (the bottom row; Actor 2 in Figure 1). During step (2) the director instructs the actors to compute the ratio between their values at the pivot column index and the constraint value (the last value in the row). The director selects the row with the minimum ratio as the pivot actor. Step (3) updates just the pivot row actor. Step (4) sends the pivot actor's data to all other actors so that they can update accordingly.

Dionysia instantiates many identical actor modules, each of which is represents a row of the Simplex tableau. Each actor implements a simple finite state machine, where each state represents an independent operation needed to complete the algorithm. These operations include: finding the minimum value, computing the ratio between the constraint value (the final value in the row) and the value in the pivot column, and updating data per the pivot data. The update operation varies per iteration depending on whether or not the actor represents the pivot row. The final actor (in Figure 1 this is Actor 2) represents the objective function; it is the only actor that needs to use the min operation, and does not need the ratio operation. Currently all actors are implemented identically but creating more specialized modules could improve the area-efficiency of Dionysia.

Actors operate in complete independence from each other, all required instructions and data are passed to each actor from the director module. This allows the actors to act completely in parallel. However this comes at the cost of duplicated operators for each actor, making this module inefficient with respect to FPGA area and limiting the number of actors that can be implemented. We will explore possible modifications to alleviate this issue in Section V.

Each step must be completed before the next may be carried out, however future work may explore how the actors can prepare ahead of time responses to the director for certain steps. Steps (1) and (3) of the algorithm the director only communicates with a single actor and no other useful operations can be carried out by the other rows. For the remaining steps of the algorithm the actors are issued directions in parallel. Actors are not pipelined, they wait for directions issued by the director. In the future some steps may be pipelined, for example, the min and ratio operations can be computed immediately after the update step.

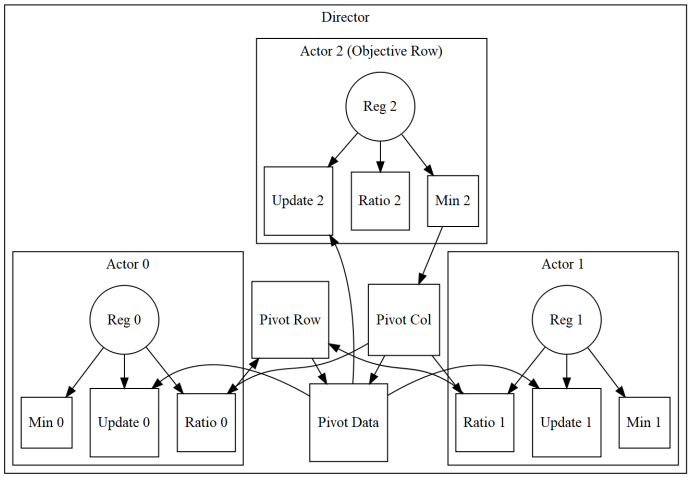

Fig. 1. Dionysia Architecture Sketch

Because this architecture allows actors to operate almost entirely in parallel we achieve strong efficiency: cycle-accurate simulations of Dionysia indicate that each iteration of the Simplex algorithm only requires 25 cycles to complete, regardless of the number of constraints and variables. However, the astute reader will note that the problem size that Dionysia can solve is limited by the number and size of actors. Strategies to work around this limitation will be discussed in Section V.

A. Parametrization

Dionysia's hardware can be generated per a number of parameters. These include the precision of integers (e.g. 64bit,

32bit, etc.), the number of actors (this corresponds to the number of constraints in the LP problem; number of rows in the tableau), the size of the actors (this corresponds to the number of variables in the LP problem; number of columns in the tableau), and the precision of the Simplex algorithm itself.

Rather than directly implementing hardware floating point operations, Dionysia uses the Simplex precision parameter to scale integers appropriately. Before sending data to the FPGA, the host scales each number in the tableau by the precision amount (e.g. 10e+5 for 5 decimal points of precision). Dionysia's output is likewise scaled back down (and cast to floating point). Integer addition and substraction require no changes. Integer multiplication must be scaled down by the precision amount, and integer division must be scaled up by the precision amount. For example, if $a = 1.23$, and $b = 4.56$, and the precision is 100. The scaled values are 123 and 456 respectively. Multiplication is implemented as $\frac{123*456}{100} = 560$ which approximates $1.23 * 4.56 = 5.6088$. Division is implemented as $\frac{123}{456}*100 = 26$ which approximates 0.2697.

While a given instantiation of Dionysia is meant to support a given tableau size, smaller problems can always be supported by larger hardware by adding non-binding constraints to pad the tableau. For an example see equation 3. Because of Dionysia's parallelism no extra cycles are required to solve a padded problem as compared to the equivalent problem on a "natively" sized instance of Dionysia.

$$\begin{bmatrix} 0.67 & 0.9 & 1 & 0 & 0 & 100 \\ 0 & 0.32 & 0 & 1 & 0 & 30 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3}$$

.

B. Using Dionysia

Using Dionysia is quite straightforward: executing the host program will prompt the user to input data for the simplex tableau (row-wise). The size of the input must match the size of the Dionysia implementation, however smaller problems can be padded using non-binding constraints (i.e. rows and columns of 0s).

IV. Validation and Benchmark Evaluation

We benchmark our accelerator against two popular LP solvers, GLPK and IBM's CPLEX, as well as a naïve implementation of Simplex in R. GLPK is currently the most used open source LP solver. While very versatile, most algorithms supported by GLPK are serial. CPLEX is one of the most popular commercial solvers, unlike GLPK it can take advantage of parallelized optimization methods. CPLEX typically scales well across multicore and distributed systems. The R implementation is completely serial, and mostly used to determine how many iterations of the Simplex algorithm are needed to solve any given problem. The R implementation also serves as a strong argument for not implementing your own LP solver. We allow both GLPK and CPLEX to use any optimization algorithm, not just Simplex. We do this to give a direct comparison between Dionysia and state-of the-art algorithms with state-of-the-art implementations. All software is run on a shared server with several Xeon E2640 processors (total of 40 cores) and 512Gb of RAM.

We use use a well-known algorithm for generating LPs to generate hundreds of bounded and feasible problems (i.e. a solution to each of these problems exists and should not cause Simplex to stall) of various sizes to benchmark software in a wide variety of scenarios. These problems are then fed to CPLEX, GLPK, and R; timing data is collected for each. The R implementation also returns the number of Simplex iterations needed to solve a particular problem. From this we estimate the number of cycles (including load cycles) Dionysias would need to solve each particular problem. Solution times are then averaged across all porblems of a particular size. The results of these benchmarks can be seen in Figure 2, where the x-axis represents the log of the size of the tableau, and the y-axis represents the time needed to load the problem and return the solution.

While Dionysia has been running on an AWS F1 FPGA instance, we were unable to fully scale it to the necessary problem sizes for proper benchmarking against the software implementations. However, since computations are done on a row-basis and the size of the row does not impact the clock speed (so long as the entire row fits on the FPGA), we were able to estimate the runtime of our Simplex accelerator on an FPGA for longer problems using the known number of cycles for solving LP problems obtained from our simulations. Thus, our accelerator results in Figure 2 come from cycle-accurate VCS simulations. Using the R implementation of Simplex we count the number of Simplex iterations required to solve a problem, and then calculate the number of cycles required by Dionysia. Using these numbers of cycles for the various benchmarks and the clock speed for our working small FPGA implementation (125MHz) we could estimate runtimes for the given benchmarks (Figure 2).

One limitation of these simulation results is that they do not account for the problem of entire rows not being able to fit in an actor at once. When that occurs, rows must be partially loaded from memory in chunks to the actors thereby increasing the runtime of the accelerator. Thus benchmarks for larger problems are likely overestimating the speed of Dionysia, as additional cycles will be needed to move data in and out of DRAM. Due to time constraints, we did not construct an accelerator which handled multiple loads from memory for individual rows. We constructed problems which loaded all rows to all actors from memory once at the beginning of the Simplex algorithm and then stored the results once at the end of the Simplex algorithm. Larger problems will require intermediate loads and stores which will impact overall performance of the accelerator. Possible approaches to mitigate the impact of the increased reliance on DRAM will be discussed in Section V.

V. Future Work

There is still much much work that needs to be done to make Dionysia a viable alternative to software LP solvers.

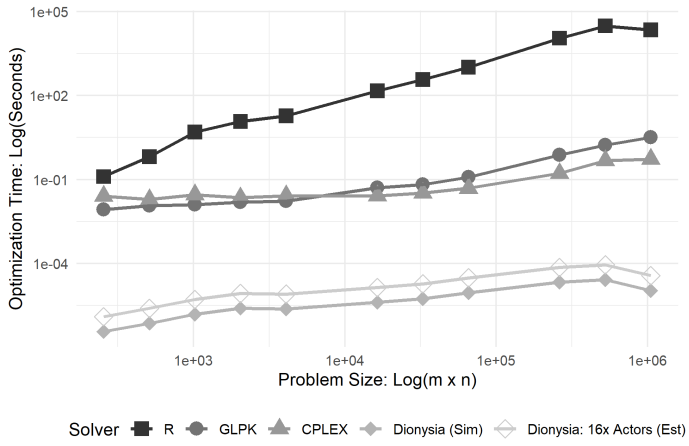

Fig. 2. Linear Programming Benchmarks

Most of this work will involve exploring the various options that exist for handling larger LP problems, however we will also need to conduct extensive benchmarking and area analysis to understand where Dionysia can be most improved.

A. Benchmarking

Our existing benchmarks are based on cycle-accurate simulations. While we believe these are fairly accurate, we would still like to properly benchmark the largest (in terms of constraints and variables) instance of Dionysia (as it exists now) which can fit on the F1 platform. We also need to conduct software benchmarks using the Gurobi Optimizer, another widely used commercial software solution.

B. Working With Larger Linear Programs

While software solutions can be distributed across datacenters to solve LP problems with thousands of variables and thousands of constraints, Dionysia is limited extremely limited (though we have yet to estimate the max problem size, we believe it to be on the order of tens of variables and tens of constraints). We have several ideas that could be combined to adress this problem.

The most immediate solution is to make the actors more efficient. Each actor currently implements the same set of arithmetic units. While these units are relatively small, their duplication uses up a large amount of FPGA area (e.g. our smallest viable example uses 39.5% of the FPGA). To use FPGA space more efficiently (and reduce redunancy) we could have each actor represent a set of tableau rows rather than a single row. This will require an increase in the number of cycles needed to carry out the Simplex iteration, but the scaling is favorable. We estimate that increasing rows held by each actor by a factor of two requires an increase in four cycles per Simplex iteration (e.g. actors holding two rows will require 29 cycles per iteration).

In addition to large actors we can also make use of tiling to split the tabluea into chunks stored in DRAM. These chunks can then be loaded as-needed by the actors. This approach can be combined with pipelining and pre-fetching to mitigate the extra cycles required by reliance on DRAM. This will also allow us to make use of the FPGAs DRAM and support problems that larger than can be stored in registers. Pipelining could also allow the accelerator to operate either on multiple problems at once or different stage of the same problem at once thereby improving performance. Additionally, the simplicity of the director would allow for the addition of this improvement without significant difficulty or changes.

Techniques which could significantly mitigate the impact of intermediate memory loads are FPGA-chaining and pre-fetching. FPGA-chaining would involve connecting multiple FPGAs together to solve LP problems using the Simplex accelerator. Due to the distinct separation between actors, there would be little communication between FPGAs thereby allowing each FPGA to hold fewer actors each of larger size so that row-sizes could be significantly larger before they need to be operated on in pieces. These larger actors would reduce the total number of intermediate memory accesses required and improve the performance of the accelerator since only occasional communication between FPGAs would be necessary and all communication would come from a single Director thereby eliminating any communication traffic.

Pre-fetching presents a potentially even larger mitigation of intermediate memory accesses than FPGA-chaining. The reason for this is that the memory access pattern for rows is always the exact same: load the first chunk, then the second chunk, and so on. Therefore, memory could be pre-fetched to limit or eliminate waiting for row values once they need to be operated on thereby masking the intermediate memory loads. A downside of pre-fetching is that it would take up some of the memory which could be used for values which are actively being operated on. However, our implementation of Simplex was significantly more logic intensive than memory intensive so it would only be a minor issue if it is one at all because many row-chunks could be pre-fetched before the computation size cannot keep up with the chunk size. Additionally, combining prefetching with FPGA-chaining would limit the number of chunks which would limit the amount of memory that need to be stored on chip at any given time.

We are also interested in tailoring Dionysia for real-time applications that could benefit from fast LP optimization on limited problem sizes. This is an area severely underserved by software solutions which largely rely on costly hardware to scale.

VI. Conclusion

We consider Dionysia to be mostly successful. In terms of our objectives: we have provided a simple interface that uses the standard simplex tableau, an accelerator that can be configured to target square (equal number of constraints and variables), tall (more constraints than variables), and wide (more variables than constraints) LP problems. The parallel-memory operations enabled by both designs allows for substantial performance increases compared to state-of-the-art software. While Dionysia has several limiting factors, we believe that the core architecture offers a solid foundation on which to explore further hardware designs. Ideally, we

would like to design a parametrizable accelerator that can take advantage of all of these approaches to allow designers to further optimize design towards specific kinds of LP problems.

REFERENCES

[1] Dickson, D. Bruce, "Ancient Agriculture and Population at Tikal, Guatemala: An Application of Linear Programming to the Simulation of an Archaeological Problem," American Antiquity, Volume 45, 1980
[2] Nikolaos Ploskas, Nikolaos Samaras, "Efficient GPU-based implementations of simplex type algorithms," Applied Mathematics and Computation, Volume 250, 2015
[3] Daniel A. Spielman and Shang-Hua Teng. "Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time." J. ACM 51, 3 (May 2004)
[4] M. N. Bojnordi and E. Ipek, "Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona. 20
[5] Bayliss, Bouganis, Constantinides, and Luk, "An FPGA Implementation of the Simplex Algorithm," 2007 IEEE Explore
[6] Coutinhoa, Lins e Silvab, Aloisec, and Xavier-de-Souza, "A Scalable Shared-Memory Parallel Simplex for Large-Scale Linear Programming," 2019 arXiv
[7] Clack, Christopher & Xie, Y. & Macdonald, A.. (2014). Linear programming techniques for developing an optimal electrical system including high-voltage direct-current transmission and storage. International Journal of Power and Energy Systems. 68. 103-114.
[8] "ILOG CPLEX Optimization Studio - Pricing." IBM, https://www.ibm.com/products/ilog-cplex-optimization-studio/pricing.
[9] Dantzig, George B., 1982, "Reminiscences about the origins of linear programming," 43 - 48, Operations Research Letters