LU Decomposition:
Parallel Algorithms for GPUs & Performance Studies

Scott Shellenhamer

December 16, 2012

Abstract

The project goal is to factor a matrix into an LU Decomposition using parallel techniques on a graphics-processing unit (GPU). Two techniques are used for the factorization and their performance results are compared with the corresponding sequential versions. These techniques are: LU decomposition without pivoting using a loop unrolling technique; LU decomposition with partial pivoting using a block algorithm.

1 Introduction

The project aims to factor dense single-precision square matrices in the size of 5,000 to 20,000 on a GPU. The GPU available memory will give an upper bound to the size of the matrix that can be used. For small matrices, CPU techniques that utilize the cache close to the processor may actually be more efficient. This is due to the overhead incurred from data transfer to the GPU and the smaller number of calculations that are actually performed.

The LU factorization problem requires approximately $2n^3 / 3$ floating point operations. Therefore, the best metric to analyze relative performance is in terms of GFlops/s achieved for each of the algorithms. I will also address memory bandwidth on the GPU, which is measured in terms of GB/s.

I used an iterative technique to work first with algorithms without pivoting and then unblocked algorithms with partial pivoting and finally blocked algorithms with partial pivoting. Each parallel algorithm is compared with the performance of its corresponding sequential version.

In researching algorithms, I found the University of Texas FLAME group to have the most applicable blocked algorithms for partial pivoting [1, 2]. For LU without pivoting, I referenced the Gamma Group at University of North Carolina and students at Osaka University [3, 4]. I used psuedocode from these papers to help me in implementation. I also made use of the Python Numpy module and the Python Scipy module that contains linear algebra routines built on top of the ATLAS/LAPACK Fortran routines. I used the **lu** function from the **linalg** module extensively throughout the blocked partial pivoting algorithm. The numpy "**where**" function was used to convert a matrix permutation into a vector.

2 LU Decomposition without Pivoting

2.1 Sequential approaches

A basic form of LU Decomposition without row pivoting can be run on certain well-behaved matrices.

$$\mathbf{A = LU}$$

This technique cannot be run on all matrices; however, it does significantly simplify the algorithm when appropriate. A number of algorithms have been developed for this method. In this project, I have implemented three such algorithms in order to determine which would adapt best to parallelism on a GPU. Detailed performance results for each of the sequential algorithms are presented in Section 4. The Doolittle and Crout algorithms each had the best results and achieved a max GFlops/s of 1.7. The main difference in the algorithms is from the for loop ordering and the dependencies between the steps. Although the right-looking (eager) algorithm was the worst performing sequentially, it does give the best opportunities for parallelism and use of cache.

```
for (k = 0; k < N; k++)
        for (i = k + 1; i < N; ++i)
                A_ik = A_ik / A_kk

        for (i = k + 1; i < N; ++i)
                for (j = k + 1; j < N; ++j)
                        A_ij -= A_ik * A_kj
```

Figure 1. Right Looking Algorithm

2.2 Parallel Algorithm

The right-looking algorithm can be parallelized to some extent, however, not all dependencies can be eliminated. Each pass through the loop **k** is dependent on the previous pass. Also, the first inner **i** loop (column normalization) must occur before the second inner **i** loop (trailing sub-matrix update). This leaves two opportunities for parallelism: a) Each iteration of the column normalization can be independently processed on the GPU; b) The nested loop in the trailing sub-matrix update can be unrolled and processed on the GPU in a grid-like fashion. I leave the outer **k** loop on the CPU because of the iteration dependencies and spawn two kernels for each of the inner loops. The unblocked algorithm has the limitation that it will always spend k iterations on the GPU. In addition, the matrix must be padded with zeros if the dimensions to not fit with our group size dimensions (16x16 for sub-matrix update). The algorithm does, however, have the advantage of being able to do all computations on the GPU. The matrix can be transferred once to and from the GPU minimizing penalties from the transfer overhead. My expectations were that the performance would improve significantly from the sequential version in terms of time elapsed and GFlops/s. The

parallel algorithm will benefit mostly from the second inner for loop running on the GPU. Using a grid-like structure on the GPU of 16x16, we can run more computations in parallel. However, the global memory access pattern is not ideal because of the unblocked approach to the algorithm. The amount of data we can reuse in local memory is limited, and therefore I do not expect the performance gains seen in a blocked matrix-matrix multiplication.

3 LU Decomposition with Partial Pivoting

3.1 Unblocked sequential approaches

To overcome the limitations of matrix decomposition without pivoting, a partial pivoting technique is employed.

$$\mathbf{PA = LU}$$

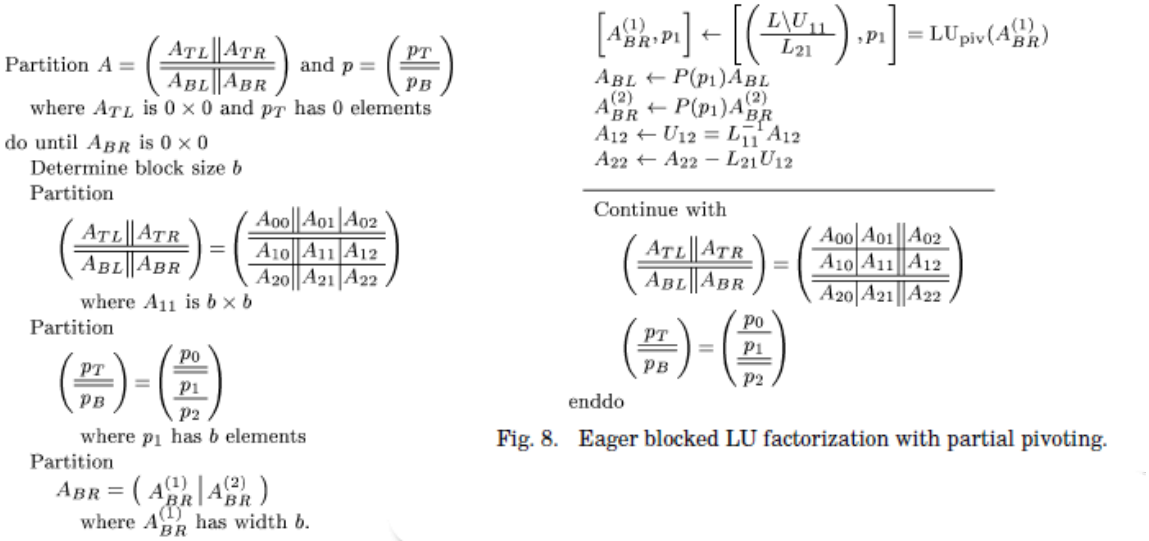
An additional permutation matrix will reorder the rows to make a viable factorization. Partial pivoting adds a level of complexity to the algorithm. While iterating through the k columns, we must find the maximum element in the column and swap this row with the k^{th} row. For unblocked sequential algorithms, this step makes the algorithm run significantly slower. I wrote an unblocked algorithm with partial pivoting in C. This algorithm has roughly the same performance as the right-looking sequential algorithm without pivoting achieving a max GFlops/s of 1.5 for a matrix size of 64x64.

3.2 Blocked sequential approaches

Like other matrix algorithms we have seen in class, the LU factorization can benefit from running a blocked version of the algorithm. Linear algebra libraries written in C and Fortran like LAPACK and ATLAS have implemented blocking to improve performance. On sequential systems, the optimal blocking scheme can take advantage of cache memory close to the processor. For baseline testing, I used the Python Scipy module which has linear algebra routines built on top of ATLAS, LAPACK, and BLAS. The full timings can be found in section 4. The blocked algorithm with partial pivoting consistently outperforms any of the naïve algorithms with no pivoting, as well as the unblocked sequential algorithms. The results are limited to matrix sizes up to 5120x5120 because of machine memory limits, but tops out around 3.3-3.4 GFlops/s.

3.3 Parallel blocked algorithms

The question is, can the blocked algorithms be improved by exploiting parallelism on the GPU? If so, then LU decomposition could see performance gains similar to matrix multiplication on a GPU. Using the FLAME [1,2] algorithm as a model, the blocked algorithm in FLAME's notation is as follows:



Partition $A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$ and $p = \left(\begin{array}{c} p_T \\ \hline\hline p_B \end{array}\right)$
where A_{TL} is 0×0 and p_T has 0 elements
do until A_{BR} is 0×0
Determine block size b
Partition
$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$
where A_{11} is $b \times b$
Partition
$\left(\begin{array}{c} p_T \\ \hline\hline p_B \end{array}\right) = \left(\begin{array}{c} p_0 \\ \hline\hline p_1 \\ \hline\hline p_2 \end{array}\right)$
where p_1 has b elements
Partition
$A_{BR} = \left(\begin{array}{c|c} A_{BR}^{(1)} & A_{BR}^{(2)} \end{array}\right)$
where $A_{BR}^{(1)}$ has width b.

$\left[A_{BR}^{(1)}, p_1\right] \leftarrow \left[\left(\begin{array}{c} L\backslash U_{11} \\ \hline\hline L_{21} \end{array}\right), p_1\right] = \mathrm{LU}_{\mathrm{piv}}(A_{BR}^{(1)})$
$A_{BL} \leftarrow P(p_1)A_{BL}$
$A_{BR}^{(2)} \leftarrow P(p_1)A_{BR}^{(2)}$
$A_{12} \leftarrow U_{12} = L_{11}^{-1}A_{12}$
$A_{22} \leftarrow A_{22} - L_{21}U_{12}$

Continue with
$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$
$\left(\begin{array}{c} p_T \\ \hline\hline p_B \end{array}\right) = \left(\begin{array}{c} p_0 \\ \hline\hline p_1 \\ \hline\hline p_2 \end{array}\right)$
enddo

Fig. 8. Eager blocked LU factorization with partial pivoting.

Figure 2. FLAME Eager blocked LU Factorization algorithm with partial pivoting.

The same algorithm is described below in psuedocode.

```
for k in block count:
        Run LU decomp on block column Ak
        Convert permutation matrix to permutation vector
        Apply permutations to all rows except block k

        #kernel rowsolve
        for j =k+1 in block_cols:
                Trisolve for U in block row k using L from LU above

        #kernel submatrix_update
        for i = k+1 in block_rows:
                for j = k+1 in block_cols:
                        Aij = Aij – Lik * Ukj
```

Figure 3. Pseudocode for block LU with partial pivoting

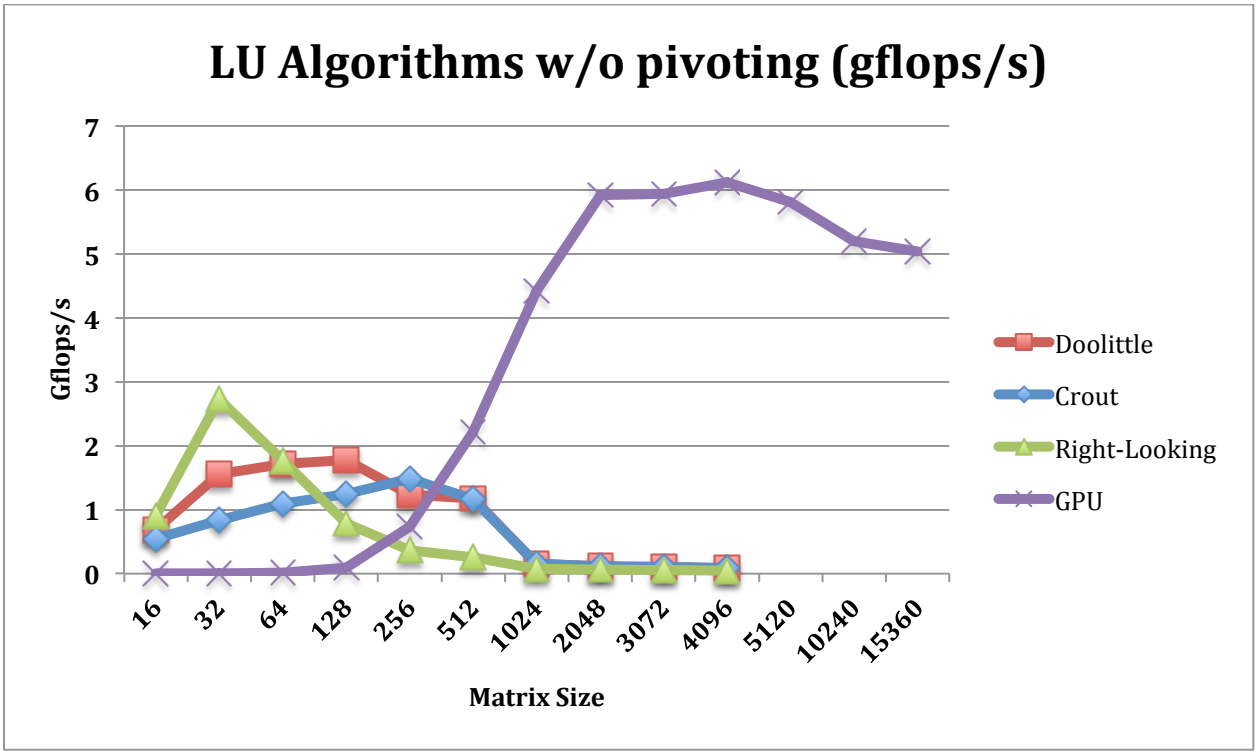
To begin, I chose to use a hybrid algorithm between the CPU and GPU to reuse functions available in Scipy to perform the LU Decomposition on each k block and obtain the permutation matrix. The permutation is then converted to a permutation vector on the CPU using the Numpy where method. This information is then passed to the GPU to update the rest of the matrix using three separate kernels: row swap, row solve, and submatrix update. With this approach, I believed the main bottleneck would come from memory bandwidth. On each block iteration through the matrix, my initial approach would transfer the whole matrix back and forth twice between the CPU and GPU. This is extremely expensive when using large matrices. The hardest step to parallelize is the row swapping. Because the permutation matrix implies each row swap is done independently, the swaps cannot be done in place. A second matrix buffer must contain the permuted matrix and then we copy this back to the original buffer to proceed. I could have proceeded through the algorithm using the second buffer, however, this would present tricky synchronization problems in keeping both matrices the same throughout the loops. I expected the performance for the row solves and the sub-matrix updates to be very fast. Because they are essentially matrix-multiplications, these kernels should have performance in the hundreds of Gflops/s. In particular, the row solve should see the benefit of using local workgroup memory for the single L matrix argument. In addition, the row solve has a coalesced global memory access pattern that is optimal for minimizing misaligned accesses. In the sub-matrix update kernel, the global memory access pattern is less optimal. The work group must hit a U matrix, L matrix, and an A matrix in three different portions of memory.

4 Performance Results

All sequential algorithms were run on a Macbook Pro with a 2.8GHz Intel Core i7 processor and 8 GB of 1333 MHz DDR3 RAM. All parallel algorithms were run on the NYU Cuda cluster using the NVIDIA GPUs.

4.1 LU Decomposition without pivoting

The performance comparisons of the LU decompositions without pivoting are presented in Graph 1 below. This experiment assumes a well-behaved matrix. The sequential algorithms outperform the GPU algorithm for matrices less than 512x512 with a peak performance averaging between 1 and 2 Gflops/s. The parallel algorithm on the GPU steadily increases performance as the size of the matrix increases and then trails off around 5120x5120. The peak performance occurs with a matrix size of 4096x4096 with around 6 Gflops/s. The GPU algorithm has an 85x performance increase over the Doolittle algorithm for a 4096x4096 size matrix. The sequential algorithms most likely peak when the matrix size corresponds to available cache size.



Graph 1. LU Algorithms without pivoting performance

	Elapsed			Gflops/s		
Matrix Size	Total	ColNorm	Submatrix	Total	ColNorm	Submatrix
16	0.02	0.01	0.01	0.000	0.001	1.394
32	0.02	0.01	0.01	0.001	0.001	1.393
64	0.02	0.01	0.01	0.012	0.001	1.393
128	0.02	0.01	0.01	0.092	0.001	1.393
256	0.02	0.01	0.01	0.736	0.001	1.391
512	0.04	0.01	0.03	2.216	0.002	3.467
1024	0.16	0.03	0.13	4.419	0.004	5.446
2048	0.97	0.05	0.90	5.915	0.008	6.350
3072	3.26	0.08	3.14	5.935	0.058	6.166
4096	7.48	0.11	7.30	6.124	0.078	6.274
5120	15.40	0.13	15.16	5.812	0.098	5.905
10240	137.70	0.28	137.02	5.198	0.190	5.225
15360	479.48	0.43	478.15	5.039	0.275	5.053

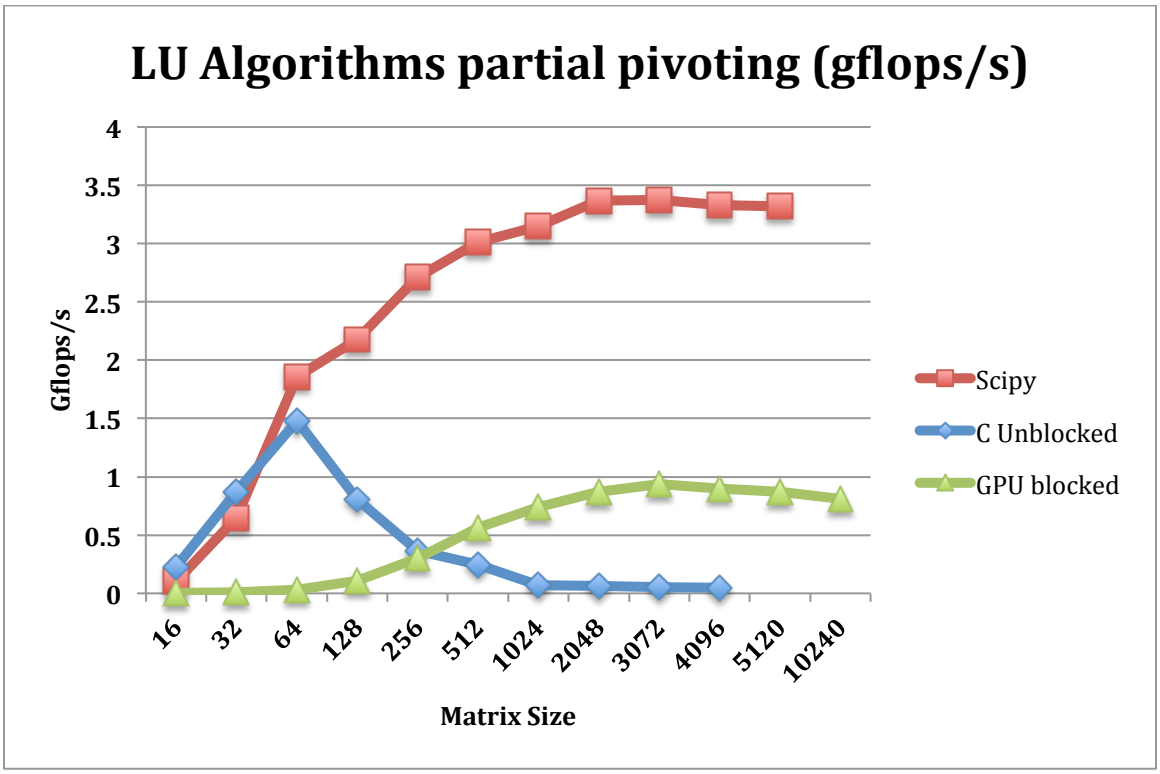
Chart 1. LU no pivoting GPU Algorithm breakdown

The breakdown of the kernels of the GPU algorithm is presented in Chart 1 above. The bulk of the time spent is in the trailing sub-matrix update kernel and the column normalization kernel time is negligible. The trailing sub-matrix kernel achieves a reasonable number of Gflops/s, however, nowhere near a matrix-multiplication. This was somewhat expected. Because this kernel is the bulk of the computation, any improvements to the kernel will significantly improve the performance of the whole algorithm. I believe there is room for improvement in the global memory access pattern. The memory transfer, which occurs at the beginning and end of the loop, was negligible when compared to the rest of the algorithm so this is not presented in the chart.

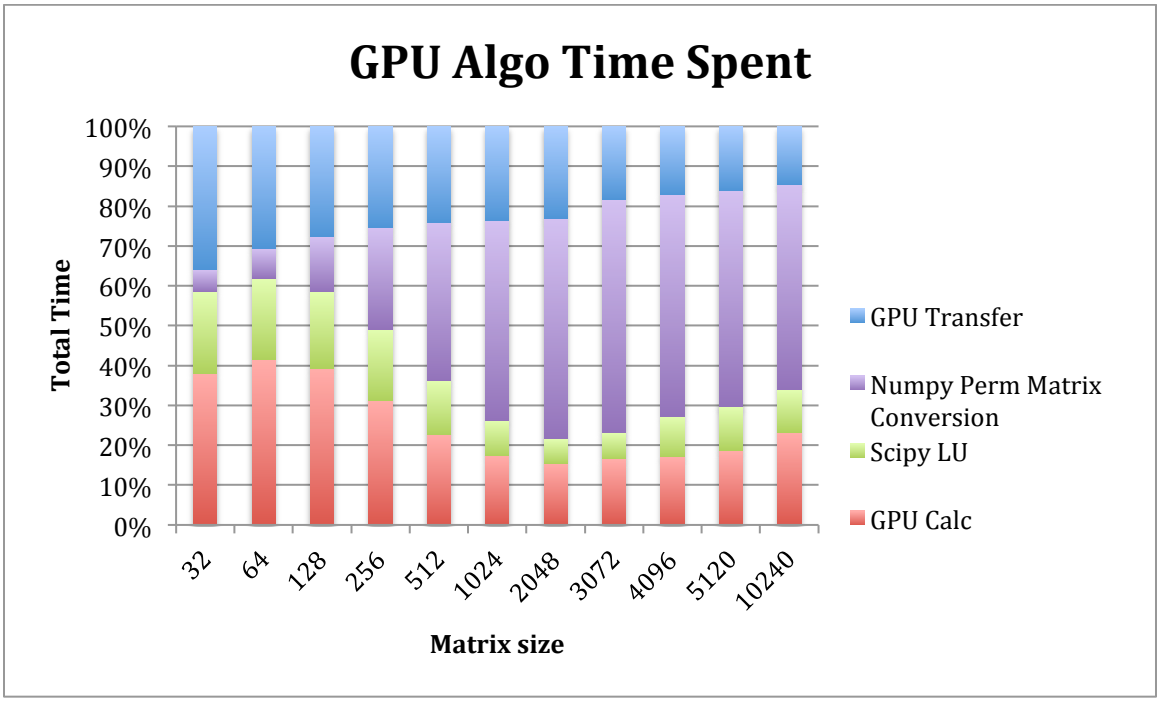
4.2 LU Decomposition with partial pivoting

The performance comparisons of the LU decompositions with partial pivoting are presented in Graph 2 below. The best performance comes from the Scipy sequential blocked algorithm using the ATLAS/LAPACK libraries. This algorithm achieves a peak performance around 3.4 Gflops/s. The GPU algorithm consistently achieves a performance just under 1.0 Gflops/s for larger matrices; however, these numbers are disappointing compared to the Scipy algorithm. When the timings are broken down into the various components of the algorithm we can see why.

Looking at Graph 3, the major parts of the algorithm are broken down as a percentage of time taken of the total. The Scipy LU function stays at a constant 10-20% for the bulk of the timings. The GPU calculation time is higher for smaller matrices (40%) than for larger matrices (<20%). The transfer time actually becomes smaller and smaller as the matrices get larger. The major problem, however, is that the time taken for the Numpy permutation matrix conversion is way to large to get the overall numbers where they should be. I anticipated that row swapping would be an issue, but did not anticipate actual permutation matrix to vector conversion would be as big of an issue. I did not emphasize this in my implementation. This CPU based algorithm (numpy.where function) should be replaced with a GPU based kernel. The other alternative would be to use the permutation matrix as is and perform a matrix-matrix multiplication on the GPU to transform the column blocks into the proper alignment. I believe either approach would have improved performance.



Graph 2. LU Algorithms with partial pivoting performance



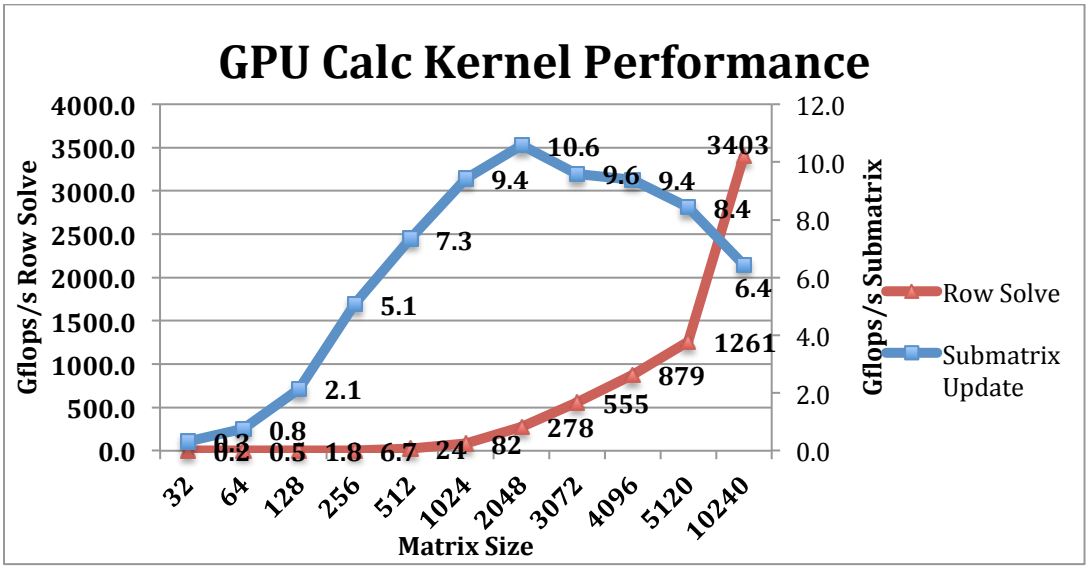
Graph 3. LU GPU Algorithm time spent as a percentage of total

In my initial implementation, I found that memory bandwidth was the main bottleneck. I was transferring the matrix back and forth between CPU and GPU too many times throughout the loop. In the end, I decided to reduce this to one transfer of the matrix buffer at the beginning of the code and one back to the CPU at the end of each loop iteration. To reduce the number of transfers, I implemented a trivial "copy" GPU kernel, to copy the results from the Scipy CPU LU function into matrix buffer on the GPU. I found it better to call a kernel that did not do any computations then to transfer the whole matrix again. I could not, however, remove the transfer back to the CPU at the end of each iteration. If this "pull" transfer was removed, the algorithm could perform even better. In a future implementation, if I still used a CPU/GPU hybrid approach, I would fix this as well by updating a smaller buffer with only the results needed for the next iteration.

GPU Partial Pivoting Algorithm Memory Transfer				
Matrix	**GPU Push**	**Push gb/s**	**GPU Pull**	**Pull gb/s**
32	0.000618	0.021731	0.000436	0.018786
64	0.000828	0.057229	0.000793	0.08262
128	0.001317	0.134121	0.001853	0.282942
256	0.002372	0.287137	0.005974	0.702115
512	0.005373	0.497398	0.029906	1.122013
1024	0.016633	0.636565	0.199878	1.342993
2048	0.054823	0.768803	1.403017	1.530619
3072	0.107381	0.881708	3.572831	2.028575
4096	0.179167	0.938687	8.304903	2.068642
5120	0.267669	0.981273	16.043493	2.091467
10240	1.069124	0.981739	126.570129	2.120844

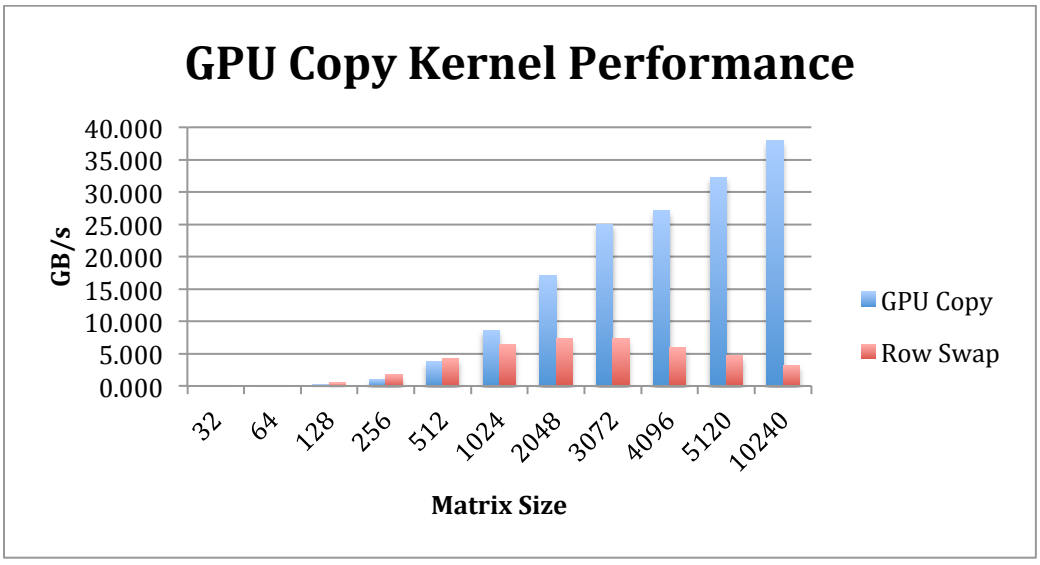
Chart 2. GPU partial pivoting algo memory transfer

The performance numbers for the GPU Kernels are presented below in terms of Gflops/s. As anticipated, the Row Solve kernel had outstanding performance that increased with matrix size. However, the performance beat my expectations. This kernel overall had a very negligible affect on the over all time of the algorithm. The Submatrix Update kernel, which performs the bulk of the computations, peaked at 10.6 Gflops/s. I believe this was because of the global memory access pattern from the kernel. Because the kernel was using data from three different regions of memory in the matrix, I was not able to come up with the optimal strategy for global memory access.



Graph 4. GPU Calculation Kernel Performance for partial pivoting algorithm

Finally, the results are displayed below for the two GPU copy kernels. As anticipated, the row swap kernel was somewhat of a bottleneck for larger matrices. The row swap has basically a fixed cost that can only be optimized to a certain extent. I would have expected the bandwidth numbers to be higher than they were. The small GPU copy kernel copied data from the CPU LU decomposition into the buffered matrix on the GPU. This kernel obtained much better performance in terms of GB/s.



Graph 5. GPU Copy Kernel Performance for partial pivoting algorithm

4.3 Comparison with existing work

For partial pivoting algorithms, the FLAME group was able to achieve performance approaching 60 GFlops/s for matrices of size 10,000 using 16 processors in a multithreaded approach in [2]. The group in [5], also associated with FLAME, achieved peak performance of 47 GFlops/s on a GPU.

For algorithms without pivoting, the Gamma group achieved peak performance of ~4 GFlops/s in [3]. In comparing with these groups, I was able to achieve slightly better performance for the algorithm without pivoting and worse performance for the algorithm with partial pivoting.

5 Code Availability

My code is available in my final project git repository <hpc12-fp-ss7064>. Instructions are listed below for building/running the code for the various sections.

Sequential LU Decomposition without Pivoting:

- Compile the **lu_c.c** file using the provided Makefile. This depends on the timing.h file (located in the project folder). The build will output a binary named **luc**. From here you can run the Gauss Elimination (1), Doolittle (2), Crout (3), and Right-Looking Algorithms (4).
- The arguments for the binary are <matrix size> <number of trips> <algorithm (1,2,3,4 above)> and [output 0=1]. The default is 0 to not output the matrix.
- The algorithm will only work for well-behaved matrices. The output can be verified against Scipy's lu function.

Parallel LU Decomposition without Pivoting for GPU:

- Compile the **lu_gpu.c** file using the provided Makefile. This depends on the timing.h file, lu_functions.h file, and cl_helper.h/c files (located in the project folder). It also uses the kernel files lu_normcol.cl and lu_submatrix.cl. The build will output a binary named **lugpu**.
- The arguments for the binary are <matrix size> <number of trips> and [output 0=1]. The default is 0 to not output the matrix.
- The algorithm will only work for well-behaved matrices. The output can be verified against Scipy's lu function.

Sequential LU Decomposition with Partial Pivoting – Unblocked:

- Using the **luc** binary produced from the first lu_c.c file, run the binary with an algorithm number **10**.
- The arguments for the binary are <matrix size> <number of trips> and [output 0=1]. The default is 0 to not output the matrix.

- The output can be verified against Scipy's lu function.

Sequential LU Decomposition with Partial Pivoting – Blocked using Scipy:

- Run the lu.py python file on a machine with Numpy and Scipy installed.
- The arguments for the program are <matrix size> <number of trips>.

Parallel LU Decomposition with Partial Pivoting for GPU in Python:

- Run the lu_gpu.py python file on a machine with the latest version of PyOpenCl (for ImmediateAllocator), Numpy, and Scipy installed.
- The arguments for the program are <matrix size> <block size> [validate=0] [output=0].
- When setting the validate flag to 1, the program will compare the results to the Scipy lu function using Numpy's "allclose" array method.
- The output flag will print the original and factored matrices with A overwritten. The function I created in the code will also return a dictionary of permutation vectors.
- The program should be run with matrix sizes divisible by your block size. The optimal block size was 16 in my testing.

6 Conclusions

LU Decomposition with partial pivoting is a difficult problem to solve on a GPU. Dependencies throughout the algorithm and permutations prove hard to parallelize. I believe the algorithm I have come up with is a good start towards high performing future algorithms. Future changes would include implementing the entire algorithm on the GPU and removing dependencies on Scipy. There are even potential areas to utilize the CPU and GPU to do calculations at the same time on blocks without dependencies. From my experience, for smaller matrices, block algorithms on the CPU prove to be the most efficient. For larger matrices, the intial results for GPU algorithms looks promising. With a few improvements, this could prove to be a solid approach.

References

[1] Gunnels, J., Gustavson, F., Henry, G., and Van de Geijn, R. 2001. FLAME: Formal Linear Algebra Methods Environment. ACM Transactions on Mathematical Software, Vol. 27, No. 4, December 2001. p422-455.

[2] Quintana-Orti, G., Quintana-Orti, E., Chan, E., Van de Geijn, R., Van Zee, F. 2007. Design and Scheduling of an Algorithm-by-Blocks for the LU Factorization on Multithreaded Architechtures. FLAME Working Note #26. September 19, 2007.

[3] Galoppo, N., Govindaraju, N., Henson, M., Manocha, D. 2005. .LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware., GAMMA research group, University of North Carolina, Chapel Hill. Proceedings of the ACM/IEEE SC/05 Conference. November 12-18, 2005.

[4] Ino, F., Matsui, M., Goda, K., Hagihara, K. 2005. Performance Study of LU Decomposition on the Programmable GPU. Graduate School of Information Science and Technology, Osaka University.

[5] Barrachina, S., Castillo, M., Igual, F., Mayo, R., Quintana-Orti, E. 2008. Solving Dense Linear Systems on Graphics Processors. Euro-Par 2008, LNCS 5168 pp 739-748, 2008.