Parallel LU Decomposition on GP-GPU using CUDA

Roll - 20162150

ZIAUL CHOUDHURY, IIIT Hyderbad

In this work, we implement a parallel LU decomposition on the GPU using CUDA. Two types of implementations are presented, namely, iterative and recursive. A study of the scalability and throughput of both the approached are carried out. The experiments show, that a mix of both the approaches outperform each of them individually by ≈ 1.5 to 1.2 times respectively.

ACM Reference Format:
Ziaul Choudhury. 2019. Parallel LU Decomposition on GP-GPU using CUDA: Roll - 20162150. *Proc. ACM Meas. Anal. Comput. Syst.* 37, 4, Article 1 (August 2019), **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Suppose we have the system of equations

$$AX = B \tag{1}$$

The motivation for an LU decomposition is based on the observation that systems of equations involving triangular coefficient matrices are easier to deal with. Indeed, the whole point of Gaussian elimination is to replace the coefficient matrix with one that is triangular. The LU decomposition is another approach designed to exploit triangular systems. We suppose that we can write

$$A = LU \tag{2}$$

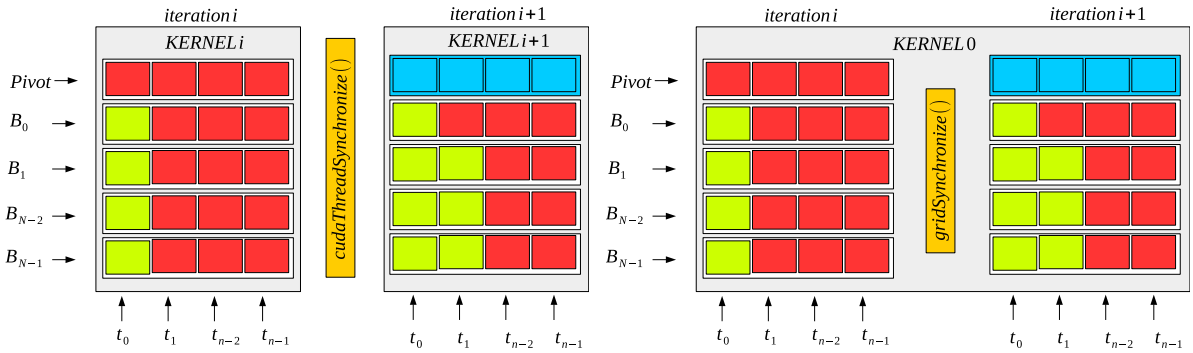
where L is a lower triangular matrix and U is an upper triangular matrix. Our aim is to find L and U and once we have done so we have found an LU decomposition of A.

1.1 Algorithm

The LU decomposition of a nxn matrix A is done iteratively by picking pivot rows 0 through $n-1$. Each iteration i picks up a pivot row i in A and creates 0 in the positions $A[i+1, i]$ through $A[n-1, i]$. As each iteration creates a new version of the matrix, they are dependent on each other and has to progress sequentially. The parallelism in the algorithm lies in the row operations being performed using the pivot row. In this project we implement two parallel version of the LU decomposition on the GPU, the first one is an iterative version, with two variants and the latter one is a recursive solution based on the blocked LU decomposition algorithm. These two approaches are discussed in the followup sections.

Author's address: Ziaul Choudhury, ziaul.c@research.iiit.ac.in, IIIT Hyderbad.

2476-1249/2019/8-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Iterative Multikernel (IMK) Approach. (b) Iterative Single Kernel (ISR) Approach.

Fig. 1. Iterative Approach for A=LU

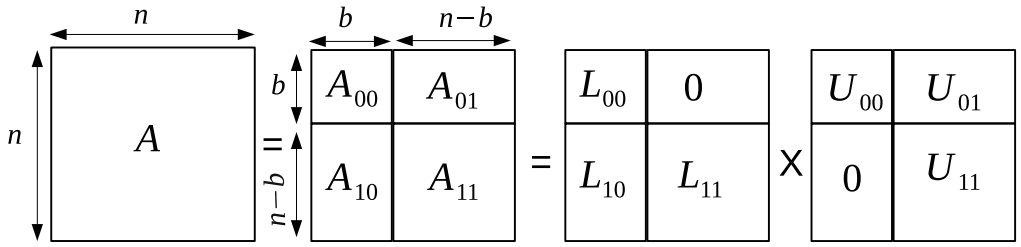


Fig. 2. Block LU Decomposition of a matrix A.

2 ITERATIVE APPROACH

In this approach, the iterations are mapped to the GPU. Each iteration is completely decomposed using a 2D mapping of threads and blocks. Each iteration has a maximum of n steps, in each step, a single row is subtracted from the pivot row. Block i executes the i^{th} step and thread j within a block subtracts the j^{th} element of the pivot row with the corresponding element of the current row.

There exists two variations in iterative approach depending on how the iterations are handled. In the multi kernel approach each sequential iteration is executed by a separate kernel. The kernel executing iteration i is synchronized by the CPU with the one executing iteration $i + 1$. In contrast, to this there is the single kernel variant, where a single kernel executes all the iterations. Iterations are synchronized, using the inter-block (grid synchronize) primitive supported by CUDA 9.0 and later.

3 RECURSIVE APPROACH

The recursive approach is based on the blocked LU decomposition technique, where the LU decomposition is carried out at a block by block basis. More specifically, the $n \times n$ matrix is first divided as shown in the figure **??**. The $b \times b$ block is decomposed in a lower and upper triangular matrix L_{00} and U_{00} respectively, using any of the methods that were described earlier.

The matrix $b \times (N-b)$ matrix U_{01} is found out using a lower triangular solver. Similarly, the $(N-b) \times b$ matrix L_{10} is found out using a upper triangular solver. Finally, the left over $n-b \times n-b$ matrix is chosen as the new matrix, which is to be recursively solved using the steps described above.

This approach progressed in two steps, in the first step, the basic LU decomposition is used to get the matrices L_{00} and U_{00}. The upper and the lower triangular solver kernels are launched in parallel. Within a kernel, a maximum of $n-b$ threads solve each column of the respective matrix. This technique, greatly solves the problem of global synchronization that appears in the iterative approach.

$$A_{00} = L_{00} \times U_{00} \tag{3}$$
$$A_{01} = L_{00} \times U_{01} \tag{4}$$
$$A_{10} = L_{10} \times U_{00} \tag{5}$$
$$A_{11} = L_{10} \times U_{01} + L_{11} \times U_{11} \tag{6}$$

4 HYBRID APPROACH

This approach is a mix of the previous two approaches. We set a parameter r, that controls the level of recursion. We start with blocked version and on reaching the r^{th} recursive step, the resultant matrix is factorized using the iterative approach. Setting r to the maximum value, completely eliminates the use of the iterative approach. On the other hand using the minimum r value, converts the entire processing to the iterative approach.

5 EXPERIMENTS

In this section, we experiment with all the approaches described above using various matrix sizes. The following command was used to compile all the programs.

$$\texttt{nvcc} - \texttt{arch} = \texttt{sm}_{\texttt{6}}\texttt{0LU2.cu} - \texttt{rdc} = \texttt{true} - -\texttt{std} = \texttt{c} + +\texttt{11} - \texttt{otest}$$

In the first experiment, we vary the number of blocks in the iterative approach, to test the scalability of our implementation. As can be seen in figure **??**, with more number of blocks, the execution time decreases almost linearly. The SKR approach achieves a lower run time. A possible reason for this is the absence of CPU intervention, as the entire processing is carried out by a single kernel.

In the MKR approach, with more number of blocks the line flattens out with high block count. This is because with more number of blocks, the grid synchronize primitive used in the implementation starts to dominate the run-time. We next, experiment with varying matrix sizes with a block count of $\frac{N}{2}$, where N is the matrix size. As can be seen in the figure, the throughput scales up almost exponentially corresponding to the exploding matrix sizes. Finally, we look into the synchronization overhead in the SKR approach on a 1000x1000 matrix. As can be seen in figure as the number of blocks crosses 100, the synchronization overheard dominates and results in a higher processing time.

We next, experiment with the recursive and the recursive hybrid approach and do a similar scalability and throughput analysis. As can be seen in figure **??**, our implementation is scalable. The run-time of the hybrid approach is lower than the the recursive approach which is again lower that the iterative approaches. Similarly in the throughput front, both the recursive and the hybrid approaches, achieves exponential throughput scaling. We next vary the initial block size in the recursive hybrid approach. As can be seen in the figure, the best result is achieved with a block size of 32. Increase in block size results in domination of the iterative approach resulting in a lower run-time.

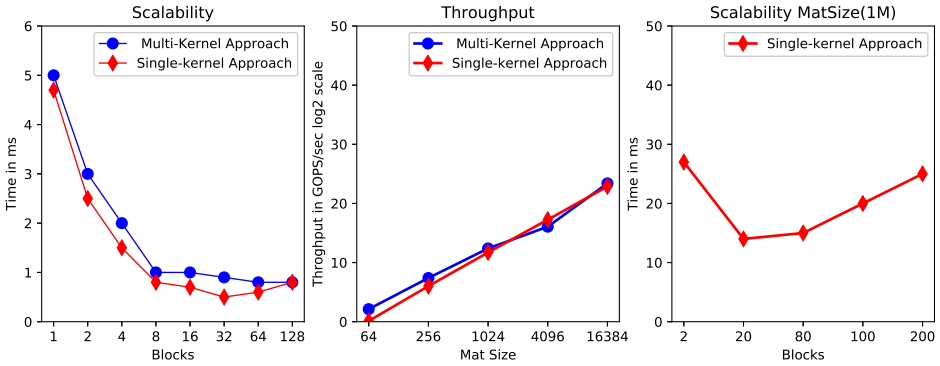


Fig. 3. Scalability and throughput analysis of the iterative approach.

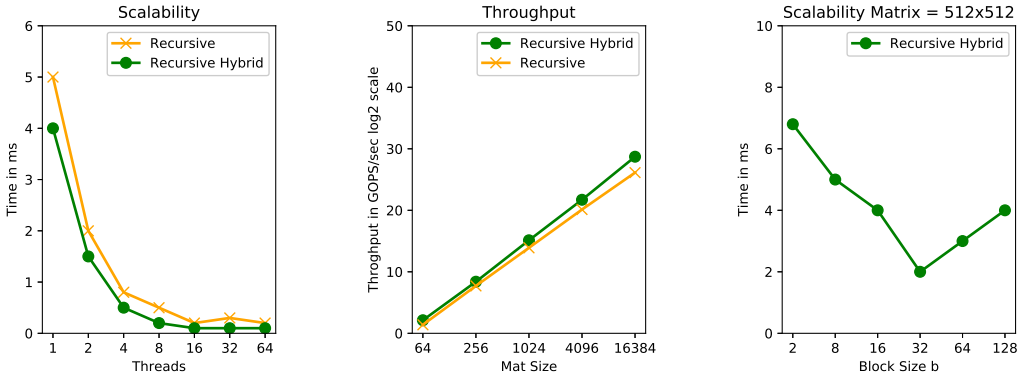


Fig. 4. Scalability and Throughput analysis of the recursive approach.

6 CONCLUSION

From this work we conclude that the recursive approach proves to scale and achieve higher throughput compared to the iterative approach. The order of efficiency is as follows.

`MKR < SKR < Recursive < Recursive Hybrid`