LU Decomposition: A Timing Study

Gopalan Iyengar 19D170009
Om Prabhu 19D170018
Sangraam Patwardhan 19D100017
Punit Madia 190100096

***Abstract*—The solution of systems of linear equations is the heart of many applications in scientific computing. However, given the increasing complexity of data, it is a computationally expensive process. We attempt to parallelize LU decomposition of a matrix, a technique often used to solve systems of linear equations. This report discusses a timing study on 3 different methods of LU decomposition parallelized using OpenMP. Experimental results on a multicore CPU show that a higher exponent for the single-thread power fit curve implies higher parallelizability for the algorithm. Finally, we analyze the results obtained from the timing study draw a comparison to some theoretical predictions.**

I. Introduction

Matrix decomposition is one of the key techniques used to solve systems of linear equations, which is a frequently encountered problem in the field of scientific computing. Depending on the case at hand, there are a variety of matrix decomposition methods available including but not limited to LU, QR and Cholesky decomposition. In this report, we will only restrict ourselves to LU decomposition.

LU (or lower-upper) decomposition on a non-singular matrix $A \in \mathbb{R}^{n\times n}$ involves factoring the matrix as the product of a lower and upper triangular matrix $L \in \mathbb{R}^{n\times n}$ and $U \in \mathbb{R}^{n\times n}$ respectively. This significantly reduces the complexity of solving a linear system $\mathbf{Ax} = \mathbf{b}$ by reducing an otherwise intimidating computation to the much simpler calculations $\mathbf{y} = \mathbf{Ux}$ and $\mathbf{b} = \mathbf{Ly}$. Given that both the matrices $\mathbf{L}$ and $\mathbf{U}$ are triangular, it is then merely a problem of simple substitution.

A more specific case of LU factorization, also known as the "right-looking" LU algorithm, deals with factoring the matrix such that either $\mathbf{L}$ or $\mathbf{U}$ are unit diagonal matrices. Two of the methods discussed herewith are sub-cases of the right-looking LU algorithm.

II. The General Procedure:

With LU factorization, the simplest approach would be to loop either over rows or over columns. We use 2 nested `for` loops to dynamically update $\mathbf{L}$ and $\mathbf{U}$ with every iteration.

A major problem with any type of matrix decomposition is the high number of computations. As a general rule of thumb, we prefer to minimize the number of loops to avoid making our code computationally expensive. In this regard, the most basic algorithm we discuss is Gaussian Elimination. The maximum number of nested `for` loops is 3. A pseudo-code outlining the Gaussian Elimination procedure is discussed alongside:

```
/*set number of threads*/
/*initialize random array of size NxN*/
for k=0 to N-1 /*loop over columns*/
  for i = k+1 to N-1
    /*update L, nested for loop
    to dynamically compute
    the original matrix to U*/
    by elementary row operations*/
  end
end
```

Thus, it is seen that the number of `for` loops is 3. Similarly, the Doolittle algorithm and Crout's method have the same number of loops, i.e. the same order of operations.

III. Techniques of Matrix Decomposition

We look at 3 different methods of LU decomposition. All the methods have a theoretical time complexity (or order of operations in Big-O notation) of $O(n^3)$, meaning that for a matrix of order n, the number of arithmetic operations scales with n^3.

- **Gaussian Elimination algorithm**: Involves elementary row operations on matrices to reduce it to an upper triangular "row echelon form".

$$\mathbf{A} = \mathbf{LU} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

- **Doolittle algorithm**: factorization into unit diagonal lower triangular matrix and upper triangular matrix.

$$\mathbf{A} = \mathbf{LU} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

- **Crout's method**: factorization into lower triangular matrix and unit diagonal upper triangular matrix.

$$\mathbf{A} = \mathbf{LU} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

The timing study performed for these algorithms is discussed in the following section.

IV. Timing Study

We parallelized the given 3 algorithms using OpenMP and carried out a timing study on each of the codes. The specifics are listed below:

- Order of the matrix N was varied from 100 to 2000 in steps of 100 (possible to go beyond 2000, but time explodes)
- Number of threads was varied between 1, 2, 4 and 8 for each value of N

Initially, the code encountered a segmentation fault at about N=800, which was corrected by dynamic declaration of variables. The graphs of execution time vs. matrix order generated using the codes are shown below:

The entire timing study was performed on a system running Ubuntu 20.04 (alongside a native Windows 10 operating system) on an Nvidia GTX 1650M GPU and a quad-core Intel i5-9300H CPU.

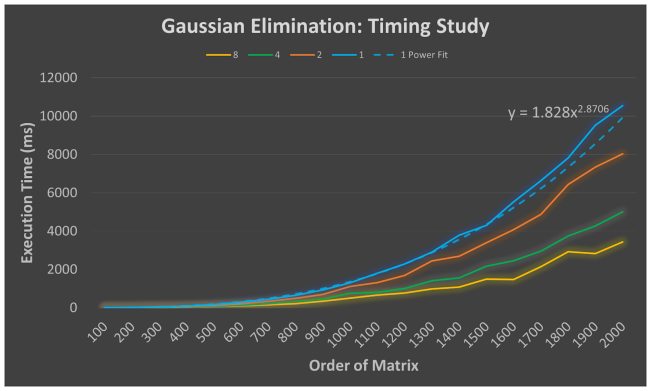

Fig. 1. LU decomposition using Gaussian Elimination

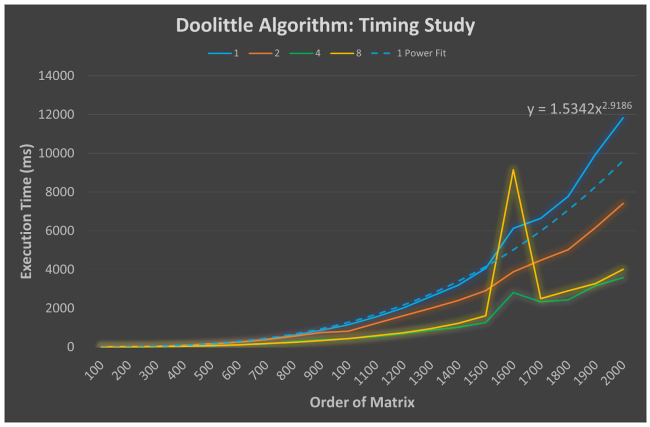

Fig. 2. LU decomposition using Doolittle Algorithm

V. Results: OpenMP

- In the preceding **Execution time (ms) vs. Order of Matrix N** graphs, the trends are color coded as follows:
  - **Blue**: 1 Thread
  - **Blue (Dashed)**: 1 Thread - Power Fit
  - **Orange**: 2 Threads
  - **Green**: 4 Threads
  - **Yellow**: 8 Threads
- The execution time for a given order of matrix N, decreases with increase in number of threads, up till a certain number of threads. eg. 4 threads perform better for Doolittle's algorithm than 8 threads. Generally, for a c-core CPU, performance increases until c threads, then decreases.
- The execution time for a given number of threads, increases proportionally with the $x-th$ power of the order of matrix n, i.e. $t \propto n^x$, such that $1 < x < 3$. This is backed by the fact that the theoretical complexity of the algorithms is $O(n^3)$.
  - **Gaussian Elimination**: x = 2.87
  - **Doolittle Algorithm**: x = 2.91
  - **Crout's Method**: x = 2.96
- It is seen that the Gaussian elimination implementation takes the least time (10.5s, 12s, 15s for Gaussian Elimination, Doolittle Algorithm, and Crout's Method resp. for N = 2000) for 1 thread (serial implementation), at a glance from the power approximations of the curves.
- These power fit approximations also provide an upper bound to the time curves for higher number of threads.
- Also, it is observed that the Doolittle algorithm and Gaussian elimination have higher parallelization possibility than the Crout's method, since the exponent of the Crout's method is higher, implying more rigid adherence to the theoretical serial order of operations (3).

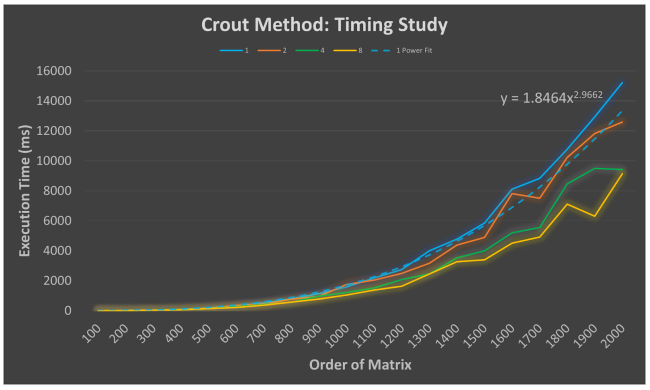

Fig. 3. LU decomposition using Crout's Method

VI. CuDA Implementation of LU Decomposition

One of the problems encountered with the OpenMP implementation was a huge time explosion for higher order matrices. For example, a matrix of order 2400 required at least 1 minute to run regardless of the number of threads.

To solve this problem, we implemented a parallel version of the LU factorisation algorithm in CuDA which runs on

the GPU instead of the less powerful CPU. Naturally, the algorithm had to be slightly modified to ensure better mapping to the GPU. This was done by separating the loops which involve updating the **L** and the **U** parts of the matrix.

Figure 4 shows the part of the matrix that is updated at the i^{th} iteration. To begin with, the most simple approach was implemented wherein each element of this sub-matrix is assigned a thread and a kernel with required number of blocks and threads per block is launched. First, the sub-matrix is divided into blocks as before, but within the block only one thread per row of the block is launched (instead of one thread per element). Second, the same thing is done with columns - one thread per column of the block is launched.

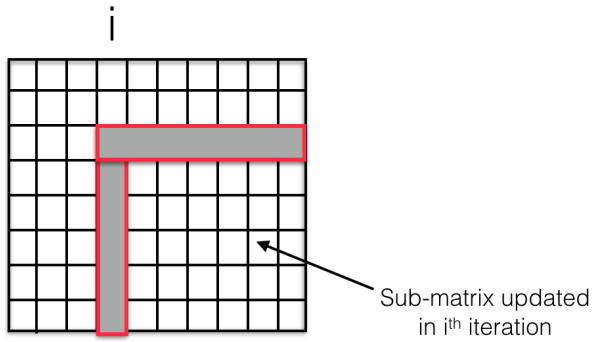

Fig. 4. LU decomposition: Matrix update at i-th iteration (CUDA)

A general algorithm is presented below as a pseudo-code:

```
  LU_kernel(&L,&U) {
    /*initialize random array of size
NxN*/
    /*calculate L and U*/
  }
  LU_main() {
    /*initialise variables & cuda
creation*/
    gpu_memory_allocation(&dev_L,&dev_U)
    cpu_to_gpu_memory_copy(dev_U,U),(dev_L,L);
    LU_kernel(dev_L,dev_U);
    gpu_to_cpu_memory_copy(U,dev_U,L,dev_L);
  }
```

As expected, the time required for LU decomposition increases on increasing the size of matrix. The resulting graph between Order of Matrix n vs. Execution Time is of exponential nature.

The timing study using CuDA was performed on a system running Windows 10 using an Nvidia GTX 1050M GPU and a quad-core Intel i5-9300H CPU.

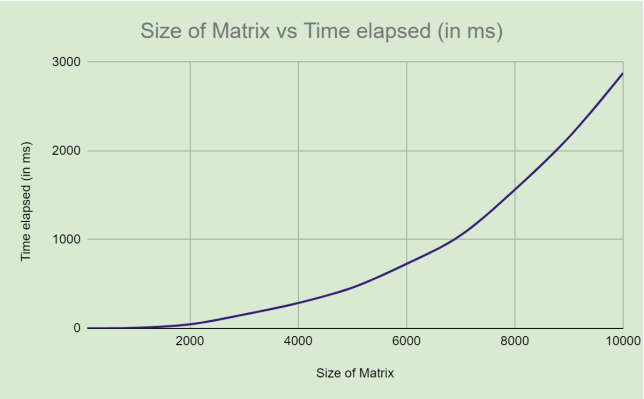

Fig. 5. Timing study using CUDA

REFERENCES

[1] http://www.mymathlib.com/matrices/linearsystems/doolittle.html
[2] https://en.wikipedia.org/wiki/LU_decomposition#Using_Gaussian_elimination
[3] https://github.com/roguexray007/LU-factorisation
[4] https://www.geeksforgeeks.org/doolittle-algorithm-lu-decomposition/
[5] MA214: Introduction to Numerical Analysis course content