LU FACTORIZATION MEASUREMENT AND OPTIMIZATION

John Cole

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

In this paper, we analyze the runtime performance of the LU factorization algorithm. To do this, we first test the basic algorithm, as well as a blocked algorithm across different compiler flags. Then, we add other specific optimizations to the blocked algorithm code in order to get the best runtime performance possible.

1. INTRODUCTION

Motivation. LU factorization is most often associated with the problem of Gaussian Elimination. In the problem of Gaussian Elimination, one needs to solve the system of equations represented by $Ax = b$. It becomes necessary to factor a given matrix A into two other matrices, a lower and an upper triangle matrix, which when multiplied together, produce the original square matrix. LU factorization is the process that accomplishes this. Once this factorization is done, the resulting L and U matrices can be used to solve for x by solving $Ly = b$, then $Ux = y$.

Previous Work. The main reference for this paper is James Demmel's *Applied Numerical Linear Algebra* [1]. In this reference, LU factorization is explained, as well as methods to use blocking to improve performance. The specific algorithm for LU factorization is presented, as well as the algorithm for blocked LU factorization. No performance plots were included with these algorithms, which led us to this paper.

What We Are Going to Do. First, we will test the baseline algorithm from [1] across two different compilers and 4 compiler optimization flags. Then, to accomplish better performance, we will utilize the blocking algorithm documented in [1]. Using this algorithm, we will evaluate the results across blocksizes of 2, 3, and 4. In addition, for each blocksize, we will attempt to unroll certain bottleneck portions of code to produce even better results.

Organization. In Section 2, we will provide the basics behind both the baseline and blocked algorithms of LU factorization. In addition, we will provide a cost analysis of the algorithm. Next, in Section 3, we will explain in detail the exact optimizations made to each blocksize, including our analysis of where the bottleneck portions of our code are. In Section 4, we will present exact measurements we made, and the results of our experiments with several runtime performance plots. And finally, in Section 5 we will provide some conclusions from our project.

2. LU FACTORIZATION ALGORITHMS AND COST ANALYSIS

The LU factorization problem seems be a straightforward algorithm. Our paper only addresses the baseline and blocked algorithms presented in detail in [1].

2.1. Baseline Algorithm

The baseline algorithm for LU factorization consists of 3 parts for each iteration. These three steps are completed at each member of the matrix's diagonal in order from top left to bottom right.

Pivoting. Later in the algorithm, it becomes necessary to divide several values of the matrix by this diagonal member. So this number cannot be zero. In our implementation, we use partial pivoting, which means this diagonal member is the largest absolute value of all member of its column below it. Since with this method we begin moving rows of the matrix, it becomes necessary to form a permutation matrix P along with L and U, so that the original matrix A is equal to PLU. So with partial pivoting, we must find the maximum absolute value of the current column below the diagonal and switch those two entire rows. Along with this switch, we must switch the corresponding rows in P to preserve P.

Division. Next, we divide each member of the current column below the diagonal by the diagonal member. Since we have just done pivoting to ensure this divisor is the largest absolute value, after this division, the absolute value of all cells in the current column below the diagonal will be less than or equal to 1. Since division is much more expensive

than multiplication, in our implementation, we will find the inverse of the divisor, and use this value in our multiplications. Assuming we are at column i, the following equation represents this step. Notice that the original matrix is overridden. L and U will be extracted after the algorithm is completed.

$$\begin{aligned} invDivisor &= \frac{1.0}{A(i,i)} \\ A(i+1:n,\ i) &= A(i+1:n,\ i) * invDivisor \end{aligned}$$

Update. Finally, we must update the lower right-hand portion of the array. This is done with a multiplication and a subtraction for each cell of this lower right-hand array. This is done through the following equation:

$$\begin{aligned} A(i+1:n,\ i+1:n) = A(i+1:n,\ i+1:n) \\ -(A(i+1:n,\ i) * A(i,\ i+1:n)) \end{aligned}$$

2.2. Blocked Algorithm

Now, moving to a blocked implementation of the LU factorization algorithm, we are no longer working with a single member of the diagonal. Now we are working with (bxb) square along the diagonal, where b is the blocksize. Once again, the blocked algorithm for LU factorization consists of 3 steps, which are described below, assuming we are at position i in the matrix. For reference, the (bxb) block is labeled as A_{11}. The series of rows to the right of A_{11} spanning rows i through $(i+b)$ is labeled as A_{12}. The series of columns below A_{11} spanning columns i through $(i+b)$ is labeled at A_{21}. And finally, the rest of the matrix to the lower right is labeled as A_{22}. Pieces of L and U with the same subscripts have the same dimensions.

Factorize A_{11} and A_{21}. We will first need to factorize these two matrices together using the baseline algorithm to form L_{11}, L_{21}, and U_{11}. While creating these, A_{11} and A_{21} are overridden according to the baseline algorithm.

Solve for A_{12}. To do this, we first must find L_{11}^{-1}. Then, we form $A_{12} = L_{11}^{-1} * A_{12}$.

$$A(i:i+b, i+b:n) = L_{11}^{-1} * A(i:i+b, i+b:n)$$

Update. Just like in the baseline analysis, before we can step b to the next iteration, we must update the lower right-hand matrix A_{22}. In this case, we do this with a matrix matrix multiplication, and a subtraction.

$$\begin{aligned} A(i+b:n, i+b:n) = A(i+b:n, i+b:n) \\ -A(i+b:n, i:i+b) * A(i:i+b, i+b:n) \end{aligned}$$

2.3. Cost Analysis

For this paper, we decided to only include additions, multiplications, and comparisons as floating point operations in our analysis. we included comparisons because most comparisons subtract the two sides of the comparison, then test if the result is above or below zero. However, we decided to not include the divisions of the algorithm because most projects do not include it in floating point operations analysis.

- **Baseline Algorithm**

  Pivoting. The pivoting section of the algorithm uses only comparisons, no additions or multiplications. At each iteration, the current diagonal member must be compared to every cell below it. This means by summing along the entire matrix, there will be $\sum_{i=1}^{n-1} i$, or $\frac{n(n-1)}{2}$ comparisons.

  Division. This step involves one division, then a multiplication for every cell below the diagonal. Since we are ignoring divisions, these n divisions do not count toward our operations count. However, the $\frac{n(n-1)}{2}$ multiplications will be counted.

  Update. Finally, the update step will have a multiplication and an addition for each of the lower right-hand matrix's cells. This means there will be $2\sum_{i=1}^{n-1} i^2$, or $\frac{n(n-1)(2n-1)}{3}$ additions and multiplications.

  Total Cost. By summing each step's cost, the total operations in the baseline algorithm is $\frac{2}{3}n^3 - \frac{2}{3}n$.

- **Blocked Algorithm**

  Factorize A_{11} and A_{21}. By using a small variation on the analysis of the baseline algorithm, the number of operations in this step is $nb^2 + nb - (1/3)b^3 - b^2 - (2/3)b$.

  Solve for A_{12}. This is equivalent to solving $(n-b)$ systems of equations, one for each column. This equates to $(n-b) * b * (b-1)$ operations.

  Update. Once again, this is a matrix matrix multiplication, resulting in $(n-b)^2 * (2b)$ operations. An important thing to notice is that this step contains the bottleneck of this algorithm. When performed on the entire array, this step is on the order of $\Theta(n^3 b)$.

  Total Cost. To compute the total cost of the blocked algorithm, we first express a single step of the iteration as $C(n,b) = 2n^2b - 2nb^2 + (2/3)b^3 - (2/3)b + C(n-b, b)$. Then, we can denote $b * m = n$ and compute $\sum_{i=1}^{m} C(bi, b)$, which gives a final cost of $C(n,b) = (2/3)n^3 - (2/3)n$.

So the cost analysis of both algorithms is identical, and our runtime analysis can be run on the same number of operations, irregardless of the algorithm used.

3. PROPOSED METHOD

As described previously, in these experiments, we tested the performance of the basic and blocked LU factorization algorithm presented in [1]. As noted previously, the lower matrix update, the third step of the blocked algorithm, dominates the recurrence. So for each of the different blocksizes, we attempted to modify this step in a way to speed up the performance.

Blocksize 2. Inside this update step, there is a matrix matrix multiplication. In the basic blocked implementation, we perform this multiplication with a simple triple loop. So as our first modification, we blocked this matrix matrix multiplication, and unrolled the innermost loop completely. As another optimization, we unrolled the innermost loop of the blocked matrix matrix multiplication again, but organized the multiplications and the additions so that the inner most loop indexing would be reversed, thereby reusing several local variables.

Blocksize 3. For the blocksize equal to 3, we again blocked the matrix matrix multiplication and unrolled the innermost loop to attempt to find some optimization.

Blocksize 4. Finally, with the blocksize equal to 4, we once again blocked the matrix matrix multiplication into chunks of 4x4 matrix matrix multiplications, which were blocked into 2x2 sub-blocks. The major optimization for this experiment was to utilize the vector instruction set provided with the Intel C++ compiler icpc to accomplish this 4x4 matrix matrix multiplication.

4. EXPERIMENTAL RESULTS

In this section, we will describe the results of our experiments with several graphs. First, we will begin with the setup of our experiment.

Setup. For these experiments, we ran the tests on our personal computer, a Dell Pentium M with Centrino technology running a Mandrake operating system, a version of Linux. Our processor is 1400MHz, with a maximum performance of 1400 mega floating point operations per second.

Compiler Flags. To start our experiment, we wanted to answer the question of which compiler optimization flag option would produce the best results. To do this, we decided to run the baseline implementation of the LU factorization algorithm across several different compiler flags. In addition, we used another C compiler, the Intel's academic version, icpc. From these results in Figure 1, one can see that the O2 and O3 optimizations produce similar results for the gcc compiler. In addition, it appears that for at least this problem, the icpc compiler produces similar results across all compile flags. From these results, we chose to run the rest of our experiments with the command line options "gcc -O2 *.c".

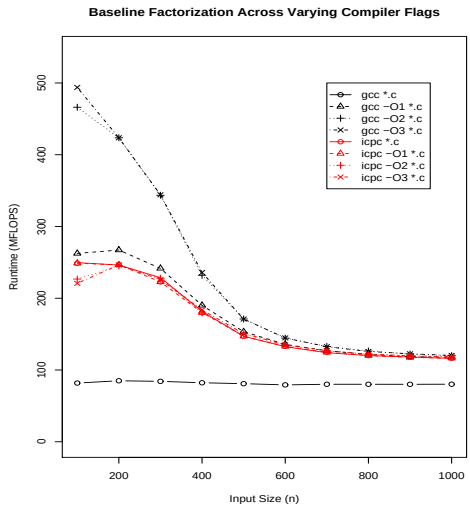

Fig. 1. LU Baseline Algorithm plotted across varying compiler optimization flags.

Basic vs. Blocked. Next, we wanted to see how the blocked LU factorization algorithm would perform against the baseline implementation. We tested the baseline implementation against blocksizes of 2, 3, and 4. From this graph in Figure

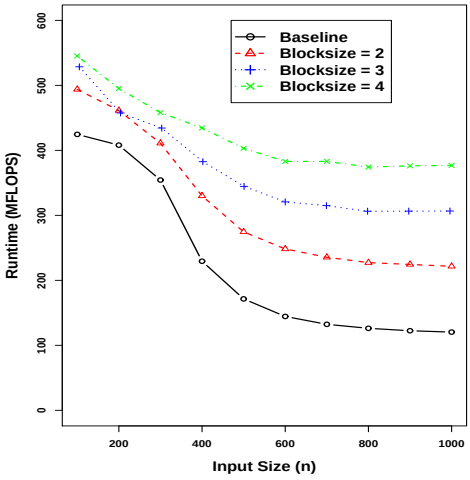

Fig. 2. LU Baseline algorithm plotted against the Blocked algorithm with varying block sizes.

2, it is easy to see that blocking does improve the MFLOP rate as the blocksize increases. From the baseline implementation to a blocksize of 4, we saw almost a speedup of 4 times the baseline MFLOP rate.

Blocksize 2. Next, we tried to speed up the bottleneck portion of the blocked algorithm with a blocksize of 2. We

wanted to see how important this bottleneck was to the MFLOP rate, and if we could improve it at all. From this graph in

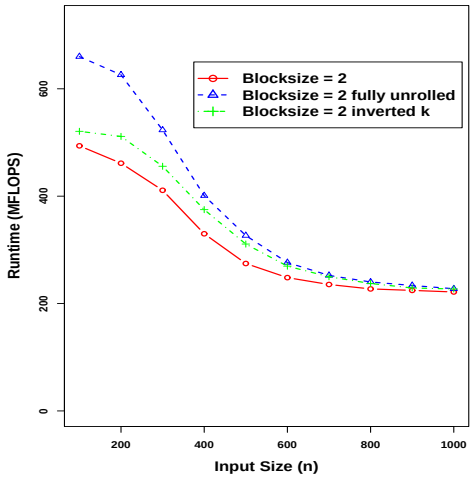

Fig. 3. LU blocked algorithm with block size of 2 plotted against several optimizations.

Figure 3, it appears that the modifications we made did not seem to have a large impact on the code's runtime. Both optimizations had slight impacts for lower values of n, but each leveled off to very similar results to the basic blocksize 2 code.

Blocksize 3. Next, we tried to speed up the bottleneck portion of the blocked algorithm with a blocksize of 3. Once

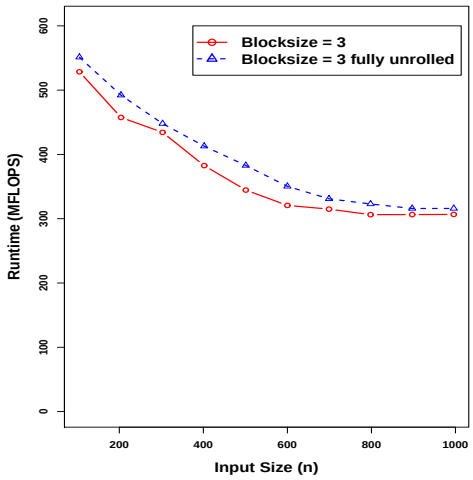

Fig. 4. LU blocked algorithm with block size of 3 plotted against a loop unrolled implementation.

again, as shown in Figure 4, the modified code did not have a large impact on our results. we attributed this failure to the amount of scalar variables needed to unroll the code. The number of scalar variables most likely ran past the number of registers, and were being pushed onto the stack.

Blocksize 4. Next, we tried to speed up the bottleneck portion of the blocked algorithm with a blocksize of 4. To do this, we unrolled the matrix matrix multiplication, and blocked this code into blocks of 2. In addition, we utilized the vector code library to improve the runtime performance. To utilize this library, we needed to run these tests with the Intel C++ compiler, icpc, with the command line of "icpc -O2 *.c". The unrolled code showed some improvement

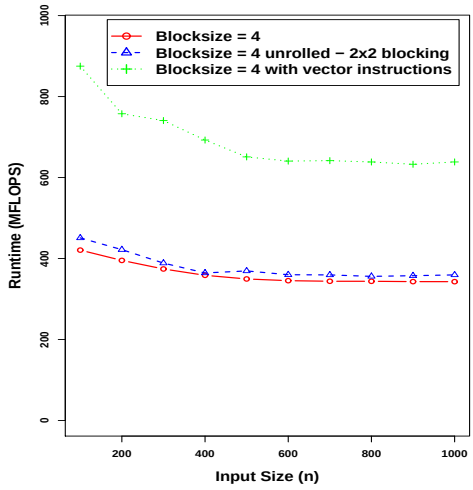

Fig. 5.

from Figure 5, but very little. However, the vector code showed outstanding improvement. This version of our code showed approximately a 7 times improvement.

5. CONCLUSIONS

From the graphs, it seems obvious that the use of the blocked algorithm provides significant improvement from the baseline algorithm. However, our improvements on these blocked versions did not seem to show much impact, apart from the use of the vector library. These vector instructions gave a great improvement on our code. As a future experiment, these vector instructions could be applied to the rest of the blocked algorithm. In addition, larger blocksizes could be tested to see which blocksize gives optimal improvement.

6. REFERENCES

[1] J. Demmel, *Applied Numberical Linear Algebra*, Siam, 1997.