Algorithmic Techniques

Dense Lower Triangular Solver

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 & 0 & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ L_{n1} & L_{n2} & L_{n3} & L_{n4} & \cdots & L_{nn} \end{bmatrix}$$

Problem Description

A Lower Triangular Matrix is a special type of square matrix where all of the values above the diagonal are zero. Lower triangular matrices most often come up when doing LU decomposition of matrices. Normally, this computation is done serially and does not immediately lend itself to being open to parallelization.

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 & 0 & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ L_{n1} & L_{n2} & L_{n3} & L_{n4} & \cdots & L_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ \vdots \\ b_n \end{bmatrix}$$

The problem is of the form Lx = b, where L is a lower triangular matrix and x and b are vectors. Both L and b are known, and the problem is to solve for the x values that satisfy the equation.

Serial Solution

The serial solution of the problem as implemented on a CPU is called forward substitution. The equations for the forward substitution algorithm are as such:

$$x_1 = \frac{b_1}{L_{11}}$$

$$x_2 = \frac{b_2 - L_{21}x_1}{L_{22}}$$

$$x_3 = \frac{b_3 - L_{31}x_1 - L_{32}x_2}{L_{33}}$$

$$\vdots$$

$$x_n = \frac{b_n - L_{n1}x_1 - \cdots - L_{n,n-1}x_{n-1}}{L_{nn}}$$

As you can see from the equations, this algorithm is inherently sequential as each x value besides the first one depends on previous x values. The pseudocode for the CPU implementation of this algorithm is shown here:

```
for i = 0; i < numRows; i++
        for  j = 0; j < i; j++
                b[i] = b[i] - L(i,j)*x[j]
        end loop
         x[i] = b[i]  / L (i,i)
end loop
```

Related Work

The only paper that we were able to find about solving the x vector in parallel was "Dense Triangular Solvers on Multicore Clusters using UPC" [1]. This solution was done on a CPU Multicore Cluster rather than a GPU, but the idea can be translated over to GPUs. Essentially the idea is similar to pipelining in Computer Architecture. Once a particular x value is known completely, then that value is broadcast to other threads. Once the threads receive that value, then they will go and multiply that value by the corresponding L value in a particular row and then subtract that result off from corresponding b value. Since all of the rows are independent of each other, the multiply and subtract operations can be done in parallel, and thus will theoretically generate the x vector faster.

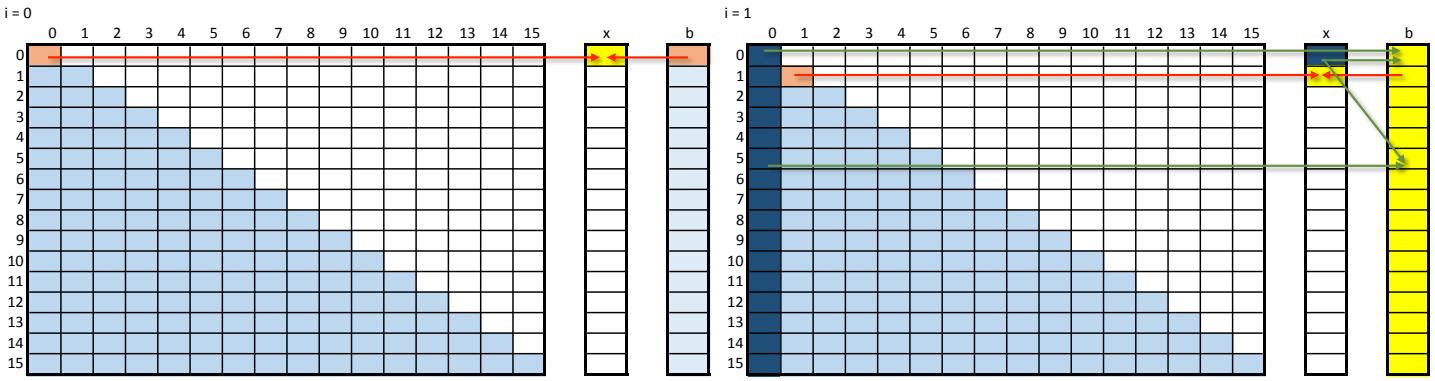
For our first GPU implementation, as will be described later, we essentially implemented the same algorithm as described in the paper. However, a key difference is that rather than having threads update multiple values of b like in the paper, we use a single thread for every value of b (one can also think of it as one thread per row).

Parallelization Strategy

The first GPU implementation that we tried was based on the implementation described in the CPU Cluster Paper from the related work.

Our parallelization strategy was split into three main implementations. These implementations are called GPU Simple, GPU Complex 1, and GPU Complex 2 and will be described below.

GPU Simple



As mentioned before in the serial solution, the ith element in the X vector is dependent on the i - 1 element. However, in order to obtain the complete solution for an x element, we have to calculate another component.

$$x_1 = \frac{b_1}{L_{11}}$$

$$x_2 = \frac{b_2 - L_{21}x_1}{L_{22}}$$

$$x_3 = \frac{b_3 - L_{31}x_1 - L_{32}x_2}{L_{33}}$$

$$\vdots$$

$$x_n = \frac{b_n - L_{n1}x_1 - \cdots - L_{n,n-1}x_{n-1}}{L_{nn}}$$

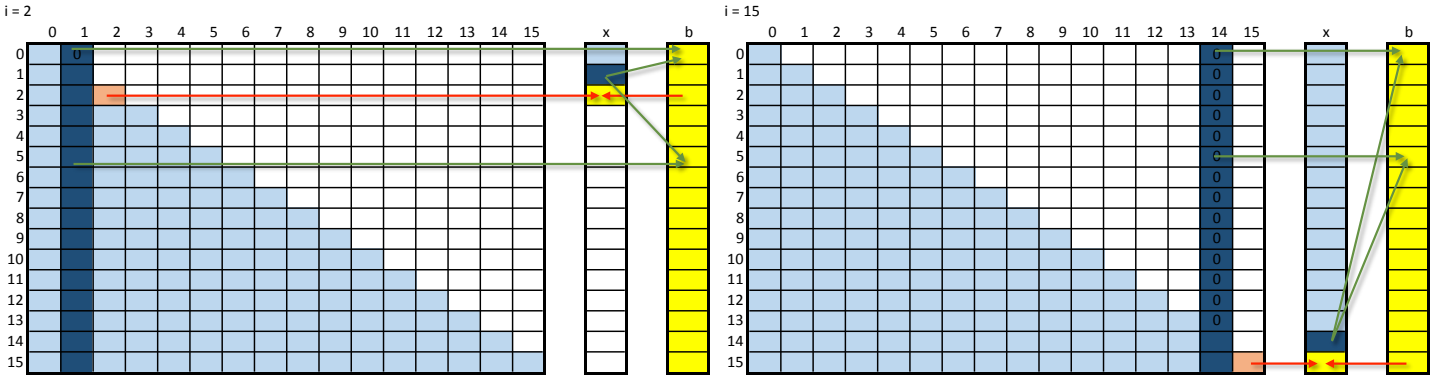
The element x_2 is dependent upon x_1 and b_2/L_{22}. The latter can be calculated independently using the diagonal elements of the L matrix and the corresponding elements in the B matrix. The aforementioned figures illustrate the following algorithm in detail:

```
for i = 0; i < numRows; i = i + 1
        Kernel (1 row per thread):
                if i != 0
                        b[y] = b[y] – L[y, i-1]*x[i-1]
                end if
                if y = i
                        x[i] = b[i] / L[i,i]
                end if
        end Kernel
end loop
```

For every row, we launch a kernel that does one to two operations. All threads update b, but one thread in each kernel launch updates the output vector. The operation depends on the boundary conditions. If the element is a diagonal element it also updates the x vector, as shown in the above figure. E.g In the leftmost figure, the value of the first element in vector x is calculated using the diagonal element 0,0 in matrix L (highlighted orange) and the first element in vector b.

If the element is not a diagonal element, it takes the value in the column in L and uses the corresponding value in x and modifies the element in b vector. The resultant modified value is then used in the next iteration. The figure on the right shows the green arrows on the first row. The blue highlighted cells are being used to modify the elements in the b vector.

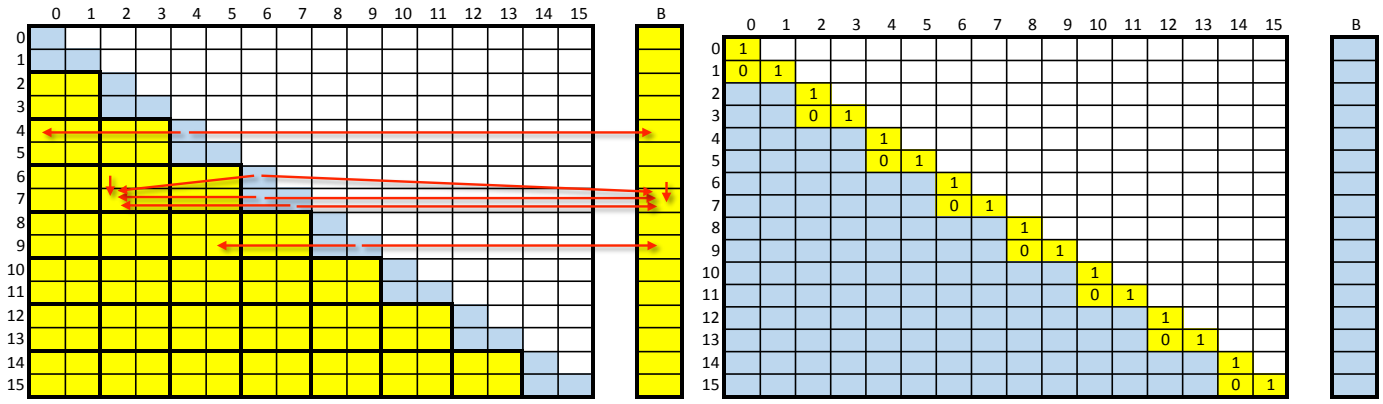
The same thread executes the independent operation procedure on the diagonal elements and calculates the element in vector x.



The above figure to the left further shows that every thread follows the same pattern. The last iteration for i=15 on the right shows that the diagonal element is used to calculate the value for the element in vector x.

GPU Complex 1

The next implementation that we tried utilized two different kernels, and will be referred to as GPU Complex 1 from now on. The first kernel (which is called triangle solver) was used to extract parallelism by breaking data dependencies so that there would be enough parallel operations to justify using a GPU in the first place.



It extracts parallelism by performing row operations on the matrix to modify its data. In particular it is subtracting rows from each other in order to solve for the "triangles", as one can see from the above image on the right. For row 4 (and all of the other even rows) of the L matrix and b vector in the left image, all of the data in the yellow squares are updated according to the following equations:

$$L[y,x] = \frac{L[y,x]}{L[y,y]}$$

$$b[y] = \frac{b[y]}{L[y,y]}$$

Looking at the special case of when x = y, we can see that we get the following result:

$$L[y,y] = \frac{L[y,y]}{L[y,y]} = 1$$

Since we are solving 2x2 triangles in this first kernel, the operations above that were performed on the even rows have essentially solved the upper half of the triangle by dividing all of the values in an even row by the value at the end of that row (where y is equal to x).

For the odd rows, in order to get the element on the end (where y is equal to x) to be 1, and the element directly the left of it to be 0, the following operations are performed:

$$L[y,x] = \frac{L[y,x] - \left(\frac{L[y,y-1]}{L[y-1,y-1]}\right) * L[y-1,x]}{L[y,y]}$$

This equation will first scale the even row that is directly above the odd row, and then subtract off the corresponding elements scaled elements from the odd row elements. This will cause the lower left element of the triangle to be 0. Next, the equation divides all of the values in a row by the value at the

end so that the lower right value of the triangle is 1. The following two equations give an example of these:

$$\text{If x = y, } L[y,y] = \frac{L[y,y] - \left(\frac{L[y,y-1]}{L[y-1,y-1]}\right) * L[y-1,y]}{L[y,y]} = \frac{L[y,y]}{L[y,y]} = 1$$

$$\text{If } x = y\text{-}1, L[y,y-1] = \frac{L[y,y-1] - \left(\frac{L[y,y-1]}{L[y-1,y-1]}\right) * L[y-1,y-1]}{L[y,y]} = \frac{L[y,y-1] - L[y,y-1]}{L[y,y]} = 0$$

Additionally, since any operation that is done on a row of the matrix must also be done on the corresponding row of the vector b, we get the following equation for updating the b values for even rows:

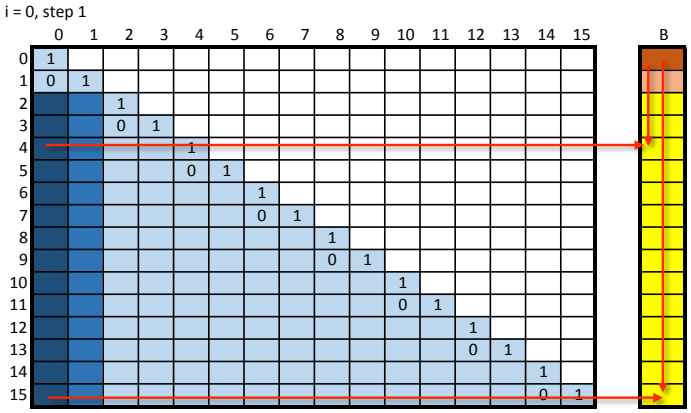
$$b[y] = \frac{b[y] - \left(\frac{L[y,y-1]}{L[y-1,y-1]}\right) * b[y-1]}{L[y,y]}$$

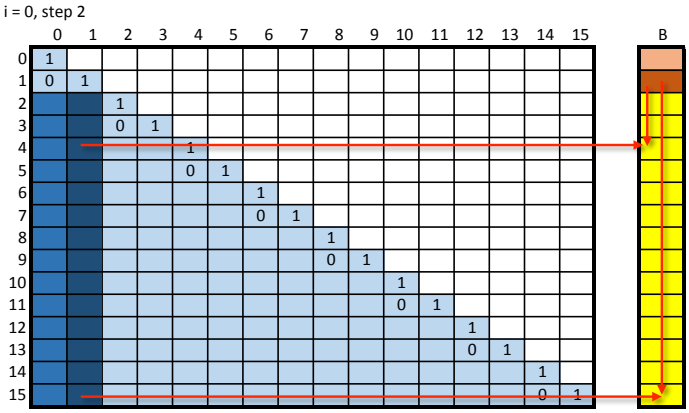
The pseudocode of the first kernel is given here:

```
Kernel (4 element square per thread):
        for each element
                if validRow(y) && validColumn(x)
                        if y % 2 = 0
                                L[y,x]=  L[y,x]/L[y,y]
                                if x = 0
                                        b[y]=  b[y]/L[y,y]
                                end if
                        else
                                L[y,x]=(L[y,x]-(L[y,y-1]/L[y-1,y-1] )*L[y-1,x])/(L[y,y])
                                if x = 0
                                        b[y]=(b[y]-(L[y,y-1]/L[y-1,y-1] )*b[y-1])/(L[y,y])
                                end if
                        end if
                end if
        end loop
end Kernel
```

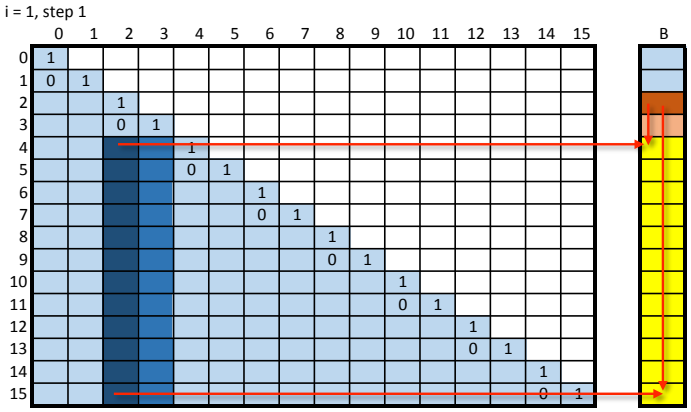
As you can see from the pseudocode, we are launching the threads such that each thread corresponds to 4 elements (a 2x2 block). We did this because the triangle size being 2x2 allowed for all of the operations to be independent of one another, and thus be done in parallel.

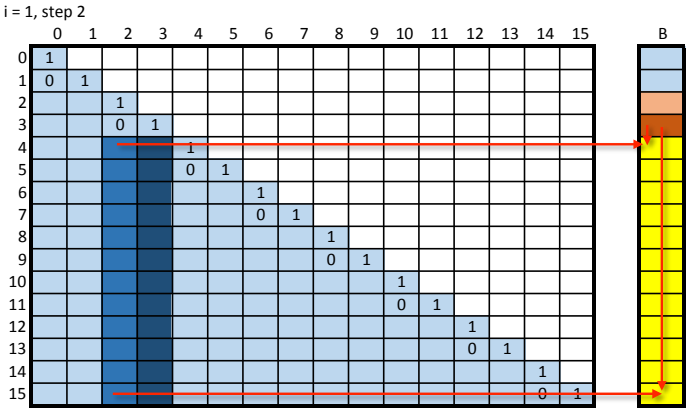
The next kernel that we used proceeded to eliminate the column values two at a time versus one column at a time for the GPU Simple implementation. Having previously modified the matrix such that there were now triangles on the end of each row allowed for us to do two operations at once. Thus, rather than having to launch a kernel for every column that we have in the L matrix like for the GPU Simple, we will launch it exactly half as much now. Just like for the GPU Simple kernel, there is a single thread launched for each row of the matrix.



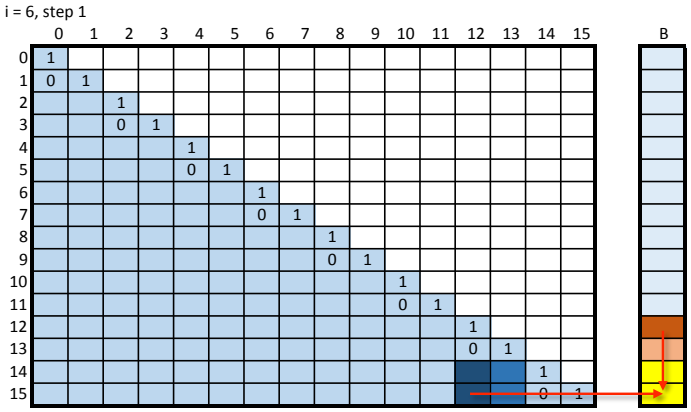


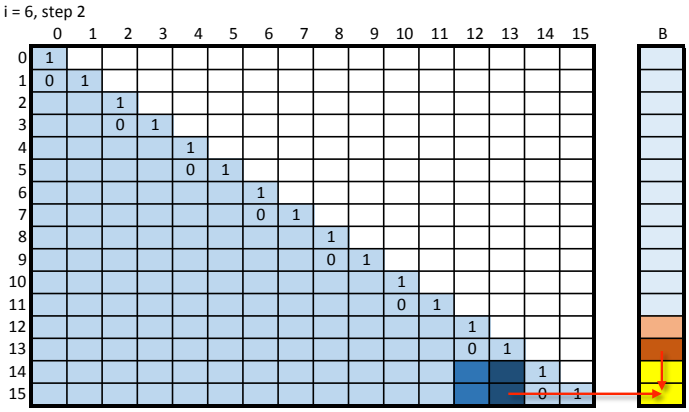
The first iteration of this kernel will use the values of $b_0 = x_0$ and $b_1 = x_1$ that were solved from the previous kernel in order to zero out all of the values shaded dark blue in column 0 and column 1 in the images above. As you can see from the images above, for step 1 and step 2 of the first iteration the dark blue value from each of the rows will be multiplied by the dark orange value of b (which is actually a solved value of x), and then that computed value will be subtracted from the b value in that row. After step 2 has completed for iteration one, the elements b_2 and b_3 will now be the solved values of x_2 and x_3.





Just like for the previous iteration, step 1 will multiply the dark blue elements by the dark orange elements and subtract off the computed value from b in order to generate two more values of x.





Finally, the last iteration is shown above, which generates the final two values of x. After this iteration has completed, all of the values in b are now the solved x values of our initial problem since the matrix L is now the identity matrix.

```
for i = 0; i < numRows/2; i = i + 1
        Kernel (1 row per thread):
                if validRow(y) && validColumn(i*2)
                        b[y] = b[y] – L[y,i*2]*b[i*2] - L[y,i*2+1]*b[i*2+1]
                end if
        end Kernel
end loop
x = b;
```

The pseudocode for this kernel is shown above, which gives a general idea of the algorithm described above. As you can see from above, each value of b is subtracted off from two values of b that are multiplied by two values of L.

GPU Complex 2

The first step of the GPU Complex 2 algorithm is the same as the first step of the GPU Complex 1 algorithm; the triangle solver. This way the diagonal is initially set to 1s and every odd numbered row of the inner diagonal is set to 0s. The second step involves calling the kernel iteratively. Each time the kernel is called, the algorithm performs row operations meant to turn certain elements into zeros:

$$d = y - (y\%i) - 1 - j$$

$$L[y,x] = L[y,x] - \sum_{j=0}^{i-1} L[d,x] * L[y,d]$$

$$b[y] = b[y] - \sum_{j=0}^{i-1} b[d] * L[y,d]$$

Effectively, the idea is to use row operations to multiply elements directly below every other identity submatrix (the solved triangles) by the 1s in that submatrix, and then subtract them from their own row, making them 0s. However, the entire row outside the submatrix needs to be multiplied by one of those elements, meaning they are row multipliers. And since multiple elements in a row need to be transformed to 0, multiple row operations need to be performed. This effectively performs an operation similar to matrix multiplication, subtracting the dot product of the row multiplier elements by the appropriate row elements, from the specific element being modified. This is the same operation performed on vector b. After each iteration is complete, the final matrix will be the identity matrix and vector b will contain the solution. The pseudocode is shown below:

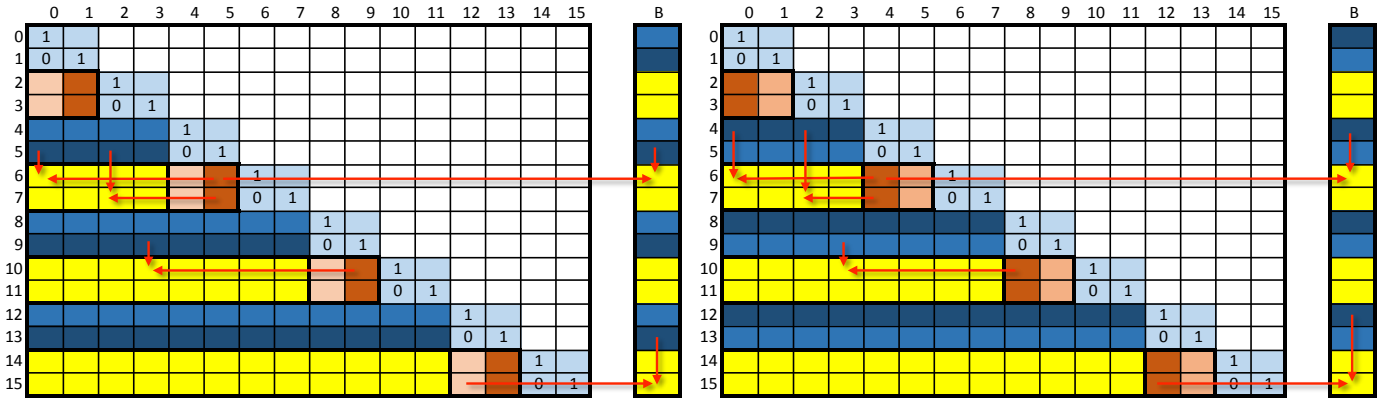
```
Second Step:

for i = 2; i < numRows; i = i*2
        Kernel (1 element per thread):
                if validRow(y) && validColumn(y,x)
                        for j = 0; j < i; j = j + 1
                                d = y-(y%i)-1-j
                                if x = numRows – 1
                                        b[y] = b[y] - b[d]*L[y,d]
                                else
                                        L[y,x] = L[y,x] - L[d,x]*L[y,d]
```

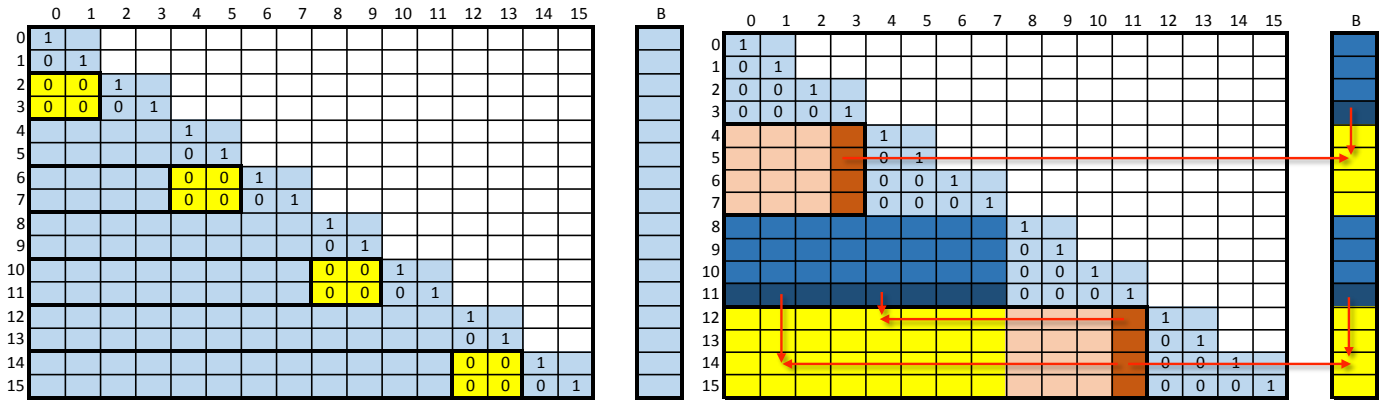
```
                                end if
                            end loop
                        end if
                    end Kernel
end loop
x = b;
```

At the first iteration, the value of index i is 2. At each iteration, the value of i doubles until it is equal to the number of rows in the matrix. Valid rows are every other i rows. The lower i rows take the dot product of the matrix formed by the upper i rows and the matrix formed by the row multipliers. Valid rows are only every other upper i rows, which can be determined using mod operations. Validity of the column depends on the row. The row multipliers cannot be modified until all the rest of the elements in the same row are modified. So only columns to the left of the row multipliers are valid. However, once these valid columns have been modified, there is no reason to actually modify the row multipliers since we know the row operations would transform them to 0 anyway. So we can instead assume they become 0 and save ourselves extra reads and writes to global memory. Since these elements are only used to transform later row multipliers to 0, this is not a problem.

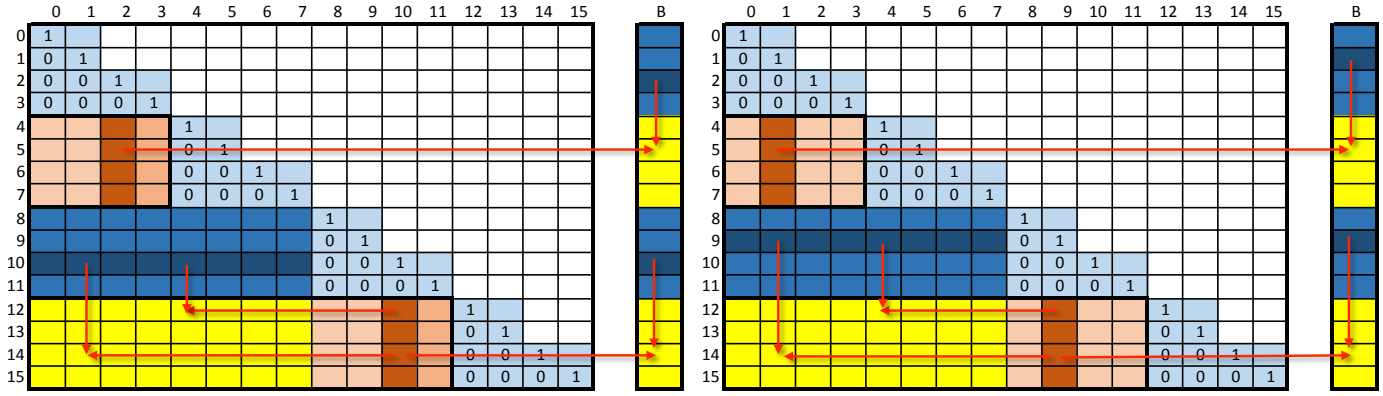
A toy example of step 2 of the GPU Complex 2 algorithm is shown below:



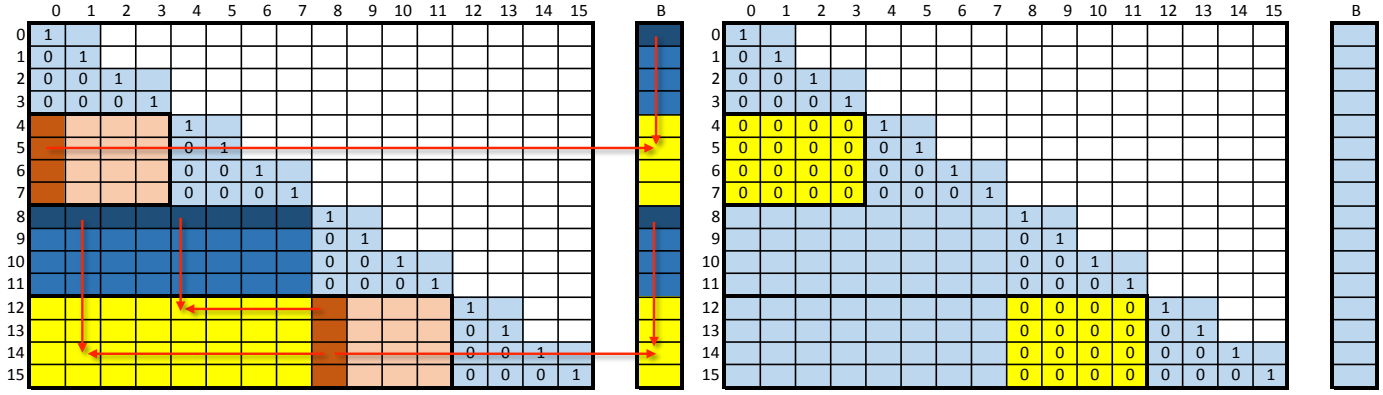
The above two figures show the first iteration where i = 2. The yellow highlighted elements are the ones being modified. These highlighted elements in the L matrix are of valid rows and columns and form solution matrices from the dot product of the row elements (blue matrices above the highlighted matrices) and row multipliers (orange matrices to the right of the highlighted matrices). Note that the 2x2 submatrices above the orange matrices are identity matrices. The red arrows show example operations formed on a few random elements. Two arrows pointing to a yellow element mean they are being multiplied together and subtracted from the value currently in the yellow element. Since i = 2, the dot product is two terms. The figure above on the left shows the first step of the dot product, pertaining to the first term. The dark blue and dark orange elements are the ones being accessed during this step. The figure above on the right shows the second step of the dot product, pertaining to the second term. Different elements are dark blue and dark orange, pertaining to different elements being accessed for the second term. These same operations are also performed on the b vector. Since the values highlighted yellow are modified multiple times, they are loaded into registers at the beginning of the iteration and written back at the end. This saves memory bandwidth by minimizing the reads and writes to global memory at these locations. This becomes more significant at larger values of i.



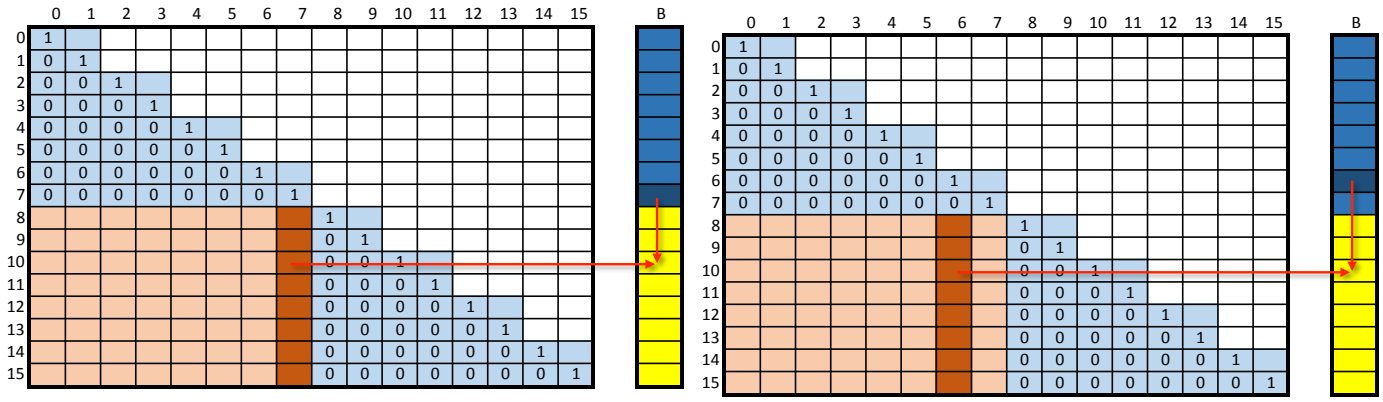
The figure above to the left shows the final step of the first iteration where the same row operations used in the previous two figures are used to transform the row multipliers to 0. However, as mentioned earlier, the program just assumes these elements are zero. The figure above on the right shows the beginning of the second iteration, where the algorithm re-calls the kernel. This is the first step of the dot product for i = 4, pertaining to the first term. There will now be four terms in each dot product. In addition, only the last 4 rows contain valid elements in the L matrix. However, rows 4-7 are still valid in vector b.



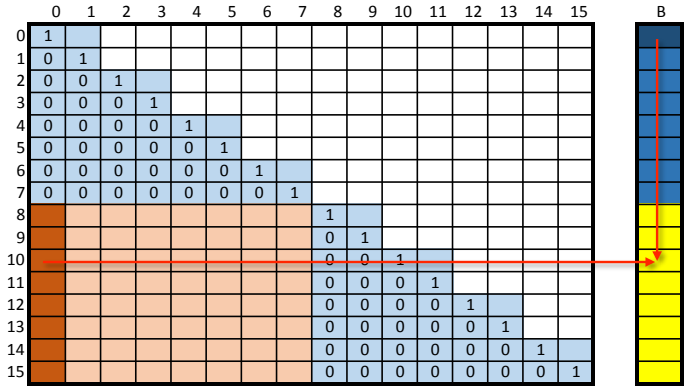
The figures above show the second and third steps of the dot product for i = 4, pertaining to the second and third terms.



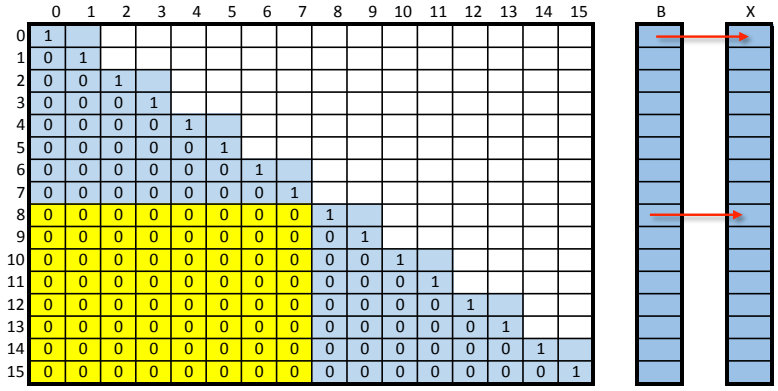
The figure above on the left shows the fourth step of the dot product for i = 4, pertaining to the fourth term. The figure above to the right shows the final step of the second iteration where the same row operations used in the previous four figures are used to transform the row multipliers to 0. However, as mentioned earlier, the program just assumes these elements are zero.



The figures above show the beginning of the third and final iteration. These are the first and second steps of the dot product for i = 8, pertaining to the first and third terms. Note there are no modifications made to the L matrix during this last iteration, only modifications to the b matrix.



The figure above shows the eighth and last step of the dot product for i = 8, pertaining to the eighth term. Steps between the second and eighth are not shown.



The figure above to the right shows the final step of the second iteration where the same row operations used in the previous four figures are used to transform the row multipliers to 0. However, as mentioned earlier, the program just assumes these elements are zero. Once this last step has been completed, the L matrix is an identity matrix, meaning b contains the solution. The algorithm then writes the b matrix into the x matrix.

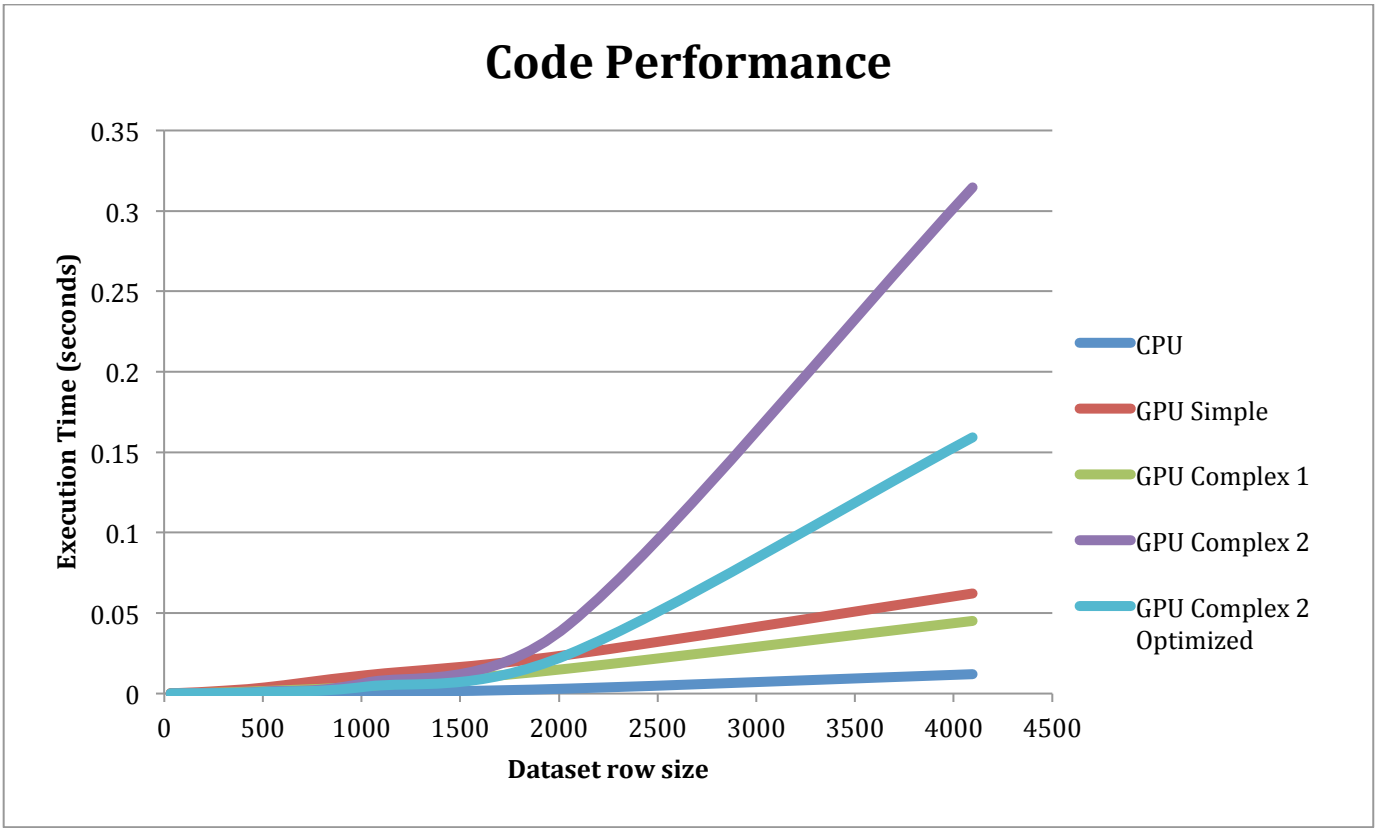
Parallel Optimizations

Because the GPU Complex 2 algorithm uses matrix multiplication as its fundamental operation, it lends itself to shared memory tiling. Since the block size is 32x32, the maximum size in CUDA, the tiles are also 32x32. At the start of an iteration, the available threads in a block load a square tile of the row elements and a square tile of the row multipliers into shared memory for use by all threads in that block. After enough steps of the dot product have been performed to use all the elements stored in shared memory, the kernel reloads another tile. This process continues until all dot products in the matrix multiplication have completed.

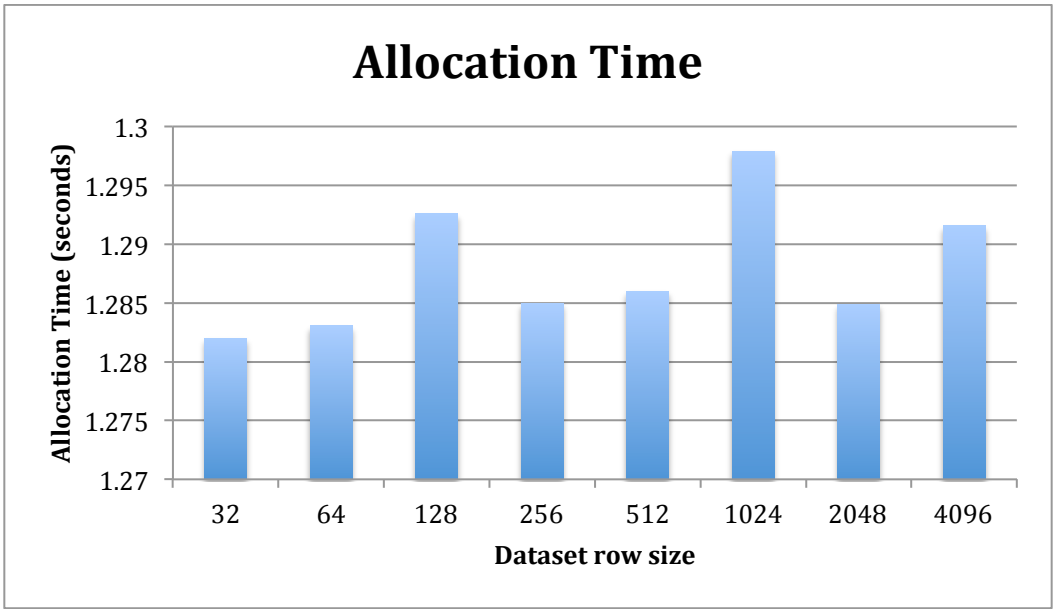
One major difference between this and normal matrix multiplication is that the multiplier and result matrices are all submatrices of a single matrix. As a result, these matrices need to be indexed using modular arithmetic. One downside of this is that there is not a linear mapping from elements in the matrix to elements in shared memory when threads load from global memory. This becomes a problem when the block overlaps valid and invalid rows and columns, which will occur when $i < 32$. So this is avoided by ensuring shared memory tiling only occurs for iterations where $i \geq 32$. Prior to that point, the program uses the GPU Complex 2 algorithm without tiling. Since the number of sequential global memory loads is proportional to i, this is less of an issue since there are fewer sequential global memory loads in these early iterations.

Evaluation

All of our code was run using the GEM GPU cluster that we were provided access to. All of the GPU times are just for the kernel to finish executing, and do not take into consideration the time it took to allocate and copy data to/from the device.



As can be seen from the graph above, the CPU implementation performed the fastest out of all of the implementations that we tried. Looking at the larger datasets gives us the most revealing data about the implementations. The GPU Complex 1 was our best performing GPU implementation, while the GPU Complex 2 was our worst GPU implementation The GPU Simple implementation was in the middle. After optimizing the GPU Complex 2 kernel through the use of shared memory, we got a speedup of approximately 2 like we had expected. We expected that the GPU Complex 2 code would perform better than the GPU Simple and GPU Complex 1 due to it performing much less kernel launches and having a lot more parallelism but that was unfortunately not the case. We believe it is due to the fact that we are doing many more global memory accesses than we are in the other implementations.



As can be seen from the graph above, the time it took to allocate the data on the device took longer than the CPU version to execute. As a result, one would not be performing this operation on a GPU unless it was doing other operations to the data as well.

Conclusion

While our most complex GPU implementation performed worse than the CPU implementation and all of the GPU implementations, it is still important to see those results. We now know that the most complex implementation is not always the best, and that sometimes it is better to go with a simpler implementation. Like we mentioned earlier, given that the time it takes to allocate the data on the device is much more than it takes to run the actual CPU implementation of the code, it is not realistic to perform this operation on a GPU unless you are using that same data for many different operations on the GPU. In the end, we believe that our GPU Complex 2 had too many global memory accesses compared to the other GPU implementations, thus giving us a massive slowdown compared to them. While the use of shared memory provided us with a speedup of 2, it was not enough of a performance boost to get us close to the other GPU implementations or the CPU implementation.

References

[1] http://upcblas.des.udc.es/publications/ICCS11.pdf