Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2012

Project Title:	**A Novel Technique for Parallelizing Sparse Matrix Factorization using Unrolling and Re-Association**
Student:	**Siddhartha**
CID:	**00507051**
Course:	**EE3**
Project Supervisor:	**Professor Kapre Nachiket**
Second Marker:	**Dr Christos Bouganis**

Abstract

Sparse matrix factorization is a very common numerical tool that is utilized for solving large sets of linear equations. This is useful in many different engineering fields, such as circuit simulation for large networks. However, it is also often a critical computational bottleneck that undesirably increases the runtime of large problem sets. The project explores a novel parallelization technique that unrolls inner loops in a matrix-solve computation and re-associates factorization terms for faster results. This novel technique is theoretically expected to reduce the latency operation from an order of complexity O(N) to O($log_2(N)$). We develop a preliminary sparse matrix preprocessor software model in Java to generate compute graphs of matrix solve operations using the novel technique, which are then used to benchmark the improvements in performance over the existing methods. We measure performance in context of the SPICE circuit solver and see reductions in SPICE runtimes as project goals.

ACKNOWLEDGEMENTS

First and foremost, I'll like to thank my supervisor, Dr. Kapre Nachiket for his guidance, encouragement and patience with me. I have had a wonderful time working with him, and I look forward to working together further in the summer. I'll also like to thank Dr. Bouganis for his invaluable advise during the meet-the-2nd-marker meeting.

Aside from that, I'll like to give a hearty thanks to my friends in London, who've all chipped in with a word of encouragement or two.

Thank you everyone, it would not have been the same without each and every one of you.

Contents

1 INTRODUCTION

Sparse matrix factorization is a very common numerical technique utilized in many engineering fields. It is, however, a compute heavy operation that has many data dependencies, which result in speed bottlenecks for large matrix sized problems. There are several methods that can be utilized to improve the compute speed, and with the advent of custom hardware technology (e.g. FPGAs), it has become feasible to parallelize the computations in a sparse matrix solve problem.

The LU Decomposition is a useful method that is often employed to solve a large set of linear equations expressed in the matrix form. Most matrices can be decomposed into two matrices; L: lower triangular matrix and U: the upper triangular matrix, such that a matrix A can be expressed as a product A = LU. This new form can then be used to solve the matrix equation in a 2-step method. We will look in detail into the matrix solve method using LU decomposition in section 2 of this report.

SPICE (Simulation Program with Integrated Circuit Emphasis) circuit solver utilizes this LU decomposition matrix solve method to arrive at its solution. It forms a very critical portion of its runtime and hence, speeding up this matrix solve process can significantly improve the SPICE runtimes. We have chosen to focus our project scope in context of the SPICE circuit solver and set our goals towards achieving a speedup in SPICE compute times. We will explain the SPICE operation in greater detail in section 2 of this report, and share some of the performance results in section 4.

It must, however, be emphasized that this novel technique is not limited to improving SPICE runtimes. Many engineering fields require solving a large set of linear equations, and this novel technique can be applied with little or no tweaking to achieve similar improvements in performance.

2 BACKGROUND INFORMATION

2.1 Solving Linear Equations

Solving sets of simultaneous linear equations is often an important step in many real world problems that come from a range from fields like engineering, sciences and economics. Depending on the nature of the problem, there could be a number of approaches used to find a solution. For a large set of simulateneous linear equations, it is often advisable to use matrix notation to represent the equation. The following example illustrates how two simultaneous equations can be represented in matrix form below:

$$\begin{aligned} 4x_1 + 3x_2 = 5 \\ 6x_1 + 3x_2 = 9 \end{aligned}$$

can be expressed as:

$$\begin{bmatrix} 4 & 3 & 1 & -1 \\ 6 & 3 & 0 & 0 \\ 3 & -3 & -5 & -1 \\ 1 & 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 5 \\ 9 \\ -2 \\ -7 \end{bmatrix}$$

which is a linear matrix equation of the form $\mathbf{LU}\vec{x} = \vec{b}$ where

$$\mathbf{A} = \begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix}$$

Such a matrix equation can be solved with several methods such as finding the inverse matrix $\mathbf{A^{-1}}$ or performing Gaussian Elimination. The technique that we would be looking into is the decomposition of matrix **A** into its lower and upper triangular factors, **L** and **U**.

An LU decomposition can be carried out by doing Gaussian Elimination. We will not look in detail into this process for the scope of our project. However, it is important to note the properties of the **L** and **U** factor matrices. The matrix **A** from the example above can be factored into the following **LU** factors:

$$\begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1.5 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 0 & -1.5 \end{bmatrix}$$

Note that the **L** and **U** factors are supposed to have the following structure (for an n x n matrix):

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ L_{2,1} & 1 & 0 & \cdots & 0 \\ L_{3,1} & L_{3,2} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n,1} & L_{n,2} & L_{n,3} & \cdots & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} & \cdots & U_{1,n} \\ 0 & U_{2,2} & U_{2,3} & \cdots & U_{2,n} \\ 0 & 0 & U_{3,3} & \cdots & U_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & Un,n \end{bmatrix}$$

This structure is crucial in the next step for solving the matrix equation.

2.2 Matrix-Solve using LU Decomposition

The example from above, after **A** is decomposed into its **LU** factors, can be solved in two steps: front-solve and back-solve.

2.2.1 Front-Solve

We start with the equation $\mathbf{A}\vec{x} = \vec{b}$, which can be written as $\mathbf{LU}\vec{x} = \vec{b}$ after matrix **A** is decomposed into its **LU** factors. We observe that $\mathbf{U}\vec{x}$ gives us another column vector, which we can denote as $\vec{y}$. Hence, the equation can be re-written as $\mathbf{A}\vec{y} = \vec{b}$, where we now solve for $\vec{y}$, given we know the values for **L** and $\vec{b}$. This is known as a front-solve, and can be sequentially calculated with ease, due to the diaganol structure of a lower-triangular matrix. The solution of an n x n front-solve can be written as follows:

$y_1 = b_1$
$y_2 = b_2 - L_{2,1} * y_1$
$y_3 = b_3 - (L_{3,1} * y_1 + L_{3,2} * y_2)$
$y_4 = b_4 - (L_{4,1} * y_1 + L_{4,2} * y_2 + L_{4,3} * y_3)$
$\vdots$
$y_n = b_{n-1} - (L_{n,1} * y_1 + L_{n,2} * y_2 + L_{n,3} * y_3 + \cdots + L_{n,n-1} * y_{n-1})$

Note the data dependencies present in this sequential front-solve, i.e. we need to calculate y_1 before we can calculate y_2, y_1 **and** y_2 before we can calculate y_3, and so on.

2.2.2 Back-Solve

Similarly, we can now solve for the column vector $\vec{x}$ by solving the linear equation $\mathbf{U}\vec{x} = \vec{y}$, where $\vec{y}$ is the solution column vector from the previous front-solve step. The solution, solved sequentially, would look as follows:

$$x_n = \frac{y_n}{U_{n,n}}$$
$$x_{n-1} = \frac{y_{n-1}-U_{n-1,n}*x_n}{U_{n-1,n-1}}$$
$$x_{n-2} = \frac{y_{n-2}-(U_{n-2,n}*x_n+U_{n-2,n-1}*x_{n-1}}{U_{n-2,n-2}})$$
$$x_{n-3} = \frac{y_{n-3}-(U_{n-3,n}*x_n+U_{n-3,n-1}*x_{n-1}+U_{n-3,n-2}*x_{n-2}}{U_{n-3,n-3}})$$
$$\vdots$$
$$x_1 = \frac{y_1-(U_{1,n}*x_n+U_{1,n-1}*x_{n-1}+U_{1,n-2}*x_{n-2}+\cdots+U_{1,2}*x_2}{U_{1,1}})$$

Again, note the data dependencies when doing the back-solve sequentially. Also, the order of computation is reversed, i.e. we compute x_n first and x_1 last.

All singular matrices can be decomposed into their L and U factors, and hence, this then provides us with a robust sequential method to solve a matrix equation of the form $\mathbf{A}\vec{x} = \vec{b}$. Algorithms to carry out the decomposition, front-solve and back-solve can be easily written in software, and many applications today utilize this technique to solve such linear systems. One such example is of the SPICE circuit solver, which models circuit components as linear devices and solves the corresponding linear equations using the LU decomposition method.

2.3 Introduction To SPICE

SPICE (Simulation Program with Integrated Circuit Emphasis) is an electronic circuit simulator. It was first developed in the University of California, Berkeley, and released in 1975 as an open-source project. Since then, it has undergone many revisions and has become the current industry standard for testing analog electronic circuits, before they are sent for integrated circuit fabrication. While both hardware capacity and processing speed have been scaling upwards in accordance to the Moores Law, SPICE simulation of future hardware has been slowing down. Several methods have been employed to reduce the SPICE runtime, such as porting the compute-heavy operations of SPICE to customized FPGA hardware.

Many of these approaches have helped to reduce SPICE compute time and further efforts are in progress to bring about more improvements.

Our approach in this project is targeted at the SPICE Matrix-Solve Phase. This is one of the three main operation phases of the SPICE circuit solver. The first phase, Model Evaluation Phase, models the circuit components that are specified in a circuit design file. The aim of this phase is to model all the circuit components (both linear and non-linear) as linear variables to output a set of linear equations expressed in the matrix form explained above. Linear components, such as resistors, can be modelled as linear components easily, whereas for non-linear components, such as transistors, the circuit solver does Newton-Raphson iterations to converge to an operating-point solution [1]. This process takes up a very significant portion of the total SPICE runtime, and is largely dependent on the number of non-linear components that need to be modelled, since they require the most number of operations to process. The results from this phase are passed on to the Sparse Matrix-Solve Phase, which essentially solves the matrix equation using the LU Decomposition method. The last major portion of SPICE runtime is consumed by the Iteration Control, which is responsible for the Newton-Raphson and adaptive time-stepping iterations that are used in the Model Evaluation phase.

For the purposes of our project, we would only look into detail at the SPICE Sparse Matrix-Solve Phase.

2.3.1 SPICE Sparse Matrix-Solve Phase

The Sparse Matrix-Solve Phase, as stated earlier, is the portion of SPICE runtime during which the linear matrix equation is solved using the LU decomposition method. This phase approximately contributes to about 38% of total SPICE runtime[1]. As circuit sizes continue to get smaller, more parasitic capacitances and inductances have to be modelled in, which can result in an increase in the percentage runtime spent in the Sparse Matrix-Solve phase. Hence, there is significant motivation to improve the Sparse Matrix-Solve phase as much as possible.

The Model Evaluation phase outputs a matrix $\mathbf{A}$ and column vector $\vec{b}$, which we use to solve for $\vec{x}$ in the equation $\mathbf{A}\vec{x} = \vec{b}$. Since the circuit components in a real-world circuit tend to be connected to only a few other circuit components, most of the matrix components of matrix $\mathbf{A}$ are zeroes, with an order of approximately O(1) entries in each row of matrix $\mathbf{A}$. Such a matrix is known as a sparse matrix, and hence, it would be in our interest to maintain the sparsity as much as possible when decomposing this matrix into its L and U factors. This is achieved by the SPICE KLU Solver.

2.3.2 SPICE KLU Solver

The SPICE KLU Solver uses algorithms that allows the matrix **A** to be decomposed into its L and U factors, which are static and do not change from iteration to iteration[2]. This had not always been the case, as matrix solve compute graphs changed with every iteration. This allows us to parallelize the computations when doing the back and front solve, as the L and U factor matrices are static, and can be pre-processed for generating a fixed compute graph.

This parallelism potential offered through the SPICE KLU Solver allows us to generate the compute graphs before the start of the solve iterations. Hence, we can in fact remove many data dependencies inherent in a conventional front or back solve and have more operations being done in parallel. Further discussion into how this could be achieved is discussed in section 3 of this report.

2.4 FPGAs

FPGAs (Field Programmable Gate Arrays) are a custom re-programmable hardware technology that has enabled and eased the process of parallelizing processes in order to rid of speed bottlenecks. FPGAs offer a plethora of options to a hardware designer to solve a compute problem in the most efficient way possible, and as demonstrated by Dr. Nachiket's research [1], porting SPICE operations to customized solutions on FPGAs has resulted in significant speedups in SPICE runtimes. In this project, we assume that the final implementation of our technique would be done through FPGAs.

3 UNROLL & RE-ASSOCIATIVITY

3.1 Matrix Unroll Technique

With the matrix unroll technique, our aim is to remove the data dependencies between the calculation of each unknown variable. Once these data dependencies are removed, we can re-arrange our compute graph such that more calculations are done in parallel, and hence, improving the speed performance. This is possible by 'unrolling' the entire matrix solve computation for both the front and back solve in an LU-matrix-solve equation. We would demonstrate how this can be achieved by using an example of a 4x4 lower-triangular matrix equation below (i.e. front solve):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ L_{2,1} & 1 & 0 & 0 \\ L_{3,1} & L_{3,2} & 1 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

By subsituting all the y_i terms in the right hand side of the equations for a front solve, we obtain the following:

$y_1 = b_1$
$y_2 = b_2 - L_{2,1} * b_1$
$y_3 = b_3 - L_{3,1} * b_1 - L_{3,2} * b_2 + L_{3,2} * L_{2,1} * b_1$
$y_4 = b_4 - L_{4,1} * b_1 - L_{4,2} * b_2 + L_{4,2} * L_{2,1} * b_1 - L_{4,3} * b_3 + L_{4,3} * L_{3,1} * b_1 + L_{4,3} * L_{3,2} * b_2 - L_{4,3} * L_{3,2} * L_{2,1} * b_1$

Note that the data dependencies have been removed, and each y_i can be computed directly using the matrix and column vector values, which are all known values from the beginning.

Similarly, the back-solve looks as follows:

$$\begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} & U_{1,4} \\ 0 & U_{2,2} & U_{2,3} & U_{2,4} \\ 0 & 0 & U_{3,3} & U_{3,4} \\ 0 & 0 & 0 & U_{4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

where

$x_4 = \frac{y_4}{U_{4,4}}$
$x_3 = \frac{y_3}{U_{3,3}} - \frac{U_{3,4}*y_4}{U_{4,4}*U_{3,3}}$
$x_2 = \frac{y_2}{U_{2,2}} - \frac{U_{2,4}*y_4}{U_{4,4}*U_{2,2}} - \frac{U_{2,3}*y_3}{U_{3,3}*U_{2,2}} + \frac{U_{2,3}*U_{3,4}*y_4}{U_{2,2}*U_{3,3}*U_{4,4}}$
$x_1 = \frac{y_1}{U_{1,1}} - \frac{U_{1,4}*y_4}{U_{1,1}*U_{4,4}} - \frac{U_{1,3}*y_3}{U_{3,3}*U_{1,1}} + \frac{U_{1,3}*U_{3,4}*y_4}{U_{4,4}*U_{3,3}*U_{1,1}} - \frac{U_{1,2}*y_2}{U_{2,2}*U_{1,1}} + \frac{U_{1,2}*U_{2,4}*y_4}{U_{4,4}*U_{2,2}*U_{1,1}} + \frac{U_{1,2}*U_{2,3}*y_3}{U_{3,3}*U_{2,2}*U1,1} -$
$\frac{U_{1,2}*U_{2,3}*U_{3,4}*y_4}{U_{1,1}*U_{2,2}*U_{3,3}*U_{4,4}}$

Note the number of multiply terms which double as we calculate every next x_i term. We can expect to have a significant increase in resource usage if we were to compute all these multiply terms in parallel for very large matrices. This is an engineering tradeoff that we are willing to make for the improved performance in compute speed.

Unrolling the computations as follows allows us to compute all the x_i terms in parallel. It should be noted that with unroll, the slowest computation would be that of the y_4 term (for front-solve) or x_1 term (for the back-solve). Hence, assuming all computations are done in parallel, the compute time for the entire unroll case can be seen as the compute time for the longest compute graph. The figures 1.1 and 1.2 on the next pages show a comparison between the without unroll and with unroll compute graphs for the front-solve example of a 5x5 matrix equation. Note that for unroll case (figure 1.2), we show the compute time for y_5.

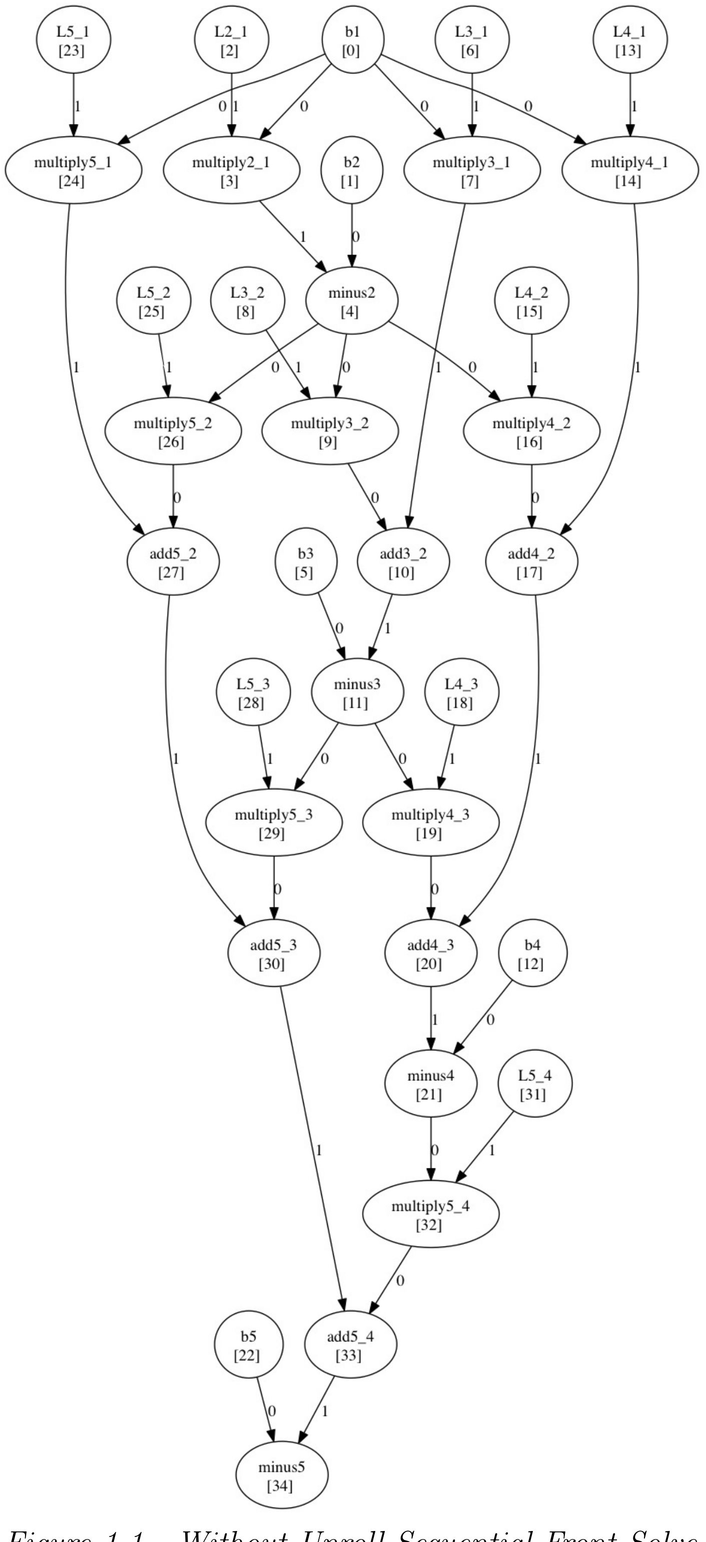


Figure 1.1 - Without Unroll Sequential Front-Solve

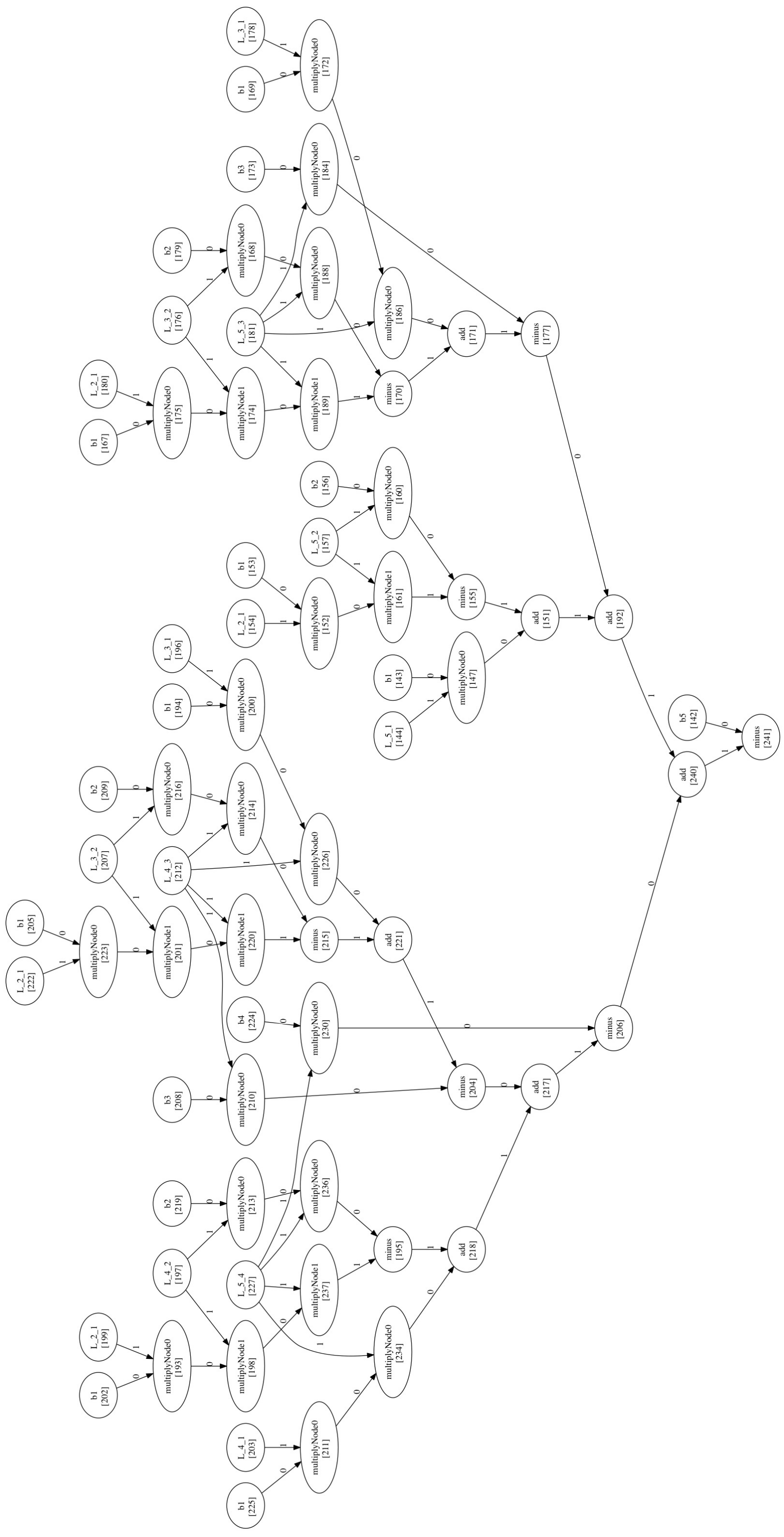


Figure 1.2 - With Unroll Parallel Front-Solve

The most important difference we observe from the two graphs is the depth and width of the compute tree. As expected, the compute tree for an unroll case is wider but shallower. We are essentially doing more computations at one time to reach the goal in less time. Note in Figure 1.1 for the without unroll case, the values for each y_i can be extracted from different nodes of the graph (not shown) and the final node outputs the value of y_5.

With reference to Figure 1.2, we also ensure that our multiply nodes are arranged in sequence to give multiply chains. For example, the term $L_{4,3} * L_{3,2} * L_{2,1} * b_1$ is arranged as a chain of multiplies before being sent to an add or subtract node, as can be seen from Figure 1.2. Arranging the multiply nodes in such chains is crucial in our next step, which makes the computation more efficient through parallellization by re-associating the inputs and operations in the compute graph.

3.2 Re-Associativity

Re-associativity is the process through which we re-arrange the inputs and nodes in a compute graph to further exploit potential parallelism.

3.2.1 Multiply Chains to Multiply Trees

We first look at how we can transform multiply chains into multiply trees. Figures 2.1 and 2.2 on the following pages illustrate a simple example, which showcases how re-association of terms can lead to an improvement in the level of parallelism in the compute order.

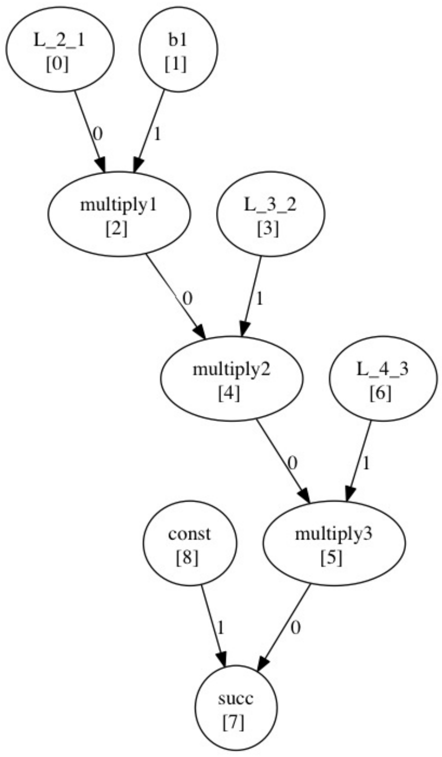


Figure 2.1 - Multiply Chains without Re-Association

In Figure 2.1, we have the trivial case where a multiply chain representing the multiply term $L_{4,3} * L_{3,2} * L_{2,1} * b_1$ is shown. The output of this multiply chain goes to a successor node, which is not a multiply operation, and *const* is a constant. Such a multiply chain can be re-arranged and re-associated in a different way to give a balanced multiply tree structure, as shown in Figure 2.2 below.

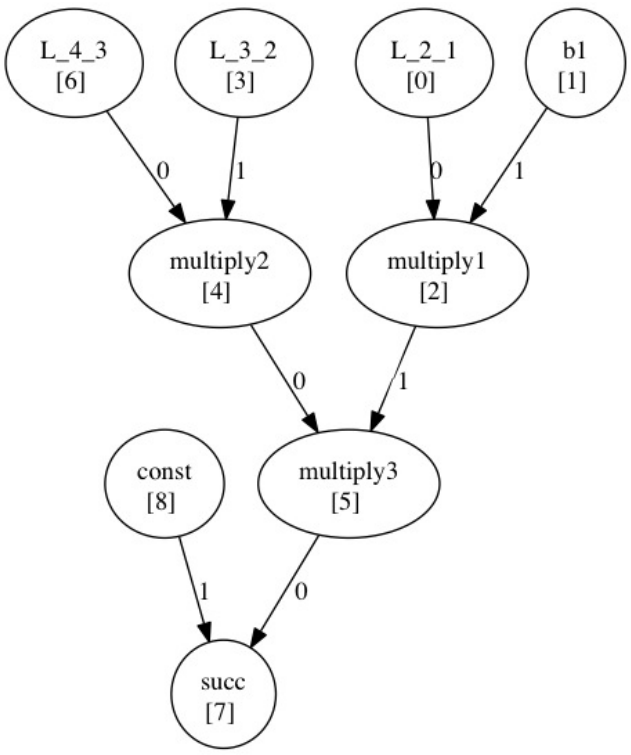


Figure 2.2 - Multiply Trees after Re-Association

The multiplies $(L_{2,1} * b_1)$ and $(L_{4,3} * L_{3,2})$ are now done in parallel and then sent to the next multiply node, giving the same desired result. Such a balanced tree structure, for a small example as follows, has saved us one multiply instruction cycle, thus exploiting the parallellism even further. For larger multiply chains, which we will observe in bigger examples, even more clock cycles can be saved. Hence, there is significant motivation to achieve a multiply tree structure as far as possible in every compute graph for each unknown x_i.

3.2.2 Add/Subtract Trees

Add trees can be constructed in the same fashion as the multiply trees from add chains. This is due to the associative law, where it does not matter how the inputs are ordered. Dealing with subtract chains is slightly more tricky, as we now have to take note of the polarity of the term being fed into the subtract node. A subtract chain can be converted into a balanced tree structure by re-associaiting inputs into add nodes, which replace a subtract node. In this project so far, however, a subtract chain is not formed due to the method of generating the compute graphs for the unrolled and re-associated cases. Hence, it is unlikely that we would need to design a solution to re-associate subtract chains into subtract/add trees.

3.3 Complexities

In this section, we look into the computational and space complexities of the unroll (with re-associativity) and without unroll scenarios. We compare the complexities in the two scenarios and develop a theoretical understanding which we can use to predict the potential speedup possible with this novel technique.

3.3.1 Comparison of Compute Complexity

In the case of this project, it is meaningful to talk about the order of the latency that we will theoretically observe for different approaches, since we are interested in reducing the compute time, i.e. latency, as much as possible.

Assuming a n x n matrix, we are interested in the latency before the solution for the x_{nth} unknown variable is computed. In a completely sequential without unroll scenario, the final unknown variable cannot be solved for until all the previous unknown variables have converged to a solution. Consider a lower-triangular matrix solve scenario. Starting from the first unknown variable, x_1, the number of multiply operations increases by 1 every iteration. This is also the same for the number of adds, and each iteration requires 1 subtract. Hence, if each compute for an unknown variable is done sequentially, we can expect the number of operations required, i.e. the latency, to be the sum of the number of adds, multiplies and subtracts. The table below summarizes this info.

	Total Number of Operations	Large n (approx.)
Multiplies	0+1+2+3+4+....+n-1	$\frac{n(n+1)}{2}$ - n
Adds	0+1+2+3+4+....+n-1	$\frac{n(n+1)}{2}$ - n
Subtracts	n	n

Therefore, adding the number of operations for large n, we get an order of complexy of O(n^2). This is an undesirable feature, since the latency increases quadratically as our matrix sizes increase.

This latency can be reduced firstly by parallelising some of the operations in the sequential compute itself. In the above case, we assume that the computation of, for example, x_5 does not *begin* until x_4 has been computed. This is, however, unnecessarily inefficient, since part of the computation for x_5 can be done as soon as x_1, x_2, x_3 are computed. By using flags for each unknown variable that is set when its value is computed, we can schedule the compute process better such that the overall latency is reduced. We can determine the complexity of this method by analyzing it intuitively. Each computation of the unknown variable x_i requires extra 2 operations (a mutliply and an add) than the previous one, x_{i-1}. The extra multiply and add involve the directly previous unknown variable, x_{i-1} and hence, for large n, the computations for the terms involving the earlier unknown variables tend to 'catch up' to the current computation of the directly adjacent unknown variable. Hence, as n tends to a large value, the complexity of the latency begins to reduce to an order of n, i.e. O(n), when scheduled in this fashion.

For an unroll case where the entire computation is unrolled and re-associated for maximum parallelism, the latency can be reduced even further. By re-associating multiply chains in balanced binary multiply trees, the latency is reduced even further by a logarithmic operator. In the best case scenario, if there is a multiply chain with n multiply nodes, where n is a power of 2, the multiply chain can be re-arranged into a balanced binary tree that has an operation depth of $log_2(n)$. Hence, the compute complexity can be as low as O($log_2(n)$) for cases with these multiply chains of length of power 2. For other cases, the complexity is harder to analyze and determine, as it varies for different cases. However, the worst-case scenario would always be a latency of O(n), as in the previous case.

All in all, by scheduling, or parallelizing our computations, we ensure that the latency is reduced significantly as the number of operations increase. This is important for engineering applications such as SPICE, which deal with millions of operations to arrive at the solution. Unroll and re-associaiting the multiply chains in trees can further reduce the complexity and give us a best possible scenario of O($log_2(n)$) latency.

3.3.2 Space Complexity

The space complexity is the amount of resources that we would need in order to implement a method.

For the sequential, without unroll case, at any one instance, the maximum number of values that we need to hold in registers is n - 1 for an n x n matrix. Hence, the space complexity for this method is O(n).

As we begin to parallelize computations, the space complexity would undoubtedly increase, as we have to hold many values in registers for the compute tree. As we unroll computations of the larger unknown variables, we find that the number of multiply chains double at every iteration. This appears to give a complexity of

O(2^n), but it can be reduced with proper scheduling and management of resources. This is an aspect of the project that needs to be further analyzed in detail and possibly work towards tackling the high demand on resources as we push to make our compute graph more parallel.

3.4 UPENN Sparse Matrix Pre-Processor

We have thus far defined a new technique which we would like to develop in order to parallelize the SPICE matrix solve computations. We have established that in theory, we should observe a significant speedup, due to the compute complexity of the unroll, with re-associativity, case being of significantly lower order. Hence, we design and develop a software model that could test and debug this new technique by virtually simulating its performance. This would enable us to test and determine performance limits before implementing the ideal design on actual hardware.

The UPENN Sparse Matrix Pre-Processor has been in development for this purpose and is a critical component of this project. The following section details on its features and developments relevant to this project.

4 UPENN SPARSE MATRIX PRE-PROCESSOR

The UPENN Sparse Matrix Pre-Processor is a software model written in Java. It has been in development for several years, and was intended to serve university research into the acceleration of SPICE runtimes. It has many features and functionalities for a variety of purposes, and in this report, we would focus on ones relevant and used in the project.

4.1 Features & Functionalities

We used two key functionalities of the sparse matrix pre-processor to work towards our goals:

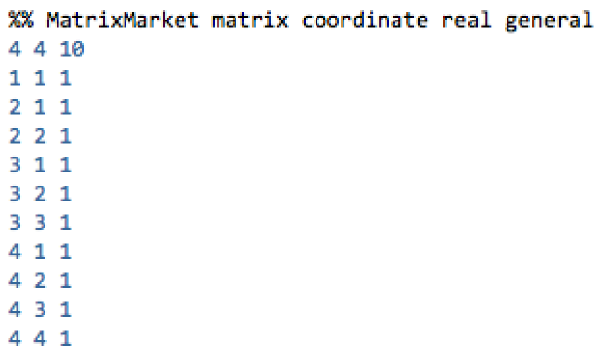
1) Generation of compute graphs using software functions. We use this functionality to write algorithms that are able to generate unroll and without unroll graphs for dense and sparse matrices. Algorithms to re-associate nodes and inputs to form multiply chains are also written to parse through compute graphs and re-associate for maximum parallelism.

2) Graph scheduling for performance analysis on virtually simulated hardware. We use this feature to compare the performance of unroll (with/without re-associativity) and without unroll cases. For the scope of this project, the graph scheduler is treated as a black-box, as it has a stable development release and understanding its operating methods is beyond the scope of this project.

4.2 A Simple Example

To demonstrate how the unroll and without unroll graphs generating algorithms can be written, we will work with a simple example in this section to demonstrate the outcomes. We work with a 4 x 4 lower triangular dense matrix solve, where all the non-zero values in matrix **A** are 1. A matrix file for such a matrix is written as follows:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$



```
%% MatrixMarket matrix coordinate real general
4 4 10
1 1 1
2 1 1
2 2 1
3 1 1
3 2 1
3 3 1
4 1 1
4 2 1
4 3 1
4 4 1
```

Figure 3 - Declaring a ".mtx" Matrix file of matrix L

and saved as *"ltran_test.mtx"*. The first line of this .mtx file is a comment, which is ignored by the parser. The second line defines the dimensions of the matrix to be constructed; first number represents number of rows - 4, second represents the number of columns - 4, and the final number represents the number of non-zero entries - 10. The following lines give the index (row number, followed by column number) of each non-zero value that has to be filled out.

For this example, we assume that the equation we are looking to solve is of the following form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

where $\vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$ is a column vector of known values, and we are solving for all the unknown y_i.

We would now use this simple example to analyze and write our algorithms that generate the unroll and without unroll compute graphs.

4.2.1 Non-Unroll Compute Graph

The solutions for the above graph, computed sequentially, would look as follows:

$y_1 = b_1$
$y_2 = b_2 - L_{2,1} * y_1$
$y_3 = b_3 - L_{3,1} * y_1 - L_{3,2} * y_2$
$y_4 = b_4 - L_{4,1} * y_1 - L_{4,2} * y_2 - L_{4,3} * y_3$

Since we are interested in the total compute time, we want our without unroll function to output the complete sequential compute tree for all unknown variables on one graph. To achieve this, we have to keep track of the last node of the graph of each unknown variable y_i, as we will need to use it (and ones preceding it) to generate the compute graph for the next unknown variable y_{i+1}. The only exception is y_1, which is the trivial case where $y_1 = b_1$ due to the properties of the lower-triangular matrix L. Below is the pseudocode that would generate a singular graph that shows the complete without unroll case for the example above.

```
y[ ] : array of graphs to store compute graphs of unknown variables $y_i$
$graph = createNewGraph();$
begin for $i := 1$ to $numberOfRows$ step $1$ do
            $addToGraph(b_i);$
            do if $i == 1$ then $y[i] = graph;$ fi od
            do else
                for $j := 1$ to $i$ step $1$ do
                    $addToGraph(L_{i,j});$
                    $lastNode = returnLastNodeOfGraph(y[j]);$
                    $multiply(L_{i,j}, lastNode);$ //Adds the two inputs to a multiply node
                od
                $addNodes($//Connect all multiply nodes from previous 'for' loop using add nodes);
            od
            $minus(b_i, addNode);$              //Connect final add node to minus node
            $y[i] = graph;$
  od
  return $graph;$
end
```

We wrote the above pseudocode in the Java language with functions provided in the UPENN Sparse Matrix Pre-Processor software framework. The full, working code, can be found in the appendix at the end of this report.

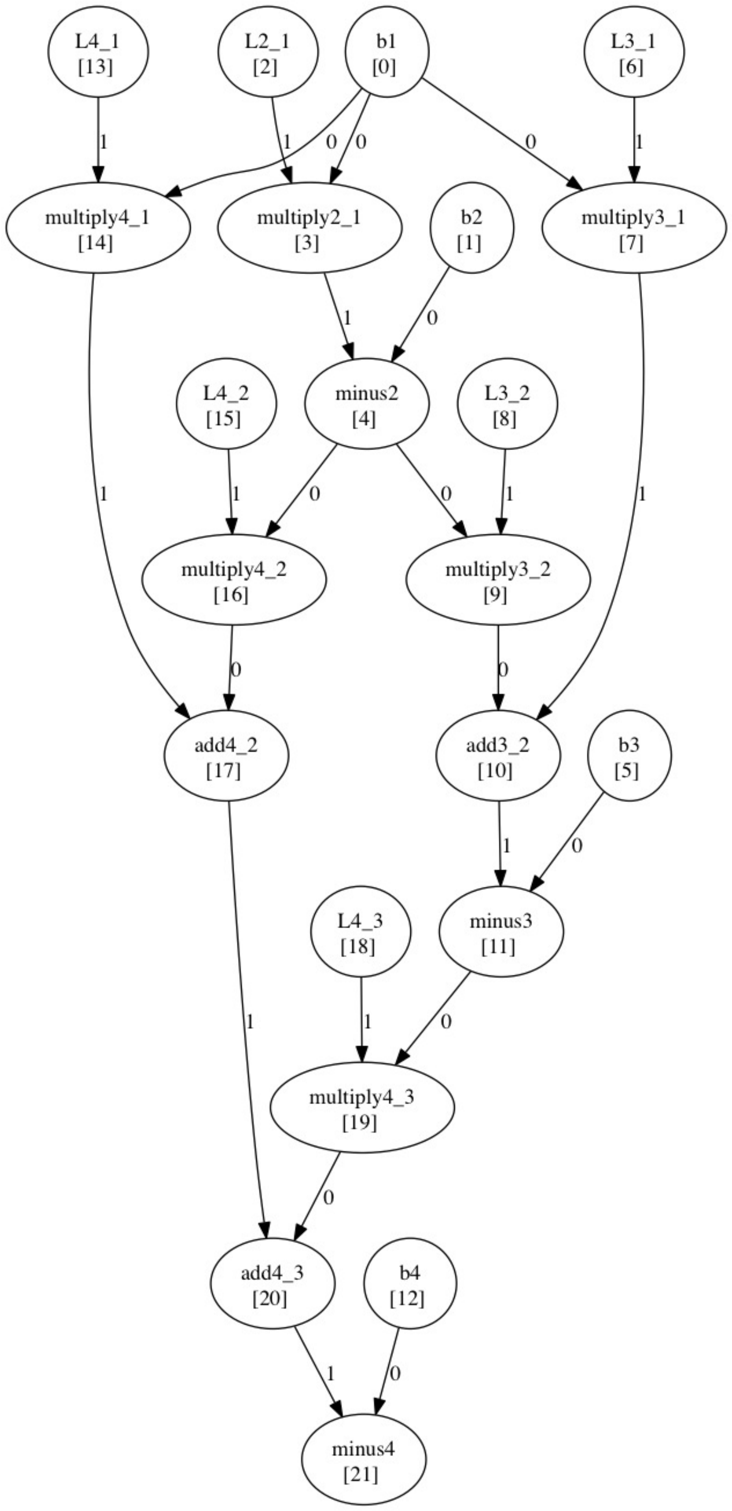
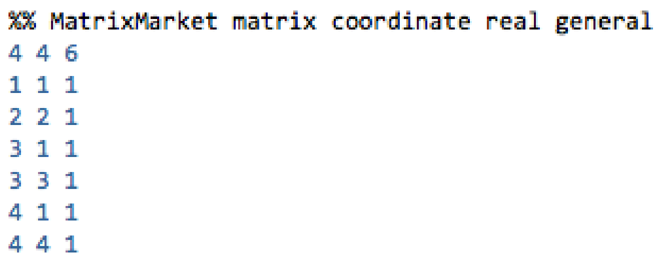


Figure 4: Without Unroll Graph generated from algorithm

Figure 4 on the previous page shows the without unroll graph that was generated using the algorithm that we wrote for the project. The outer *for* loop iterates through the rows of the matrix, while the inner *for* loops iterates through the columns in each row. The outputs, i.e. the values of the unknown variables y_i, can be extracted from each of the *minus* nodes, except for y_1, which is equal to b_1. Due to the way the software model is able to parse and store the matrix components, it is possible to run this algorithm on a sparse matrix, and the correct graph would be generated. As an exercise, the following matrix was input:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$



```
%% MatrixMarket matrix coordinate real general
4 4 6
1 1 1
2 2 1
3 1 1
3 3 1
4 1 1
4 4 1
```

Figure 5 - Declaring a Sparse Matrix

And the following graph was produced:

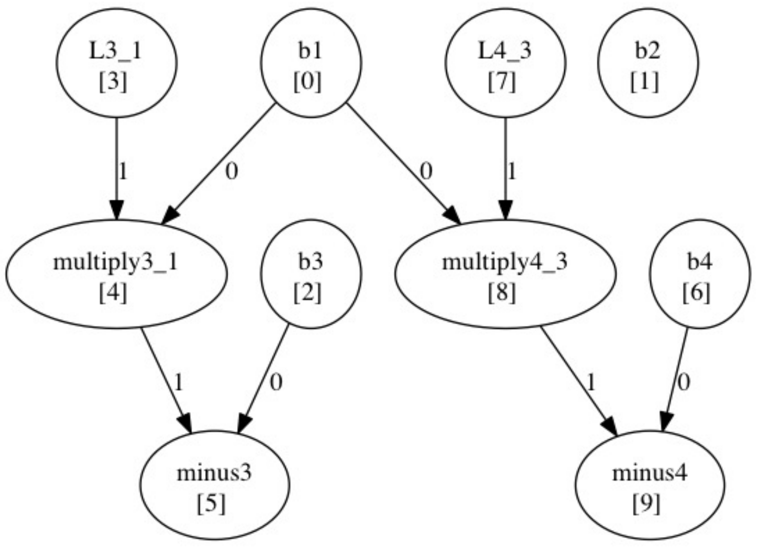


Figure 6 - Compute Graph of a Sparse Matrix (Without Unroll)

Note that the solution for y_1, y_2, y_3, y_4 can be read from the outputs of the nodes b_1, b_2, *minus3, minus4* respectively.

4.2.2 Unroll Graph

The solution for a parallel computation to solve the unknown variables would look as follows for the above example:

$y_1 = b_1$
$y_2 = b_2 - L_{2,1} * b_1$
$y_3 = b_3 - L_{3,1} * b_1 - L_{3,2} * b_2 + L_{3,2} * L_{2,1} * b_1$
$y_4 = b_4 - L_{4,1} * b_1 - L_{4,2} * b_2 + L_{4,2} * L_{2,1} * b_1 - L_{4,3} * b_3 + L_{4,3} * L_{3,1} * b_1 + L_{4,3} * L_{3,2} * b_2 - L_{4,3} * L_{3,2} * L_{2,1} * b_1$

For the unroll case, we would like to generate individual compute graphs to solve for each unknown variable y_i. Hence, for this example, 4 different graphs will be output. As explained above, the y_4 compute tree has the largest number of nodes to process and hence, if all the unknown variables are computed in parallel, the time taken to compute all variables can be equated to the time taken to compute y_4.

Methodology: To compute each unknown variable y_i, except y_1, we would require the graphs of the preceding $y_{i-1}, y_{i-2}, \cdots, y_1$. Furthermore, to compute y_i, we can view each solution to have the following general structure:

$$y_i = b_i - (L_{i,i-1} * y_{i-1} + L_{i,i-2} * y_{i-2} + \cdots + L_{i,1} * y_1)$$

where each $y_{i-1}, y_{i-2}, \cdots$ is a graph stored from previous solve iterations.

Hence, we can construct the compute graph for the current unknown y_i by multiplying the appropriate L matrix constant to each previously stored graph $(y_{i-1}, y_{i-2}, \cdots)$ and adding them together using add nodes. The final step would be to subtract the result after the add nodes from b_i.

The pseudocode for this algorithm would look as follows:

```
y[ ] : array of graphs to store compute graphs of unknown variables $y_i$
begin for $i := 1$ to $numberOfRows$ step $1$ do
        $graph = createNewGraph();$
```

```
            $addToGraph(b_i)$;
            do if $i == 1$ then $y[i] = graph$;  fi od
            do  else
                for $j := 1$ to $i$ step $1$ do
                    $addToGraph(L_{i,j})$;
                    $graph.addGraphToGraph(y[j])$; //Adds graph in parantheses to current graph
                    $multiply(L_{i,j}, y[j])$;                  //Multiplies graph with constant
                od
                $addNodes$(//Connect all multiply nodes from previous 'for' loop using add nodes);
            od
            $minus(b_i, addNode)$;              //Connect final add node to minus node
    od
    return $graph$;
end
```

A new graph is created for each iteration of i, which is running through the rows of the matrix, while the variable j runs through the columns in each vector. The multiply function in the inner loop is also very important, as it is responsible for propagating the multiply node up a compute tree and inserting it in the right parts of the graph. This allows us to create the multiply chains that we need in order to implement the re-associativity into multiply trees later in the project. The following figures below show how a multiply node is to be propagated up a compute tree. Here, we wish to propagate a multiply by "L_00" up a compute graph.

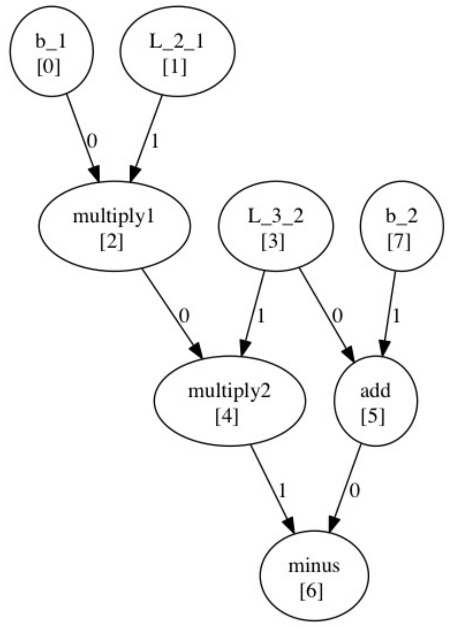


Figure 7 - Sample compute tree, before multiplication by "L_ 00"

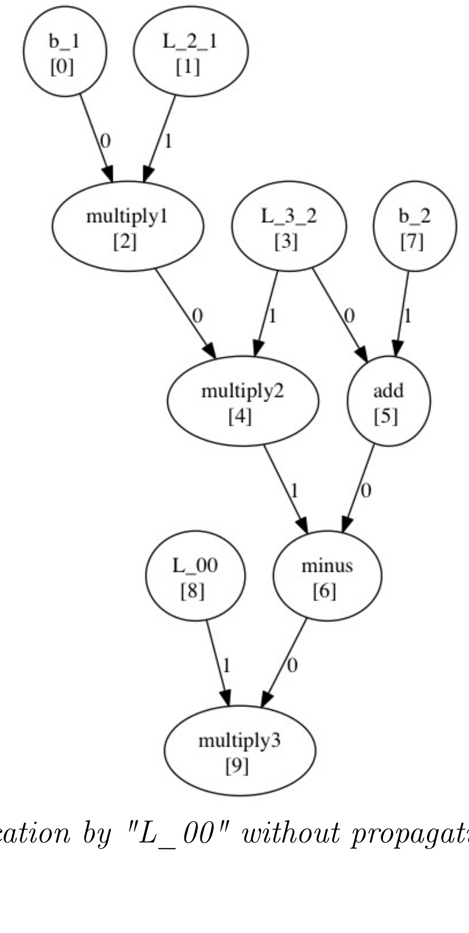


Figure 8 - Multiplication by "L_ 00" without propagating multiply node up

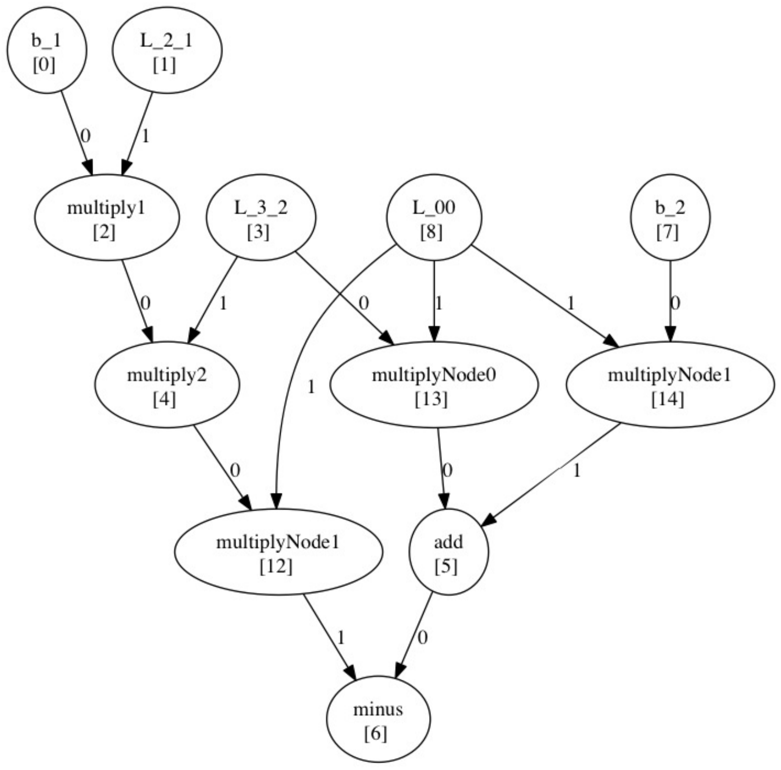


Figure 9 - Multiplication by "L_00" with propagation of multiply node up

In Figure 9, note that the multiply node is propagated up and duplicated appropriately up the graph as desired. This propagation up a tree is similar to opening up the brackets and multiplying a constant into the terms within the bracket. In this sample scenario, the original expression was $L_{2,1} * b_1 * L_{3,2} - (L_{3,2} + b_2)$, which when multiplied by "L_00", without first opening up the brackets would look like $L_{00} * (L_{2,1} * b_1 * L_{3,2} - (L_{3,2} + b_2))$. Opening up the outer bracket results in the following expression: $L_{00} * L_{2,1} * b_1 * L_{3,2} - (L_{00} * L_{3,2} + L_{00} * b_2)$. Note how duplicates of the L_{00} multiplies are created in each multiply term. While this appears to create additional computations through duplicate multiply nodes, the important outcome from this operation is the creation of multiply chains. From Figure 8, if the multiply node was not propagated, it would not be possible to make the computation any more efficient or parallel. However, in Figure 9, we have created a multiply chain consisting of the multiply nodes *multiply1, multiply2* and *multiplyNode1*. This particular multiply chain can be re-associated to give a balanced multiply tree of 4-input-multiply. We will look into how this can be created in the next section.

The algorithm for propagating the multiply nodes was written in Java to further develop the software model and can be found in the appendix at the back of the report. Using this algorithm, we are now able to generate unrolled graphs for the matrix-solve for the above-mentioned example at the start of this section (4 x 4 lower-triangular matrix solve).

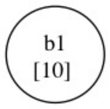


Figure 10 - Unrolled Compute Graph for y_1 (Trivial)

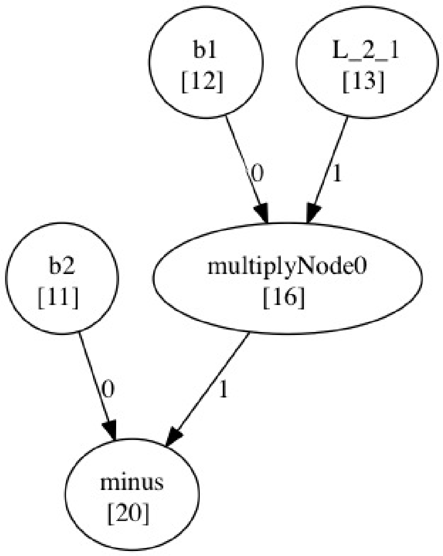


Figure 11 - Unrolled Compute Graph for y_2

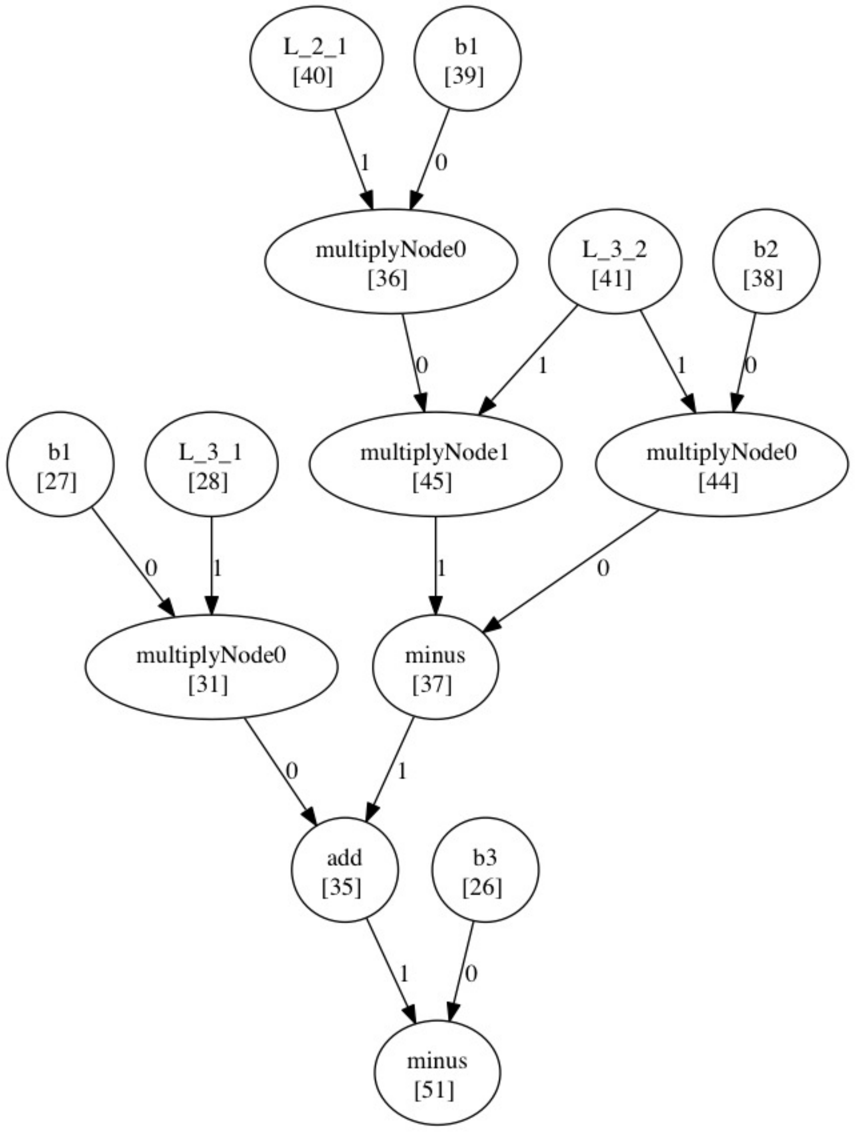


Figure 12 - Unrolled Compute Graph for y_3

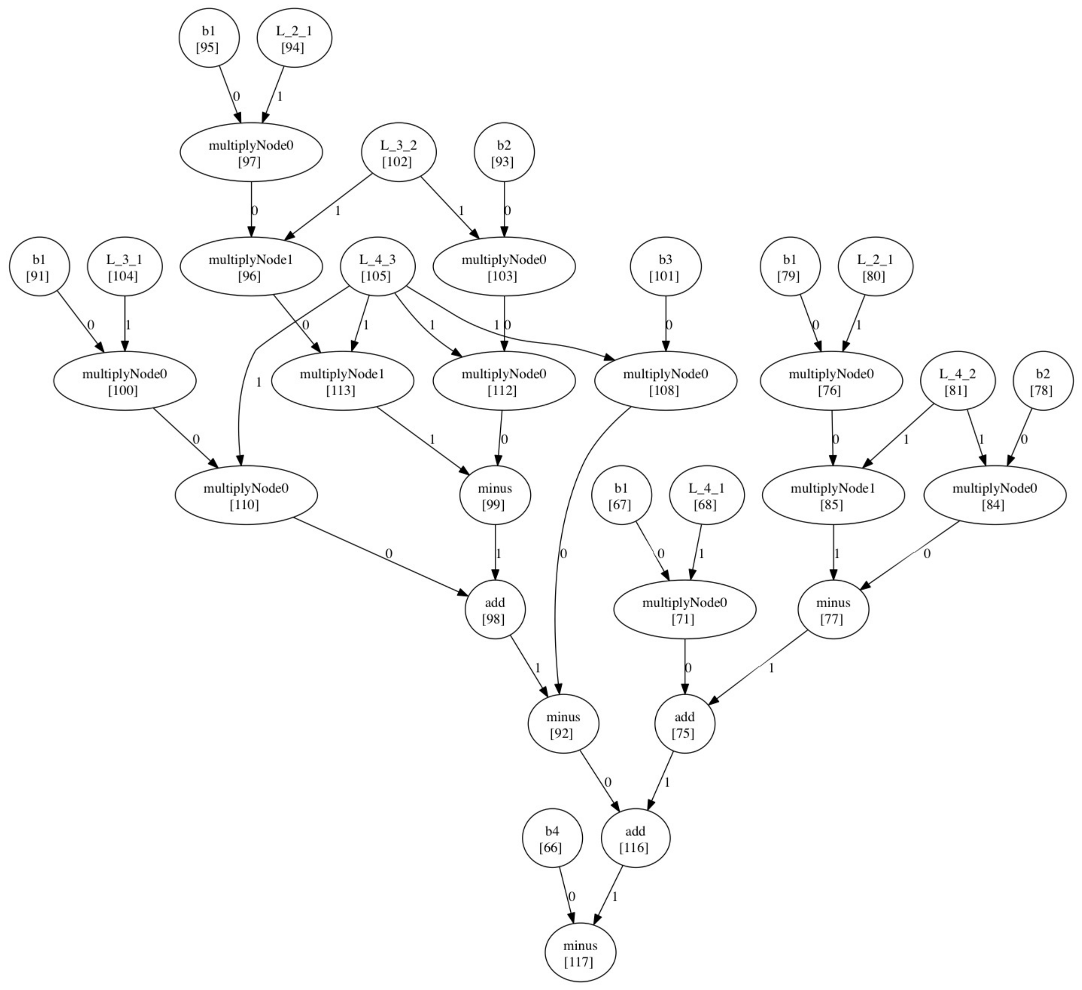


Figure 13 - Unrolled Compute Graph for y_4

The solution above is for a dense matrix equation solve. Currently, the algorithm is under further development such that it is able to handle cases with sparse matrices and output a much smaller compute graph. This is especially useful for SPICE, since matrices from the SPICE Model Evaluation Phase are often sparsely populated. The current version of the code can be found in the appendix at the back of the report[1].

4.2.3 Unroll Graph with Re-Associativity

The aim of re-associativity in this project is to construct a balanced binary tree structure as far as possible. Balanced binary trees are the easiest to construct when the number of inputs to be multiplied (or added, in the case of add trees) is a power of 2. We will start by first taking a look at the simplest non-trivial case of when we have 4 constants that need to be multiplied together. Figure 14 below shows how this multiplication can be carried out in a sequential multiply chain, while Figure 15 shows the more efficient balanced binary tree structure.

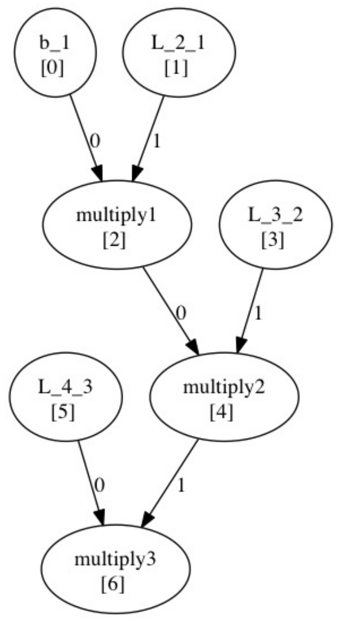


Figure 14 - A multiply chain with 4 inputs

[1]Update: The current version of the code has slight bugs, which, despite generating accurate graphs visually, results in scheduling errors when testing for performance.

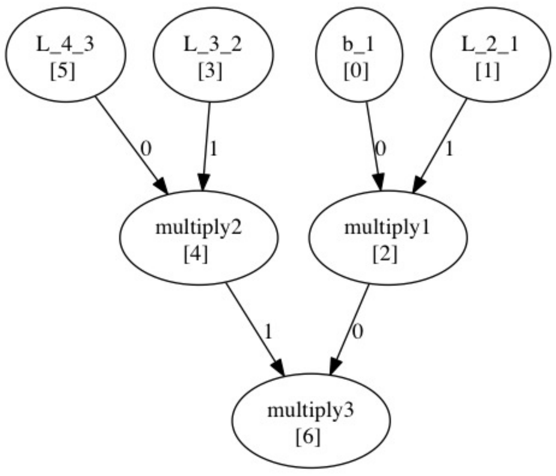


Figure 15 - A balanced binary multiply tree with 4 inputs

The above multiply chain in Figure 14 was converted to the balanced binary multiply tree structure in Figure 15 by an algorithm that we wrote for this project. The algorithm is currently under further development to tackle longer multiply chains, including chains with number of inputs that are not a power of 2. It is not possible to create a completely balanced binary multiply tree structure with these cases, and instead, they have to be composed of combinations of smaller balanced binary tree structures as best as possible. For example, a multiply chain with 13 inputs would be balanced by an 8-input balanced multiply tree, connected with a 4-input balanced multiply tree with the final input being connected in a chain-like fashion, as there is no other alternative. Figure 16 on the next page shows a rough sketch of what this would look like.

The existing code for the algorithm for converting 4-input multiply chains into a balanced tree can be found in the appendix at the back of this report. The algorithm will be further developed in the future to support a complete conversion method to transform an arbritrary length multiply chain into a best possible balanced multiply tree.

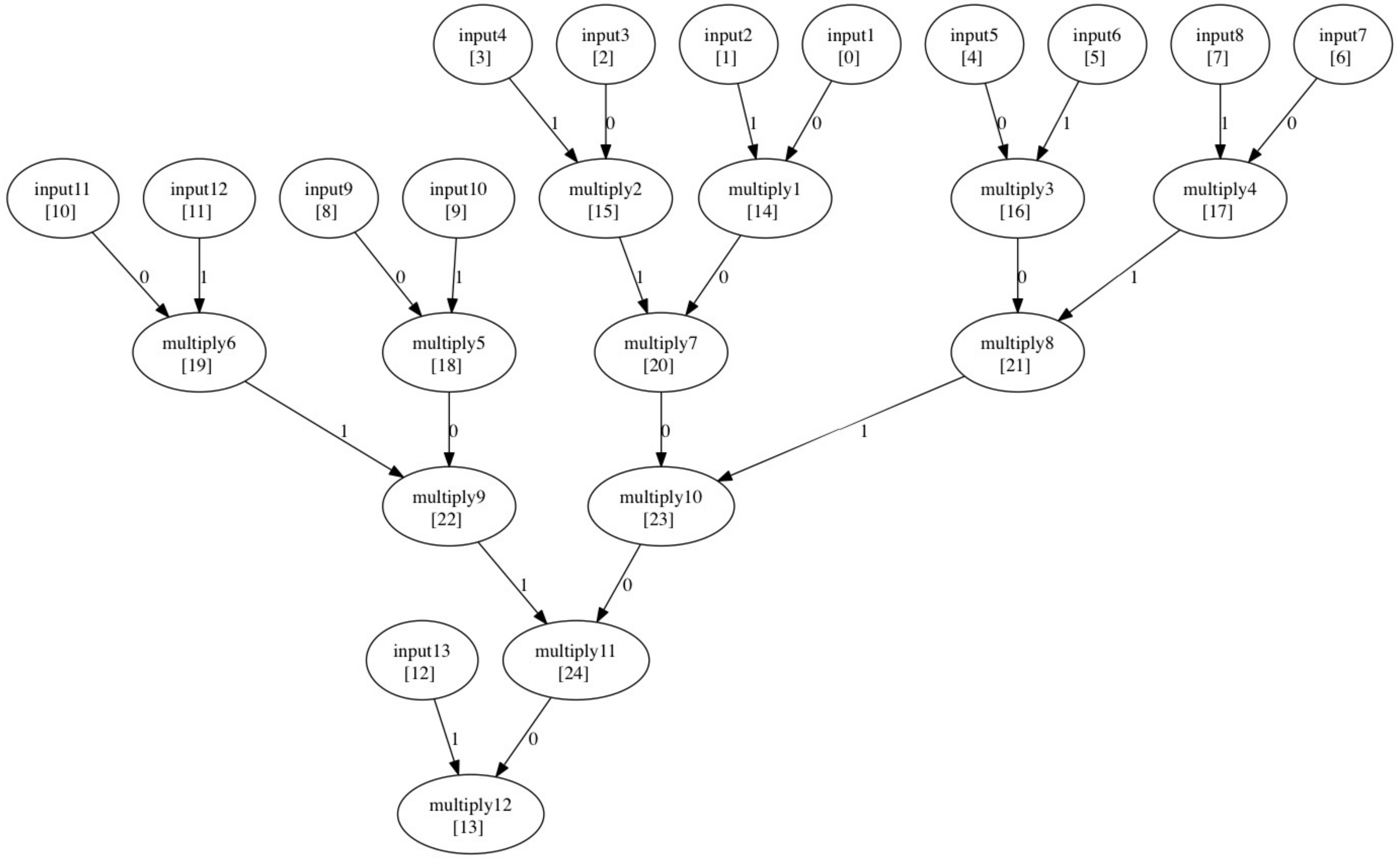


Figure 16 - Best balanced tree output for 13-input multiply chain after re-association

4.3 Performance Comparison

In this section, we analyze some of the performance data we obtain for the different matrix solve methods we have mentioned thus far. As explained in Section 3.3 (Complexities) earlier, we expect a theoretical improvement in the runtime of the SPICE matrix solver.

The performance measurements are made by passing a compute graph to the scheduler, which routes the connections on a virtual hardware simulation of an FPGA. As mentioned before, the scheduler is treated as a black box for the scope of this project, and the clock cycles to execute a compute graph are recorded as displayed by the scheduler.

Unfortunately, as of today, there are some bugs in the algorithms developed, which cause them to not be scheduled and routed correctly by the scheduler. As such, some compute graphs of small matrices (n x n matrices, where $n < 5$) were manually generated and scheduled to get a feel on the initial performance improvements. The table below summarises the clock cycles recorded.

n x n	Cycles (Unroll)	Cycles (Without Unroll)
2 x 2	101	107
3 x 3	327	309
4 x 4	362	481

While the results are not conclusive, and not overly helpful, there is a slight speedup observed with unroll for these small matrix equation solve cases.

5 CONCLUSIONS & FUTURE WORK

5.1 Parallelism Advantages

As demonstrated from the project findings, we conclude that their is indeed potential scope for further development as the speedup advantages delivered from exploiting parallelism are significant. Parallelising the entire SPICE matrix solve phase could play an important role in reducing SPICE runtime significantly, especially for simulation of the future generation of hardware. As devices continue to scale down, the parasitics play a more important role and hence, the matrices in the SPICE matrix solve phase are becoming less sparse. Finding an essential solution to curb the trend of increasing SPICE runtimes is important for the computing industry, and we believe that parallelising the matrix solve process is a viable option.

Furthermore, matrix solve processes are needed in a variety of fields, and not just limited to SPICE. A successful matrix solver running parallel computations through customised hardware can be easily adapted for use with other applications by porting their compute heavy operations to our customised solution. Eventually, this could result in potentially new technological advances in many fields of science and engineering.

5.2 Disadvantages & Shortcomings

One of the key disadvantages of this unroll and re-associativity technique is the strain on resources. Resource requirement can easily get out of control with large cases and careful resource management is required in order to implement a stable and efficient solution.

Also, it is a complex and challenging problem to parallelize computations. Care has to be taken to not end up computing a wrong result, and rigourous testing and debugging is likely required to reach a stable build.

5.3 Future Research & Development

This project does not stop here, and we would continue our development of the UPENN Sparse Matrix Pre-Processor throughout summer this year as part of the university research programme (UROP). Several important features need to be developed/debugged:

1) Unroll algorithm debug for smooth operation with the scheduler.

2) A complete convert-chain-to-tree algorithm, which is capable of constructing (best possible) balanced trees from a chain of operations (adds or multiplies).

3) A verification script to verify the graphs being generated are valid. As more computations are unrolled in larger matrices, it is very difficult to verify that the compute graph is representing the correct compute path due to the large graph size.

4) Tweak to the unroll algorithm to support unroll of sparse matrices as well.

5.3.1 Testing with Larger Examples & Benchmarking Performance

Further rigourous testing needs to be done to ensure that all matrix forms produce the right solve results. SPICE matrices usually contain millions of values, and hence, there is a critical need to test the performance of this novel technique with very large sparse matrices. This would give us a better idea on level of speedup possible with real world problems.

We also need to benchmark performance with key SPICE benchmark matrices. This would give us an idea on what level of parallelism is feasible and is able to give us the maxmium amount of speedup possible.

6 BIBLIOGRAPHY

References

[1] Nachiket Kapre, *SPICE âĂŞ A Spatial Parallel Architecture for Accelerating the SPICE Circuit Simulator*. California Institute of Technology, 2010.

[2] K. S. Kundert, A. Sangiovanni-Vincentelli
Sparse User's Guide: A Sparse Linear Equation Solver. 1988.

[3] Chung-Wen Ho, A. Ruehli, P. Brennan
The modified nodal approach to network analysis. 1975.

[4] Nachiket Kapre, Andre DeHon
Parallelizing Sparse Matrix Solve for SPICE Circuit Simulation using FPGAs 2009.

7 APPENDIX