KOLEJ UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY

Assignment

BMCS3003 DISTRIBUTED SYSTEMS AND PARALLEL COMPUTING

2021/2022

Student's name/ ID Number	:	Wong Sai Seng / 21WMR05340
Student's name/ ID Number	:	Teh Jin Yang / 21WMR05336
Student's name/ ID Number	:	Wong Zi Xiu / 21WMR05341
Programme	:	RST3S1
Tutorial Group	:	G2
Date of Submission to Tutor	:	25/09/2022

Name(s): Wong Sai Seng, Teh Jin Yang, Wong Zi Xiu Programmer: RST3 Group: G2 Date:21/08/2022

Criteria	Weak	Average	Good	Excellent	Mark
Introduction	No or very little discussion on existing problem and the project The proposed project already exists, or with very minor change. No discussion or very little of introduction given to the related system or technology. 0-9	Little discussion on existing problem and introduction of proposed projects. Minor ideas are modified from existing system(s). Introduction to the related system is given, but no evaluation provided. 10-18	Good discussion and evaluation of existing problems and the proposed project. Ideas modified from existing systems, with some creative ideas are added. Good discussion and evaluation of the related system. 19-24	A very good discussion and evaluation of existing problem and the proposed project. Majority of the ideas are creative. A very good discussion and evaluation of the related system. 25-30	
Literature Review	Contents are retrieved directly from the literature without any paraphrasing. Selected literature was from unreliable sources (e.g. Web sources). No critical evaluation was provided. 0-9	Summary is lengthy, contents are retrieved directly from the literature without any critical evaluation. Selected literature was from unreliable sources. Literary supports were vague or ambiguous. 10-18	Summary is concise and clear. Able to identify key constructs and variables related to the research questions, from the relevant, reliable theoretical and research literature. Discussions on the validity, opinions, arguments and evidence are presented. 19-24	Summary is concise and clear, which integrates critical and logical details from the peer- reviewed theoretical and research literature. Attention is given to different perspectives, conditionality, threats to validity, and opinion vs. evidence. 25-30	
Methodology	No discussion or very little of introduction given to the methods applied. Brief design is provided but lack of explanation. Algorithm suggested is not appropriate or less relevant. 0-9	Brief description of system design, with some explanations. Introduction to the related application of the methods is given, but no evaluation provided. Algorithm suggested is relevant, but lack of understanding or explanation. 10-18	System design is well-illustrated, and with clear explanation. Good discussion and evaluation of the methods applied. Algorithm suggested is relevant and good explanation shows understanding. 19-24	System design is well-illustrated, with good explanation. Good discussion and evaluation of the relevant and practical methods applied. Algorithm suggested is relevant and good explanations to relate to the proposed project. 25-30	
Reference	No proper referencing is done. Only rely on website content, but no research papers. Reference list is not provided. 0-2	Reference list is provided but not with Harvard Referencing standard. Some mixture of reference sources. 3-5	Referencing is done properly. Some mixture of reference sources with at least one journal. 6-8	Rich mixture of reference sources especially good quality of research papers. Proper citations are done whenever necessary. Reference list follows proper Harvard Referencing standard and the information are complete. 9-10	
				Sum of scores	
				Final score = sum of scores/100*40 (base 40%)	

Name(s): Wong Sai Seng, Teh Jin Yang, Wong Zi Xiu Program: RST3S1 Group: G2 Date: 14/09/2022

Program (60%) - CLO1

No	Item	Criteria			Final Marks
		Poor	Accomplished	Good	
1	Output (10)	Inadequate information/outputs needed are generated. Most of the information/outputs generated are less accurate. Results visualization is overly cluttered or the design seems inappropriate for problem area. Lack of information that are useful for the user 0-4	Adequate information/outputs needed are generated. The information/output generated are accurate but some with errors. Pleasant looking, clean, well-organized results visualization The information displayed are useful for the user, but some details are omitted. 5-7	All the necessary information/outputs are generated. All or most of the information/outputs generated are accurate. Minor errors can be ignored. The results are visually pleasing and appealing. Great use of colors, fonts, graphics and layout. The information displayed are useful to the users and complete with necessary details. 8-10	
2	Programming (10)	The end product fails with many logic errors, many actions lacked exception handling. Solutions are over-simplified. Programming skill needs improvement. 0-4	Major parts are logical, but some steps to complete a specific job may be tedious or unnecessarily complicated. Program algorithm demonstrates acceptable level of complexity. The student is qualified to be a programmer 5-7	Correct and logical flow, exceptions are handled well. Demonstrates appropriate or high level of complex algorithms and programming skills. 8-10	
3	Degree of completion (10)	Too much still remain to be done. Basic requirements are not fulfilled. The end product produces enormous errors, faults or incorrect results. 0-4	All required features present in the interface within the required scope, but some are simplified. Or one or two features are missing. The system is able to run with minor errors. 5-7	All required features present in the interface within or beyond the required scope. No bugs apparent during demonstration. 8-10	

4	Program Model Optimization (10)	The model is not optimized. Most of the processes are executed in serial. Only 1 parallel program model is used. 0-4	The model is optimized by using more than 1 parallel program model, i.e. SPMD, loop parallelism. 5-7	The model is optimized by using more than 1 parallel program model, i.e. SPMD, loop parallelism. The model is tested on different parallel platform, i.e. OpenMP (Homogenous), CUDA, OpenCL (Heterogenous). 8-10	

No	Item	Criteria			Final Marks
		Poor	Accomplished	Good	
5	System implementation (10)	The end product is produced with different system design or approach, which is not related to the initial proposal. 0-4	The end product conforms to most of the system design, but some are different from the specification. 5-7	The end product fully conforms to the proposed system design. 8-10	
6	Presentation (10)	The student is unclear about the work produced, sometimes not even knowing where to find the source code. 0-4	The student knows the code whereabouts, but sometimes may not be clear why the work was done in such a way. 5-7	The student is clear about every piece of the work done. 8-10	
				Sum of Score	

Final Report (40%) – CLO3

No	Item	Criter ia				Final Marks
		Missing or Unacceptable	Poor	Accomplished	Good	
1	Title and abstract (10)	Title or abstract were omitted or inappropriate given the problem, research questions and method 0-2	Title or abstract lacks relevance or fails to offer appropriate details about the education issue, variables, context, or methods of the proposed study. 3-4	Title and abstract are relevant, offering details about the proposed research study. 5-7	Title and abstract are informative, succinct, and offer sufficiently specific details about the educational issue, variables, context, and proposed methods of the study. 8-10	
2	Results (Performance measurement) (10)	Analytical methods were missing or inappropriately aligned with data and research design. Results were confusing. 0-2	Analytical method was identified but the results were confusing, incomplete or lacked relevance to the research questions, data, or research design. 3-4	The analytical methods were identified. Results were presented. All were related to the research question and design. Sufficient metric or measurement is applied. 5-7	Analytical methods and results presentation were sufficient, specific, clear, structured and appropriate based on the research questions and research design. Extra metric or measurement is applied. 8-10	
3	Discussion and Conclusion (10)	Discussions or answers to the research question and system performance were omitted or confusing. No or very little conclusion could be yielded. 0-2	Little discussions were presented. Answers to the research question and system performance were unclear or confusing. 3-4	Discussions of the results were presented. The research question and system performance were answered and identified. 5-7	The significance of the results of the work was discussed, sufficiently inclusive of the information that concluded and answered the research question and system performance is evaluated comprehensively. Limitations and future improvements of the studies were identified. 8-10	
4	Organization (5)	The structure of the paper was incomprehensible, irrelevant, or confusing. Transition was awkward. 0-1	The structure of the paper was weak. Transition was weak and difficult to understand. 0-2	A workable structure was presented for presenting ideas. Transition was smooth and clear. 3-4	Structure was intuitive and sufficiently inclusive of important information of the research. Transition from one to another was smooth and organized. 5	
5	Spelling, Grammar and Writing Mechanics (5)	There were so many errors that meaning was obscured, make the content became difficult to understand 0-1	Some grammar or spelling errors were spotted. Some sentences were awkwardly constructed so that the reader was occasionally distracted. 0-2	There were occasional errors, but they did not represent a major distraction or obscure meaning. 3-4	Sentences were well-phrased. The writing was free or almost free of errors. 5	

Sum of Score	
Final score = sum of scores/100*60 (base 60%)	

Table of Content

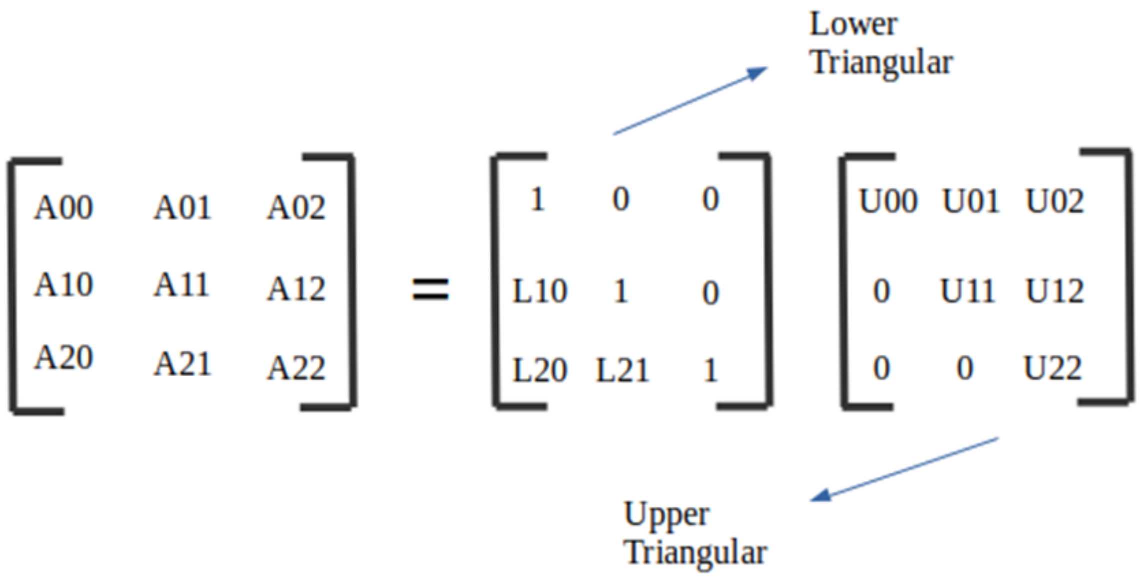
Abstract

For explaining the interactions and processes between two variables, generating predictions, calculating rates, performing conversions, etc., linear equations have gained popularity in the physical world, science, and our daily uses. One of the scientific disciplines that employs a set of linear equations to create a matrix that can be solved using Lower-upper (LU) decomposition is image reconstruction. However, if the matrix is very large, the LU decomposition's performance will deteriorate and maybe become worse due to the difficulty of O(N3) calculation time. In order to increase the efficiency and speed of the LU decomposition's calculation, this article aims to parallelize it using OpenMP, CUDA, and MPI. OpenMP and CUDA had some success as a result of the implementation, however MPI performed worse than serial execution due to the distributed memory problem. Because both the GPU and the CPU are used, OpenMP performs better in smaller matrix sizes whereas CUDA performs better in bigger sizes when comparing the two.

LU Decomposition to Solve Systems of Linear Equations

1. Introduction

Often in computer science, it is important to solve a linear equation of the form A*x = B, with A being a matrix, x being a vector, and B being the solution vector. Rather than directly solving by matrix multiplication, A can be broken into two matrices L and U which is Lower Matrix and Upper Matrix. Lower Matrix which has non-zero values in a lower diagonal region of the matrix and the upper matrix which has non-zero values in the upper diagonal region of the matrix. A picture will likely do better than just words at explaining the forms these matrices take:



The reason for doing this is to get the L matrix and U matrix. Then we can solve for y by using L*y = b (forward substitution), and then use U*x = y to solve for the vector x(background substitution). So with this method, we can solve for many vectors. It can also be used to find the determinant of a matrix. After finding L and U, it can now be used to get the product of their diagonal indexes, which can be used to find the determinant of A.

The goal, however, is not just to write a regular, sequential algorithm that computes the L and U matrices for any non-invertible matrix A. That is just the first step. For the next three steps are to implement three parallel programs based on the sequential programs we create in step 1. Those three programs are Cuda (GPU-based), OpenMp (shared memory), MPI (distributed memory). For the original because LU decomposition has a format O(n^3), so for large matrix sizes, running them sequentially is not feasible. With the three parallel programs that can run parallel would make larger matrix sizes more practical to solve.

2. Literature Review

LU-decomposition which is also known as lower–upper (LU) decomposition is an algorithm that is used in factors of a square matrix into two triangular matrices, a lower triangular matrix and an upper triangular matrix. LU-decomposition was introduced by the Polish Mathematician, Tadeusz Banachiewicz in 1938. LU-decomposition can be viewed as a modified form of the matrix form of Gaussian Elimination. LU decomposition is an effective procedure for solving Linear Equations, this method usually used in numerical linear algebra to solve linear equations. LU-decomposition is an easy and attractive method to solve large-scale Linear Equations. The equation for the LU-decomposition is reviewed (check out figure 2.1 for its equation). This method is usually used whenever it is possible to model the problem to be solved into matrix form. As, by converting it into matrix form and solving with triangular matrices, in order to make it easy to do calculations in the process of finding the solution (Nishant Arora, 2021).

$$\mathbf{A} = \mathbf{LU}$$

Figure 2.1: LU-decomposition Equation

A = LU which A represents the square matrix while L represents lower triangular matrix and U represents upper triangular matrix, L and U have the same dimension of A. Below (figure 2.2) is an example, for a 3 × 3 matrix A, it's how the LU-decomposition looks like:

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

Figure 2.2: 3 × 3 Matrix in LU-decomposition

$$\begin{aligned}
&1)\ a_{11} = 1 \cdot u_{11} \\
&2)\ a_{12} = 1 \cdot u_{12} \\
&3)\ a_{13} = 1 \cdot u_{13} \\
&4)\ a_{21} = \ell_{21} \cdot u_{11} \\
&5)\ a_{22} = \ell_{21} \cdot u_{12} + 1 \cdot u_{22} \\
&6)\ a_{23} = \ell_{21} \cdot u_{13} + 1 \cdot u_{23} \\
&7)\ a_{31} = \ell_{31} \cdot u_{11} \\
&8)\ a_{32} = \ell_{31} \cdot u_{12} + \ell_{32} \cdot u_{22} \\
&9)\ a_{33} = \ell_{31} \cdot u_{13} + \ell_{32} \cdot u_{23} + 1 \cdot u_{33}
\end{aligned}$$

Figure 2.3: Standard matrix multiplication procedure

Based on the figure 2.3 above, it clearly shows how to find the unknown entries of matrices L and U, from the first three equations we immediately get the values of U11, U12 and U13. Then, the 4th and 7th equations allow us to find L21, L31. Thus, the 5th and 6th equations allow us to find U22, U23. Finally, the last two equations will produce the solutions for L32 and U33.

Based on the figure (figure 2.4 & 2.5) below, the matrix on the right side is the upper triangular matrix that has all the elements below the main diagonal as zero. While the left matrix is the lower triangular matrix which has elements above the main diagonal as zero is called a lower triangular matrix. The figure also shows the difference between the upper triangular matrix and the lower triangular matrix. Matrix are helpful for mathematical calculations and these triangular matrices are easy to solve.

Lower Triangular Matrix (L) Upper Triangular Matrix (U)

$$\begin{bmatrix} * & & & 0 \\ * & * & & \\ * & * & * & \\ * & * & * & * \end{bmatrix} \qquad \begin{bmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ 0 & & & * \end{bmatrix}$$

Figure 2.4 & 2.5: Lower & Upper triangular matrix

2.1 LU-decomposition

LU-decomposition is a method to decompose a matrix which means to decompose (or factorize) a square matrix and to produce two triangular matrices. This method can significantly simplify some of the matrix operations because the decomposition of the original matrix has special properties, so that we can easily perform various operations on the rather produced triangular matrices than on the original matrix. Consider the matrix shown below is the matrix A:

$$\begin{bmatrix} 1 & 1 & 1 \\ 3 & 1 & -3 \\ 1 & -2 & -5 \end{bmatrix}$$

Figure 2.1.1: Matrix A

Solution:
Initial Matrix, identify the matrix L

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boxed{1} & 1 & 1 \\ 3 & 1 & -3 \\ 1 & -2 & -5 \end{bmatrix}$$

R2 -(3)R1 ->R2, subtract row 1 multiplied by 3 from row 2

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boxed{1} & 1 & 1 \\ 0 & -2 & -6 \\ 1 & -2 & -5 \end{bmatrix}$$

R3 -(1)R1 ->R3, subtract row 1 from row 3

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & \boxed{-2} & -6 \\ 0 & -3 & -6 \end{bmatrix}$$

R3 -($\frac{3}{2}$)R2 ->R3, subtract row 2 multiplied by $\frac{3}{2}$ from row 3

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 1 & \frac{3}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & -2 & -6 \\ 0 & 0 & 3 \end{bmatrix}$$

Finally

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 1 & \frac{3}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & -2 & -6 \\ 0 & 0 & 3 \end{bmatrix}$$

The figure shown above is the result of the Matrix A, which left side represents the lower triangular matrix while the right side represents the upper triangular matrix.

2.2 Doolittle Algorithm

Doolittle’s method provides an alternative way to factor A into an LU decomposition without going through the hassle of Gaussian Elimination. Consider, there’s a general n×n matrix A and assume that an LU-decomposition exists, and write the form of L and U explicitly. Then, use the systematic manner to solve the entries in L and U from the equations that result from the multiplications necessary for A=LU.

The terms that has be used for the U matrix are described as belows:

$$\forall j$$
$$i = 0 \rightarrow U_{ij} = A_{ij}$$
$$i > 0 \rightarrow U_{ij} = A_{ij} - \sum_{k=0}^{i-1} L_{ik} U_{kj}$$

And the term that used for the L matrix is described as belows:

$$\forall i$$
$$j = 0 \rightarrow L_{ij} = \frac{A_{ij}}{U_{jj}}$$
$$j > 0 \rightarrow L_{ij} = \frac{A_{ij} - \sum_{k=0}^{j-1} L_{ik} U_{kj}}{U_{jj}}$$

Example:

$$X_1 + X_2 + X_3 = 5$$

$$X_1 + 2X_2 + 2X_3 = 6$$

$$X_1 + 2X_2 + 3X_3 = 8$$

Figure 2.2.1

Solution:
In order to solve the equations shown above (figure 2.2.1) with Doolittle’s method, first it will create matrices A, B, and X. where A is used to indicate the augmented matrix, B used to indicate the constant, while X used to constitute the variable vectors.

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} \quad X = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \quad B = \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix}$$

Consider A = LU

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ b & c & 1 \end{bmatrix} * \begin{bmatrix} d & e & f \\ 0 & g & h \\ 0 & 0 & i \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} d & e & f \\ ad & ae+g & af+h \\ bd & be+cg & bf+ch+i \end{bmatrix}$$

$$\begin{array}{lll} d = 1 & e = 1 & f = 1 \\ ad = 1 & ae + g = 2 & af + h = 2 \\ a = 1 & g = 1 & h = 1 \\ bd = 1 & be + cg = 2 & bf + ch + i = 3 \\ b = 1 & c = 1 & i = 1 \end{array}$$

Figure 2.2.2

Based on the figure 2.2.2 above, it clearly shows how to find the unknown entries of matrices L and U with Doolittle’s method. So, we just need to calculate the value of each entry and that's the result of the lower triangular matrix and the upper triangular matrix.

2.3 Conclusion

Conclusion, the LU-decomposition method will be chosen instead of the Doolittle’s method because it involves simpler arithmetic calculations and is easier to be parallelised. As there's a lot of calculation, we used lock so that it would not get the wrong value or multiple values at the same time.

3. Improvement of Chosen Method

The suggested serial code is the same as the proposal. Three parallelism implementations are suggested for each of the methods from the serial code. Our code's primary goal will be to parallelize the ComputeLUDecomposition(float** a, float** L, int n) function, which actually does LU decomposition. The other functions, like printing, getUserInput and delete matrix are not parallelized since they are very implementation-dependent, and some of them are even impossible to parallelize without generating out-of-order problems.

3.1 OpenMP

OpenMP is a library that allows shared memory multiprocessing. All threads in an OpenMP application share memory and data since SMP (symmetric multi-processors) also known as shared-memory processors) is the programming paradigm used. In this topic that we need to implement, #pragma omp parallel is the most basics directive to implement to the program and it used to fork more threads in order to complete the task contained in the block after the #pragma construct.

```
for $k \leftarrow 0$ to $n-1$ do
    /* Division step */
    #pragma omp parallel for
    for $i \leftarrow k+1$ to $n-1$ do
        $a[i][k] \leftarrow a[i][k]/a[k][k]$
    /* Elimination step */
    #pragma omp parallel for
    for $i \leftarrow k+1$ to $n-1$ do
        for $j \leftarrow k+1$ to $n-1$ do
            $a[i][j] \leftarrow a[i][j] - a[i][k] * a[k][j]$
```

Figure 3.1.1 LU factorization using the OpenMP parallel row block method

Figure above shows the OpenMP parallel row block algorithm of LU factorization. The nxn coefficient matrix A is block-striped across p threads or cores in row block data distribution, with each core receiving np consecutive rows of the matrix (Panagiotis D. Michailidis; Konstantinos G. Margaritis, 2011). RowBlock is an example of an OpenMP parallel algorithm that makes use of the row block data distribution in figure 3.2.1.1 above.

Beside that, the #pragma omp for schedule(static,bs) can be also implemented to the topic that we study for solving the lu decomposition using the OpenMP parallel row cyclic method. We have control over how the threads are scheduled with OpenMP. The many schedules that are offered include which are static, dynamic and guided. In this case, the static schedule is used in this program. Each thread is given a set number of iterations in a static manner (round robin). Equal amounts of iterations are allocated to each thread. When the chunk argument is given an integer, a certain thread will get the chunk number of consecutive iterations.

```
for $k \leftarrow 0$ to $n-1$ do
    /* Division step */
    #pragma omp parallel for schedule(static, bs)
    for $i \leftarrow k+1$ to $n-1$ do
        $a[i][k] \leftarrow a[i][k]/a[k][k]$
    /* Elimination step */
    #pragma omp parallel for schedule(static, bs)
    for $i \leftarrow k+1$ to $n-1$ do
        for $j \leftarrow k+1$ to $n-1$ do
            $a[i][j] \leftarrow a[i][j] - a[i][k] * a[k][j]$
```

Figure 3.1.2 LU factorization using the OpenMP parallel row cyclic method

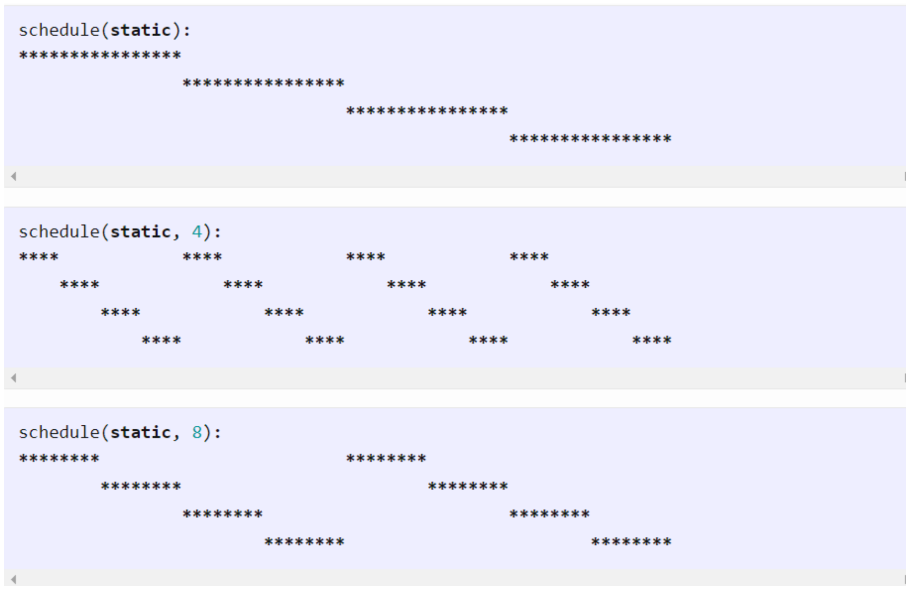
In the figure shown above, it employs a static schedule with a variable-specified chunk size for row-cyclic data distribution, which defines a static distribution of iterations to threads that distribute blocks of size bs in a round-robin way to the available threads. bs can take the values 1, 2, 4, 8, 16, 32, or 64 rows (Panagiotis D. Michailidis; Konstantinos G. Margaritis, 2011).

```
#pragma omp parallel private(k, i, j, row) shared(a)
{
long $my\_rank$ = omp_get_thread_num();
$bsize = n/p$;
if $(my\_rank$ != $0)$ then
    for $k \leftarrow (my\_rank * bsize)$ to $(my\_rank * bsize) + bsize$ do
        $row$ = Get($my\_rank$);
        Put($my\_rank + 1, row$);
        /* Division step */
        for $i \leftarrow k$ to $k + bsize$ do
            $a[i][row] = a[i][row]/a[row][row]$;
        /* Elimination step */
        for $i \leftarrow k$ to $k + bsize$ do
            for $j \leftarrow k$ to $n + 1$ do
                $a[i][j] -= (a[i][row] * a[row][j])$;
else
    for $k \leftarrow (my\_rank * bsize)$ to $(my\_rank * bsize) + bsize$ do
        Put($my\_rank + 1, k$);
        /* Division step /*
        for $i \leftarrow k + 1$ to $k + bsize$ do
            $a[i][k] = a[i][k]/a[k][k]$;
        /* Elimination step */
        for $i \leftarrow k + 1$ to $k + bsize$ do
            for $j \leftarrow k + 1$ to $n + 1$ do
                $a[i][j] = (a[i][k] * a[k][j])$;
}
```

Figure 3.1.3 LU factorization using the OpenMP pipelining method

The suggested Get() and Put() functions can be used to implement the receive and send operations in OpenMP, respectively. While the Put() function can take two parameters, one of which is the rank of the thread that will submit the data and the other of which is a data element, the Get() process can only accept one argument, the rank of the thread that will receive the data, and it returns a data element (Panagiotis D. Michailidis; Konstantinos G. Margaritis, 2011). The following paragraph will go into the internals of the Get() and Put() methods. An OpenMP parallel pipeline technique which is identified as Pipe in figure above is displayed.

3.1.1 OpenMP Scheduling



```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```

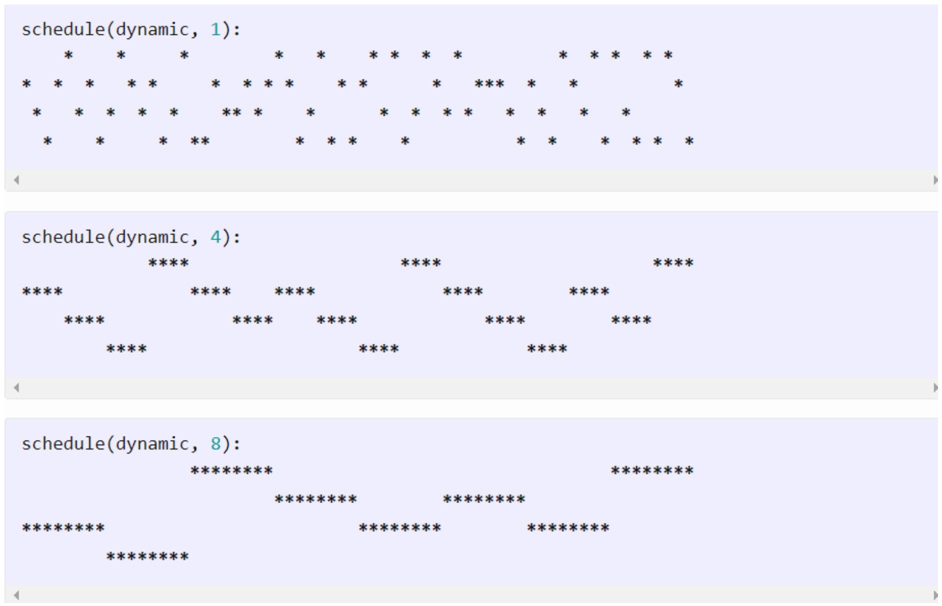
```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

Figure 3.1.1.1 Static scheduling with chunks size 1,4 and 8

In OpenMP, scheduling is a technique for allocating for loop iterations to various threads. The loop construct's schedule(static, chunk-size) clause indicates that the for loop has a static scheduling type. The iterations are split into chunks of size chunk-size by OpenMP, which then distributes the chunks across the threads in a cyclical fashion (Jaka's Corner, 2016).

When there is no chunk-size supplied, OpenMP breaks iterations into roughly equal-sized chunks and assigns no more than one chunk to each thread.



```
schedule(dynamic, 1):
    *    *     *         *   *    * *  *  *         *  * *  * *
*  *  *   * *     *  * * *    * *       *  ***   *   *         *
 *   *  *  *  *    ** *     *      *  *  * *   *  *   *   *
  *    *     *  **         *  * *    *          *  *    *  *  *  *
```

```
schedule(dynamic, 4):
            ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

```
schedule(dynamic, 8):
                ********                                ********
                        ********        ********
********                        ********        ********
        ********
```

Figure 3.1.1.1 Dynamic scheduling with chunks size 1,4 and 8

Besides that, the loop construct's schedule(dynamic, chunk-size) clause identifies the for loop's dynamic scheduling type. Iterations are divided into chunks of size chunk-size by OpenMP. Up until there are no more chunks left, each thread does a chunk of iterations before requesting another piece (Jaka's Corner, 2016).

The distribution of the pieces to the threads is done in any sequence. When we run the for loop, the order is different each time.

Therefore, the static and dynamic scheduling method will be implemented to this OpenMP program to improve the performance of the program. But in the program we will only use the default chunk size which is 1 to let n thread to run one loop for the initializeMatrix, initializeLowerMatrix and the ComputeLuDecomposition function.

3.1.2 Shared, Firstprivate and private

The shared clause states that all of the threads in a team will share the variables in the list. The same storage location for shared variables is accessed by each thread within a team.

Every thread may obtain a unique copy of a shared variable by using the private clause, but the shared variable's initial value is still ambiguous.

Firstprivate specifies that a variable should be initialized with the value of the variable because it existed prior to the parallel construct and that each thread should have its own instance of the variable.

In this program, there are some of the variables that will be defined as shared variable, firstprivate and private variable to make sure the the data in those variables won't crash while the multiple threads are going to access the program. For example the A variable is storing the value of the array and it can be shared by the multiple threads to access the array while the A array is updated.

3.1.3 SetLock

A collection of all-purpose lock functions that can be used for synchronization are included in the OpenMP runtime library. These OpenMP locks are represented by OpenMP lock variables, and these general-purpose lock functions work with those locks. Programs that access OpenMP lock variables in any other way are non-conforming. Only the procedures defined in this section may access OpenMP lock variables (OPENMP API Specification, 2018).

An OpenMP lock may be initiated, unlocked, or locked, among other states. A task can set a lock if it is in the unlocked state, changing its status to locked. It is thus argued that the job that sets the lock owns it. A lock that is owned by a task can be unset, bringing it back to the unlocked state. A non-conforming programme is one in which one process releases a lock that belongs to another job.

In this program, the simple lock is only the lock that is used to lock a certain part of the code to ensure that no two processes change it at the same time.

3.2 Mpi

The MPI stands for Message Passing Interface that is a well-liked library-based architecture for large-scale parallel programming. In other words, MPI is a standardized way for computers executing a parallel application in distributed memory to communicate with one another.

According to the study (Ben LeMarc, 2019), the best method to make any implementation work was to initially only focus on sending all of the data to the threads, so performing the computation correctly was not a problem. For each iteration of the outer loop in the research, he gathered and broadcast the L and U array once, which is quite expensive. It costs money to assemble the columns, which is essential to calculate U, and to convert the gather operation's output from column-major order to row-major order.

First and foremost, create an MPI implementation that operates as intended, where each worker thread sends back a single row of L (calculated in the first middle loop) and a single column of U (calculated in the second middle loop). This functioned as long as the number of rows and the number of threads used were equal. The MPI functions for sending and receiving data took the place of the middle loops in this programme version.

He modified it to compute rows per process, which is derived by taking the number of rows divided by the number of processes, to handle cases when there are more rows than threads. The number of rows (or columns) computed per thread and returned to process 0 is determined by this. This works as long as the number of threads being used divides the number of rows. The middle loops were reinstated with this new version, and each worker thread now iterates over them as many times as there are rows (or columns) to calculate.

3.2.1 MPI_Send and MPI_Recv

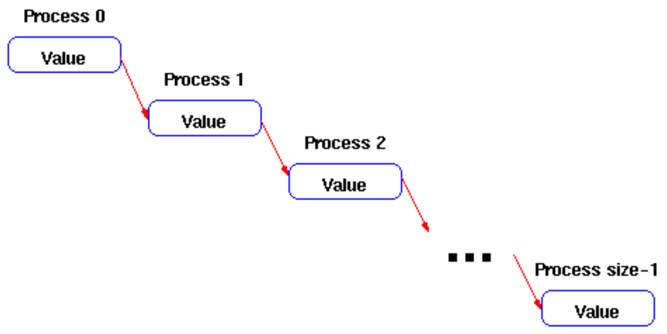


Figure 3.2.1.1 Send and receive in MPI

The send and receive calls of MPI function as follows. Process A first determines that Process B needs to receive a message. Then, process A gathers all of the data it requires into a buffer for process B. Since the data is being bundled into a single message before transmission, these buffers are sometimes referred to as envelopes (similar to how letters are packed into envelopes before transmission to the post office). The communication device (which is frequently a network) is in charge of routing the message to the appropriate place once the data has been packed into a buffer. The rank of the process determines where the message should be placed (Wes Kendall, 2020).

3.2.2 MPI_Bcast

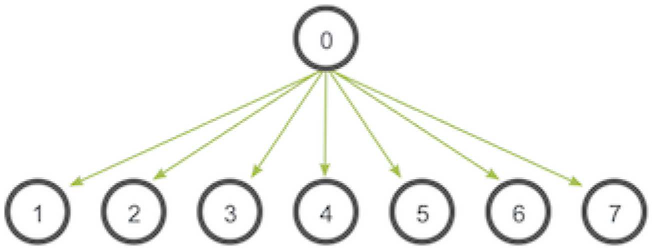


Figure 3.2.2.1 Broadcasting with MPI_Bcast

Figure above shows the broadcasting with MPI_Bcast, One of the common methods of group communication is the broadcast. One process provides the identical data to all other processes in a communicator during a broadcast. Sending user input to a parallel programme or all processes with configuration settings is one of the primary purposes of broadcasting (Wes Kendall, 2022).

The same MPI Bcast function is called by both the root and recipient processes, despite the fact that they perform separate tasks. The data variable is delivered to all other processes when the root process (in our case, process zero) executes MPI Bcast. The data variable will be populated with the data from the root process when all of the recipient processes run MPI Bcast(Wes Kendall, 2022).

3.2.3 MPI_Barrier

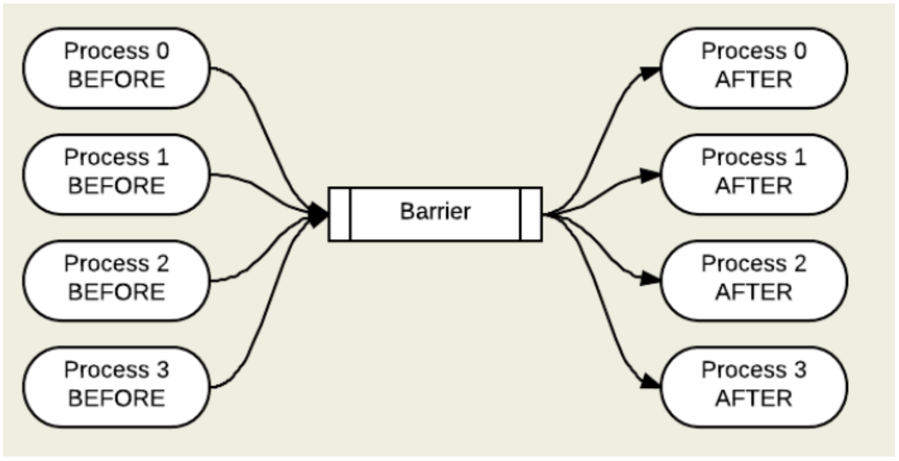


Figure 3.2.3.1 MPI_Barrier

MPI_Barrier, a barrier that is used when you want all the processes to finish a certain section of code before moving on. When you call the MPI Barrier function after adding the call, use this exercise to confirm that it is happening. The BEFORE strings should all be printed before all of the AFTER strings after the barrier call has been added. With time going from left to right, you can see the barrier function used to execute the programme as the figure shown above.

3.3 Cuda

A platform and programming language for GPUs with CUDA support is called CUDA. GPUs for general purpose computing are exposed by the platform. C/C++ language extensions and APIs for controlling and programming GPUs are provided by CUDA. Both CPUs and GPUs are utilized for computation in CUDA programming. CPU and GPU systems are typically referred to as hosts and devices, respectively. Separate platforms with their own memory spaces are CPUs and GPUs. Typically, we offload parallel computation to GPUs and execute serial tasks on CPU.

The primary focus of the work is on how to achieve parallel processing for the LU calculation. On assuring performance for the L and U matrices, it is decided how to allocate processors. The LU kernel () function's algorithm stages are listed below.

```
LU_kernel(matrix_L,matrix_U) {
        initializing_variables;
        calculation_matrix_L;
        calculation_matrix_U;
}
```

The following steps are taken to prepare the LU main () function for the solution of a system of linear equations. The essential codes for the solution of the equation system are written, as well as the definition and initialization of the variables.

```
LU_main()
{
        initializing_variables;
        cuda_event_creation{&start, &stop};
        cuda_device_properties_check{&prop, 0};
        gpu_memory_allocation{&dev_L, &dev_U};
        timer_record{start, 0};
        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}};
        calculation_result_matrix_X;
        timer_record{stop, 0};
        gpu_memory_deallocation{dev_L, dev_U};
}
```

For the purpose of calculating working time, programme codes have been created. It is possible to allocate memory for the CPU and GPU. Data is copied from CPU to GPU, followed by a call to the LU kernel() function and another copy of computed values from GPU to CPU. Computing the result vector X comes after calculating the L and U matrices (Caner Ozcana, Baha Sena, 2011). Because of parallelism, the operation carried out in the main function cannot calculate the result vector x. At the program's conclusion, computed matrices and vectors are shown on the screen.

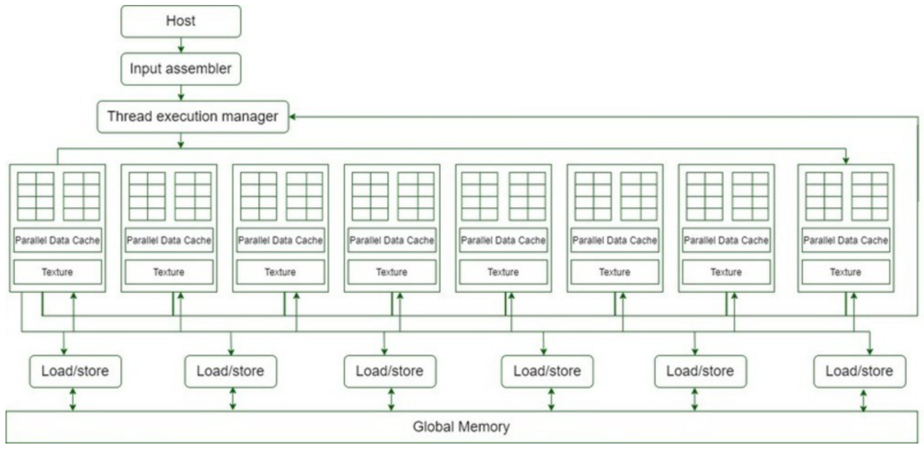


Figure 3.2.3.1 Architecture of CUDA

The aforementioned graphic displays 16 Streaming Multiprocessor (SM) diagrams. We obtain a total of 128 streaming processors since each streaming multiprocessor contains 8 streaming processors (SPs). The MAD unit (Multiplication and Addition Unit) and an extra MU are now present in each Streaming processor (multiplication unit). More than 1 TFLOP of computing capability is available on the GT200's 240 Streaming Processors (SPs). A single Streaming Processor may accommodate thousands of threads per programme and is smoothly threaded. Each Streaming Multiprocessor may have 768 threads supported by the G80 card (note: not per

SP). Eventually, each SP will handle a maximum of 96 threads once each Streaming Multiprocessor has 8 SPs. There are a total of 128 * 96, or 12,228 possible threads. Thus, the term "massively parallel" refers to these processors. The memory bandwidth of the G80 processors is 86.4 GB/s. Additionally, an 8GB/s communication channel (4GB/s for uploading to and 4GB/s for downloading from the CPU RAM) is available.

3.3.1 Indexing Arrays with Blocks and Threads

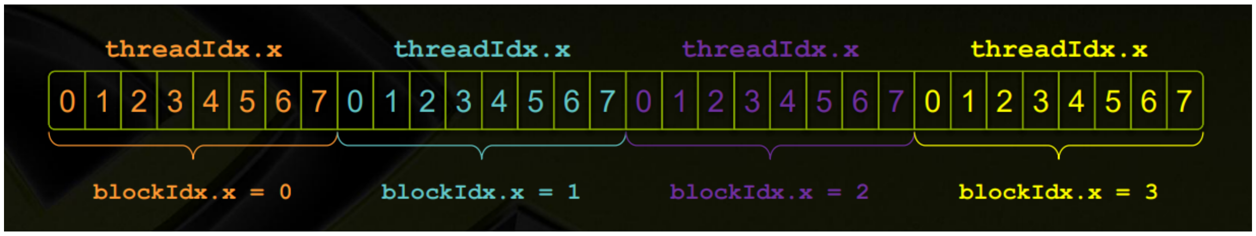


Figure 3.3.1.1 blockIdx.x and threadIdx.x indexing arrays

Figure above shows the indexing arrays with the blocks and threads. The 32 banks in shared memory are arranged so that following 32-bit words correspond to following banks. Then it will allocate a __shared__ array big enough to store all the threadblock's private arrays for the sake of the example (Maxim Milakov, 2015).

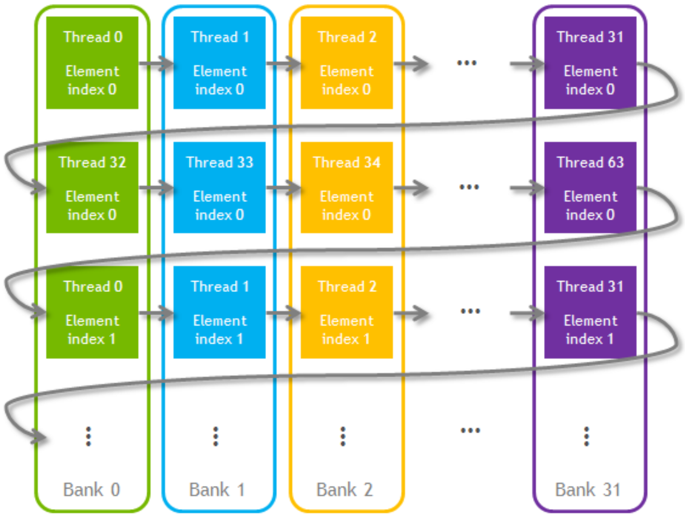


Figure 3.3.2 private arrays can be stored without conflict in shared memory. The thread block size in this illustration is 64.

To ensure that every member of our new virtual private array for each thread is saved in its own shared memory bank, we will logically assign elements of this new __shared__ array to the threads of the thread block. To specify the size of the thread block, it will use THREADBLOCK SIZE (this value should be evenly divisible by the warp size, 32). Here, all of the 0-index items of the private arrays for each thread in the thread block are contained in the first THREADBLOCK SIZE elements of our shared memory array. All 1-index elements from the private arrays are contained in the following THREADBLOCK SIZE elements of the shared memory array, and so on. Figure above provides an illustration of this strategy (Maxim Milakov, 2015).

4. Results

4.1 Serial Runtime Output

Test Environment	Detail
CPU :	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
Motherboard :	HP 85FA 42.34
Graphic Card :	NVIDIA GeForce GTX1050
RAM :	8.00 GB

Table 4.1: Testing Environment for Serial

Matrix size	Runtime
64	0.0001
128	0.0030
256	0.0200
512	0.1350
1024	1.0400
2048	8.0250
4096	65.6840

Table 4.1.1: Runtime for Serial code

4.2 OpenMP Runtime Output

Test Environment	Detail
CPU :	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
Motherboard :	HP 85FA 42.34
Graphic Card :	NVIDIA GeForce GTX1050
RAM :	8.00 GB

Table 4.2: Testing Environment for OpenMP

Thread	Matrix Size						
	64	128	256	512	1024	2048	4096
2	0.0019	0.0043	0.0249	0.1291	0.7883	5.4266	45.3904
4	0.0027	0.0367	0.0367	0.1189	0.6258	3.9554	32.5799
8	0.0038	0.0089	0.0293	0.1013	0.4749	2.8785	21.9018
16	0.0046	0.0094	0.0424	0.1123	0.5131	3.1234	23.8261
32	0.0110	0.0352	0.0500	0.1435	0.5023	2.9732	24.5766
64	0.0194	0.1242	0.1013	0.2225	0.5571	3.6703	27.6794
128	0.0568	0.0659	0.1109	0.2331	0.7388	3.6760	27.5726

Table 4.2.2: Runtime for OpenMP with different numbers of thread

4.3 Mpi Runtime Output

Test Environment	Detail
CPU :	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
Motherboard :	HP 85FA 42.34
Graphic Card :	NVIDIA GeForce GTX1050
RAM :	8.00 GB

Table 4.3 Testing Environment for MPI

Thread	Matrix Size						
	64	128	256	512	1024	2048	4096
2	0.0033	0.0024	0.0130	0.0930	0.6898	5.7716	48.9760
4	0.0028	0.0036	0.0145	0.0746	0.5670	4.7155	34.1139
8	0.0113	0.0290	0.0363	0.1239	0.4709	4.5415	32.1879
16	0.0387	0.1157	0.2006	0.6053	2.4968	12.0876	54.6287
32	0.1709	0.4935	1.2312	1.9523	3.9260	37.3333	121.9214

64	0.2487	1.0198	2.5664	5.9817	12.0286	109.6732	>300.000
128	-	1.9173	6.0338	11.3617	27.2769	286.2489	>300.000

Table 4.3.2: Runtime for MPI with different numbers of thread

4.4 Cuda Runtime Output

Test Environment	Detail
CPU :	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
Motherboard :	HP 85FA 42.34
Graphic Card :	NVIDIA GeForce GTX1050
RAM :	8.00 GB

Table 4.4: Testing Environment for Cuda

Thread	Matrix Size	Runtime
64	64	0.2350
128	128	0.1570
256	256	0.1670
512	512	0.1640
1024	1024	0.1750
2048	2048	0.2090
4096	4096	0.2970

Table 4.4.2: Runtime for Cuda

5. Discussion

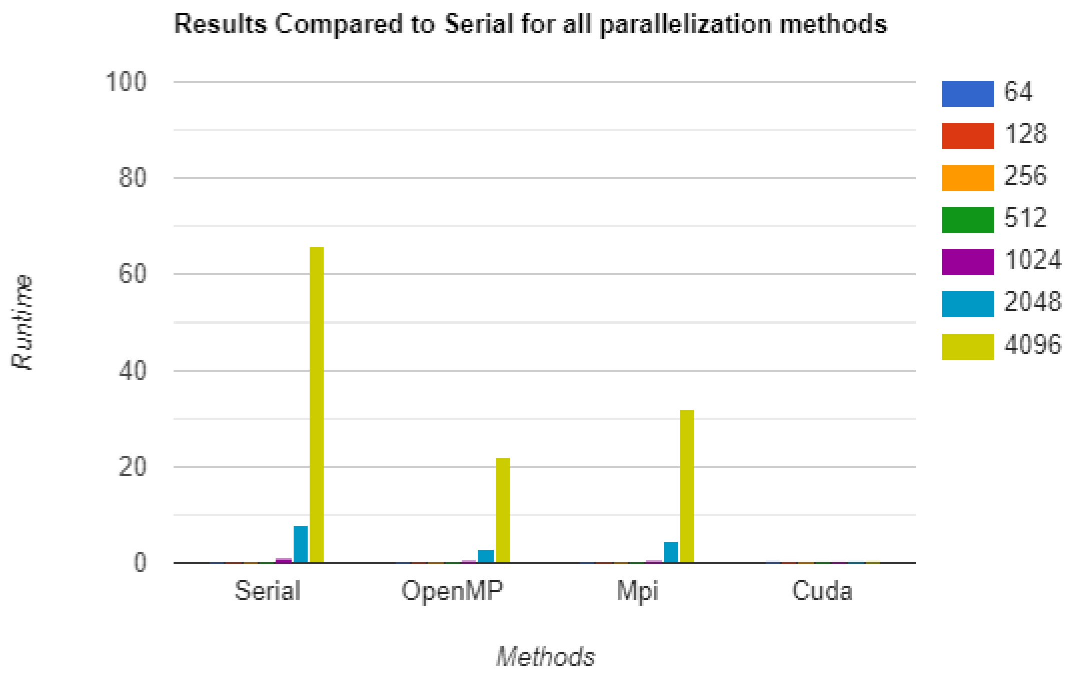


Figure 5.1 Results Compared to Serial for all parallelization methods

5.1 Serial

The main reason a serial algorithm is implemented is to provide a basis for the rest of our parallel algorithm, provide a baseline execution time, as well as to verify that the answer is correctly computed after parallelization. Firstly, the serial algorithm is executed for 7 times on 64x64, 128x128, 256x256, 512x512, 1024x1024, 2048x2048 and 4096x4096. Afterwards, an online calculator is used to confirm whether the values are accurate. Once the sequential algorithm is verified to produce correct results, optimizations can then be done to parallelize the serial algorithm.From the table 4.1.1, we can see that the execution time is sequentially increased when the matrix size becomes larger.

5.2 OpenMP

The first column in table 4.2.2 shows the performance of different matrix sizes under different numbers of threads. First of all, speedups below a factor of 1 are considered to be a slowdown.The performance of the execution time follows a similar trend to the serial execution time.

For matrix size at 256x256 and above, as the number of threads increases, the performance generally increases, up to a certain point before it degrades, depending on the size of the matrix. Maximum speedup is observed at the highest matrix size, 4096x4096 as expected, with a speed up of 2.3822 times compared to the serial runtime. The larger the matrix, the higher the threshold for the number of threads for optimal speedup. The main reason for this is due to the increased overhead. When there are more software threads than hardware threads available, Windows resorts to round-robin scheduling. Each thread then gets a short turn or a time slice, to run on a hardware thread. When the turn runs out, the scheduler pauses the thread and allows the next thread in waiting to run on the hardware thread (). Overall we have observed that when the thread is 8, it has the fastest execution time in every matrix size.

The main issue is that switching between the LU decomposition time slices requires saving the register and cache state of a thread, and then restoring them later. Saving a register state and then restoring requires a significant amount of time even with the optimizations built into the CPU. This is because cache is very finite, the processor needs to evict data from the cache for new data, and so typically, the least recently used data will be evicted, which is typically data from the earlier time slice. Too many software threads will cause cache fighting issues which will hurt the performance.

Another interesting part is for the matrix size at 128x128 and below, negative speedup is observed. Because the matrix size is too small, overheads arising from the initializing time, finalization time, external libraries, and communication cost between threads is too significant for the parallelization to overcome when the processing requirements are too small.

5.3 MPI

For the LU decomposition, the MPI application implements data parallelism to improve the performance of it. The matrices have been partitioned and distributed equally to all the processors and each of them only needs to operate a part of the data. The reason for using this is to let each of the processors work independently to the parts of data that are handled by eac processor which will increase the computation time. However, the processes are still required to exchange the data periodically through the communicator because every part of the data needed for a process and we need to keep tracking it to ensure the calculation process performs correctly.

Based on the result shown in the table 4.3.2, that 2 processors have faster speed in this LU decomposition calculation than 4, 8, 16, 32 or 64 processors because two processors require lesser communication time and thus lesser parallel run time is needed as well. So, the process that the matrices run by using 2 processors with the size of 64x64, 128x128, 256x256, are the fastest among the sizes mentioned. While the size of 512x512, 4 processors had the fastest and for the size of 1024x1024, 2048x2048 and 4096x4096 were 8 processors had the fastest run time.

The method (MPI_Send and MPI_Recv) has been used as a point-to-point communication method. According to this method, the channel is only shared between two processes where one is the transmitter, and the other is the receiver. A loop is required for the root process to send the same message to each process in point-to-point communication, which is cumbersome.

In summary, the MPI method that runs in 2 threads overall performs the best among other threads as it has lesser execution time and lesser parallel run time is required.

5.4 Cuda

The CUDA has been implemented by using shared Memory where blocks with static size are used. The thread size of 128 (fine-grained parallel) provides the most better performance compared to the other thread sizes. So, it is selected for the overall comparison among the OpenMP, MPI and CUDA solutions. In other words, the fine-grained decomposition is performed when the LU decomposition problem is decomposed into a large number of tasks. In this case, because the CUDA was working on the same threads and matrix size , it doesn't provide a clear answer on whether the LU decomposition is working with small problem size, or big problem size. So, there will be no thread size that is optimal to utilize. Based on the result on table 4.4.2, which indicates that when the matrix size increase the threads needed to perform the process also increased and cause the longer run time needed to complete the entire process but for the threads and matrix size of 64 it having an negative speedup which took longer run time than the thread size of 128.

6. Conclusion

In conclusion, in order to process a huge amount of data in a short amount of time the parallel processing is required as it is partitioned and distributed equally to all the processors in order to perform the parallel processing in the shortest time. Even though OpenMP, MPI and CUDA will not fit every case perfectly, by understanding the difference between them could definitely be helpful and know more well which is the best to perform better in some sort of situation. In our case, the CUDA had the fastest run time compared to the other two which are OpenMP and MPI. As, if the target was to get the shortest runtime, CUDA would be the choice since it is designed to handle very big matrices with minimal runtime. Next, the result shown above shows that OpenMP is faster than MPI as both using the same threads and the same matrix size. This is because OpenMP uses shared memory which is faster than the distributed memory that is used by MPI. There are many different ways to implement parallel processing even though all of them are able to utilize parallel processing, but the way they are implemented into the code would affect the runtimes.

7. Reference

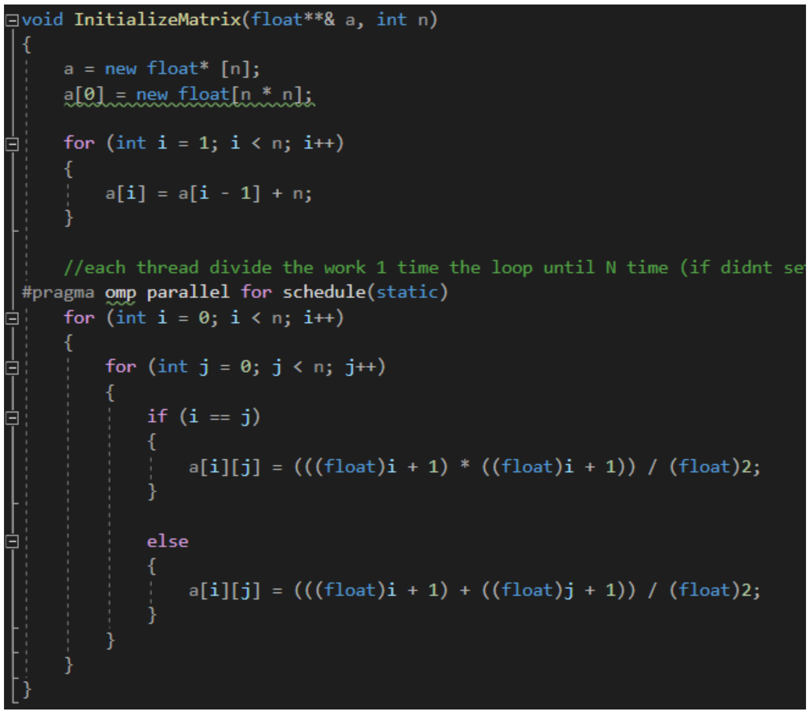
1. Chegg, 2022. LU Decomposition Definition. [online] Available at: <https://www.chegg.com/homework-help/definitions/lu-decomposition-33> [Accessed 21 August 2022]
2. Jaka's Corner, 2016. OpenMP: For & Scheduling. Available at: <http://jakascorner.com/blog/2016/06/omp-for-scheduling.html> [Accessed 18 September 2022]
3. 2018. OPENMP API Specification. Available at: <https://www.openmp.org/spec-html/5.0/openmpse31.html#:~:text=An%20OpenMP%20lock%20can%20be,said%20to%20own%20the%20lock.> [Accessed 18 September 2022]
4. Wes Kendall, 2020. MPI Broadcast and Collective Communication. Available at: <https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/ > [Accessed 18 September 2022]
5. Wes Kendall, 2020. MPI Send and Receive. Available at: <https://mpitutorial.com/tutorials/mpi-send-and-receive/ > [Accessed 18 September 2022]
6. Maxim Milakov, 2015. GPU Pro Tip: Fast Dynamic Indexing of Private Arrays in CUDA. Available at: <https://developer.nvidia.com/blog/fast-dynamic-indexing-private-arrays-cuda/ > [Accessed 18 September 2022]
7. Cyril Zeller, 2011. CUDA C/C++ Basics. Available at: <https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf > [Accessed 18 September 2022]
8. En.wikipedia.org. n.d. *LU decomposition - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/LU_decomposition> [Accessed 19 August 2022].
9. West, M. and Erin, C., 2017. *LU Decomposition for Solving Linear Equations - CS 357*. [online] Courses.physics.illinois.edu. Available at: <https://courses.physics.illinois.edu/cs357/sp2020/notes/ref-9-linsys.html> [Accessed 19 August 2022].
10. Anas, B., 2020. *Investigating the Use of Pipelined LU Decomposition to Solve Systems of Linear Equations*. [online] Ieeexplore-ieee-org.tarcez.tarc.edu.my. Available at: <https://ieeexplore-ieee-org.tarcez.tarc.edu.my/document/9213785> [Accessed 19 August 2022].
11. Math.ucr.edu. 2022. Analysis and Implementation of Parallel LU-Decomposition with Different Data Layouts. [online] Available at: <https://math.ucr.edu/~muralee/LU-Decomposition.pdf> [Accessed June 2000].
12. Tildesites.bowdoin.edu. 2022. Intro to Parallel Programming with OpenMP. [online] Available at: <https://tildesites.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall16/Lectures/openmp.html#:~:text=OpenMP%20in%20a%20nutshell,threads%20share%20memory%20and%20data> [Accessed 18 August 2022].

13. Ieeexplore-ieee-org.tarcez.tarc.edu.my. 2022. Implementing Parallel LU Factorization with Pipelining on a MultiCore Using OpenMP. [online] Available at: <https://ieeexplore-ieee-org.tarcez.tarc.edu.my/document/5692483> [Accessed 11 January 2011].
14. Condor.cc.ku.edu. 2022. Introduction to the Message Passing Interface (MPI) using C. [online] Available at: <http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml> [Accessed 18 August 2022].
15. Ieeexplore-ieee-org.tarcez.tarc.edu.my. 2022. Implementation of parallel convolution based on MPI. [online] Available at: <https://ieeexplore-ieee-org.tarcez.tarc.edu.my/document/6967057> [Accessed 1 December 2014].
16. Ozcan, C. and Sen, B., 2022. Investigation of the performance of LU decomposition method using CUDA. [online] Available at: <https://www.sciencedirect.com/science/article/pii/S2212017312000126> [Accessed 11 February 2012].
17. GeeksforGeeks. 2022. Introduction to CUDA Programming - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/introduction-to-cuda-programming/#:~:text=CUDA%20is%20a%20programming%20language,while%20providing%20well%2Dformed%20speed.> [Accessed 16 August 2021].

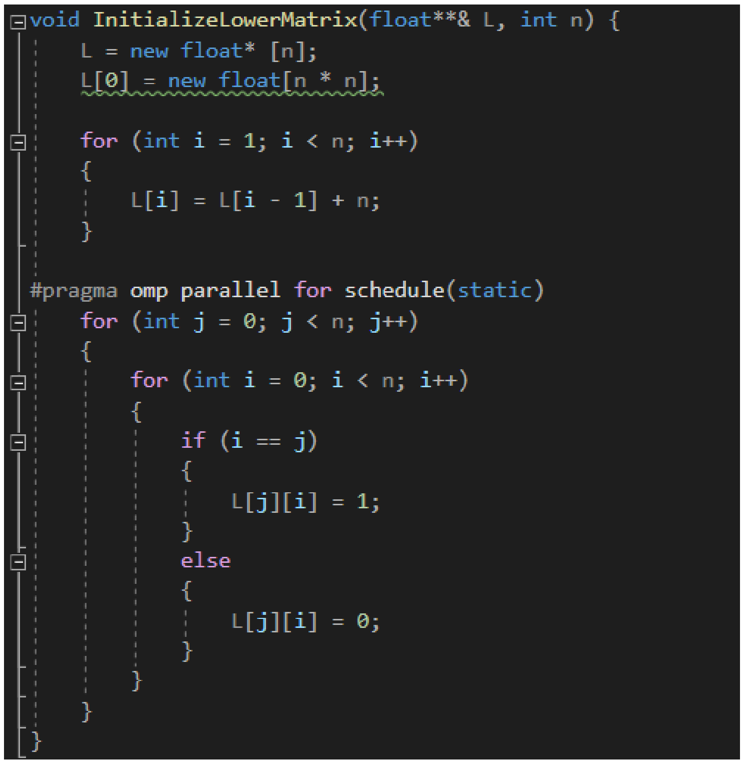
18. Parallel Patternlets, 2022. Barrier Synchronization. [online] Available at: <http://selkie.macalester.edu/csinparallel/modules/Patternlets/build/html/MessagePassing/Barrier_Tags.html> [Accessed 16 August 2021].
19. GeeksforGeeks, 2022. Doolittle Algorithm : LU Decomposition. [online] Available at: <https://www.geeksforgeeks.org/doolittle-algorithm-lu-decomposition/#:~:text=Doolittle's%20method%20provides%20an%20alternative,of%20L%20and%20U%20explicitly>
20. Javatpoint, 2022. Doolittle Algorithm : LU Decomposition. [online] Available at: <https://www.javatpoint.com/doolittle-algorithm-lu-decomposition> [Accessed 21 August 2022].
21. Chegg, 2022. LU Decomposition Definition. [online] Available at: <https://www.chegg.com/homework-help/definitions/lu-decomposition-33> [Accessed 21 August 2022].

8. Appendix

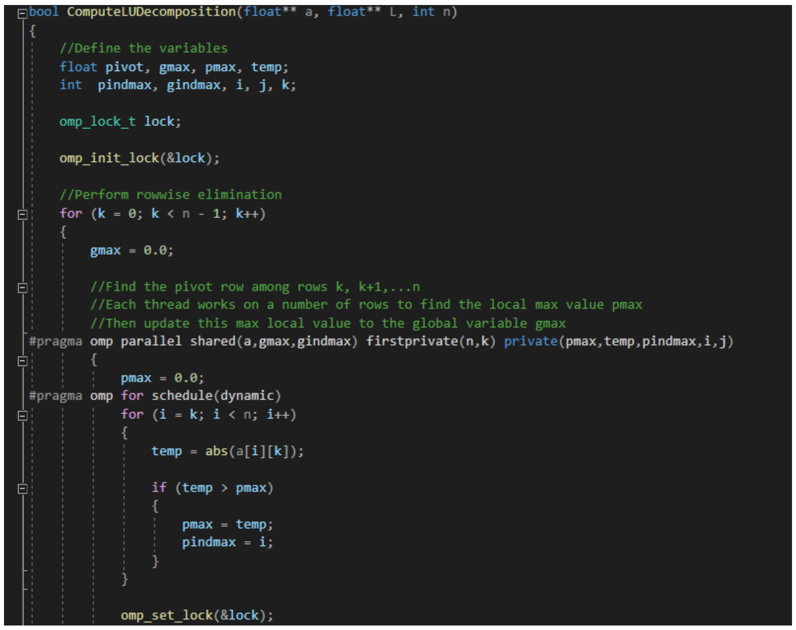
8.1 OpenMP



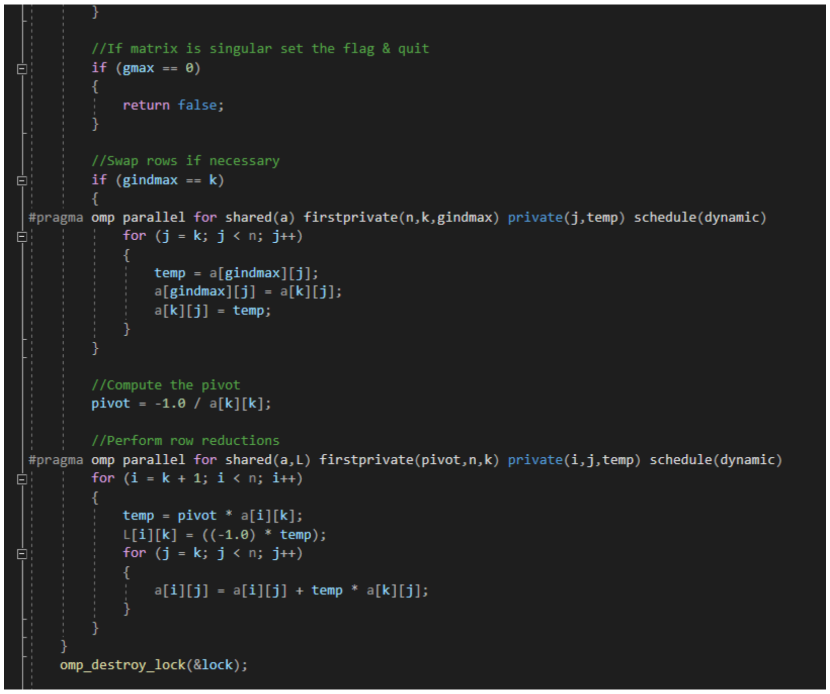
```
void InitializeMatrix(float**& a, int n)
{
    a = new float* [n];
    a[0] = new float[n * n];

    for (int i = 1; i < n; i++)
    {
        a[i] = a[i - 1] + n;
    }

    //each thread divide the work 1 time the loop until N time (if didnt se
#pragma omp parallel for schedule(static)
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (i == j)
            {
                a[i][j] = (((float)i + 1) * ((float)i + 1)) / (float)2;
            }

            else
            {
                a[i][j] = (((float)i + 1) + ((float)j + 1)) / (float)2;
            }
        }
    }
}
```



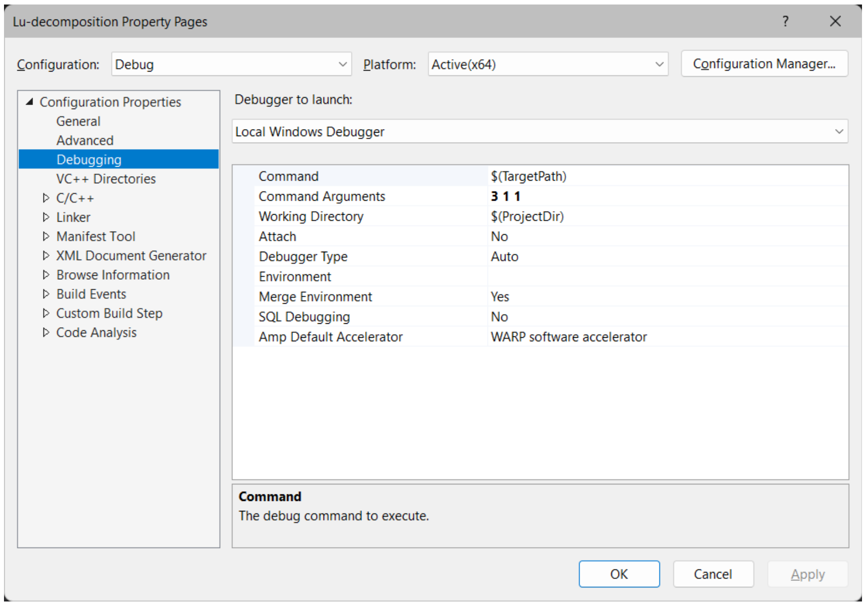
```
void InitializeLowerMatrix(float**& L, int n) {
    L = new float* [n];
    L[0] = new float[n * n];

    for (int i = 1; i < n; i++)
    {
        L[i] = L[i - 1] + n;
    }

#pragma omp parallel for schedule(static)
    for (int j = 0; j < n; j++)
    {
        for (int i = 0; i < n; i++)
        {
            if (i == j)
            {
                L[j][i] = 1;
            }
            else
            {
                L[j][i] = 0;
            }
        }
    }
}
```



```
bool ComputeLUDecomposition(float** a, float** L, int n)
{
    //Define the variables
    float pivot, gmax, pmax, temp;
    int  pindmax, gindmax, i, j, k;

    omp_lock_t lock;

    omp_init_lock(&lock);

    //Perform rowwise elimination
    for (k = 0; k < n - 1; k++)
    {
        gmax = 0.0;

        //Find the pivot row among rows k, k+1,...n
        //Each thread works on a number of rows to find the local max value pmax
        //Then update this max local value to the global variable gmax
#pragma omp parallel shared(a,gmax,gindmax) firstprivate(n,k) private(pmax,temp,pindmax,i,j)
        {
            pmax = 0.0;
#pragma omp for schedule(dynamic)
            for (i = k; i < n; i++)
            {
                temp = abs(a[i][k]);

                if (temp > pmax)
                {
                    pmax = temp;
                    pindmax = i;
                }
            }

            omp_set_lock(&lock);
```



```
        }

        //If matrix is singular set the flag & quit
        if (gmax == 0)
        {
            return false;
        }

        //Swap rows if necessary
        if (gindmax == k)
        {
#pragma omp parallel for shared(a) firstprivate(n,k,gindmax) private(j,temp) schedule(dynamic)
            for (j = k; j < n; j++)
            {
                temp = a[gindmax][j];
                a[gindmax][j] = a[k][j];
                a[k][j] = temp;
            }
        }

        //Compute the pivot
        pivot = -1.0 / a[k][k];

        //Perform row reductions
#pragma omp parallel for shared(a,L) firstprivate(pivot,n,k) private(i,j,temp) schedule(dynamic)
        for (i = k + 1; i < n; i++)
        {
            temp = pivot * a[i][k];
            L[i][k] = ((-1.0) * temp);
            for (j = k; j < n; j++)
            {
                a[i][j] = a[i][j] + temp * a[k][j];
            }
        }
    }
    omp_destroy_lock(&lock);
```



Lu-decomposition Property Pages

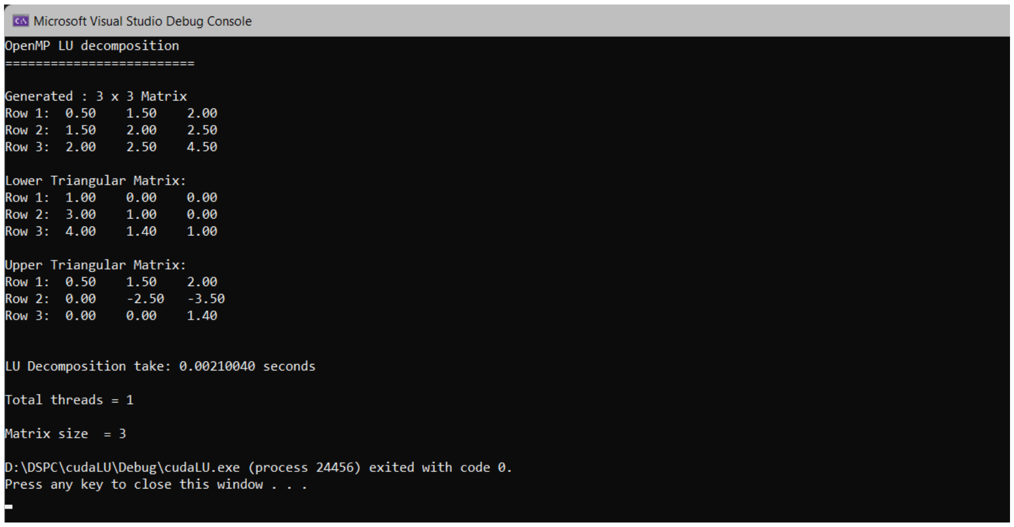
Configuration: Debug   Platform: Active(x64)

Debugger to launch:

Local Windows Debugger

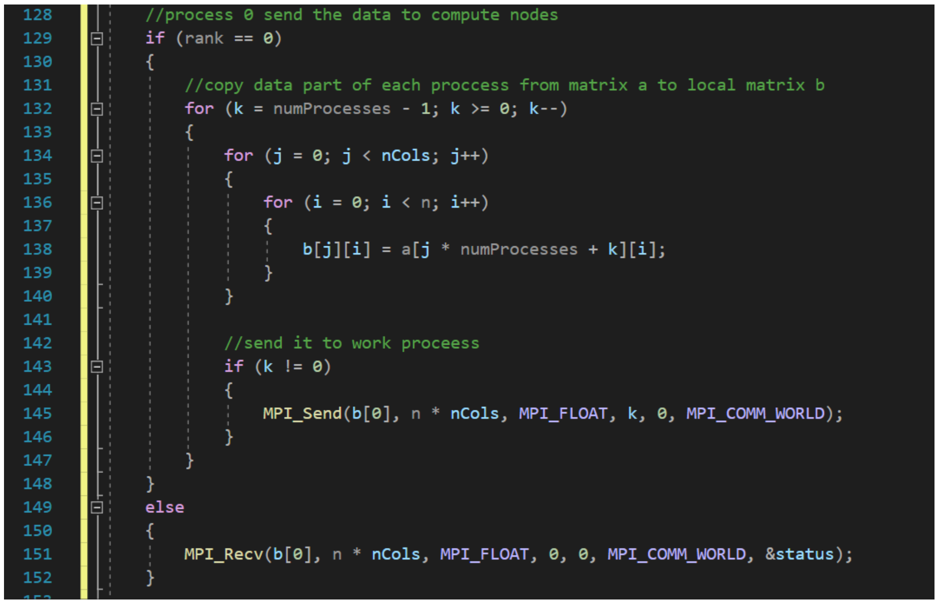
Command	$(TargetPath)
Command Arguments	**3 1 1**
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No
Amp Default Accelerator	WARP software accelerator

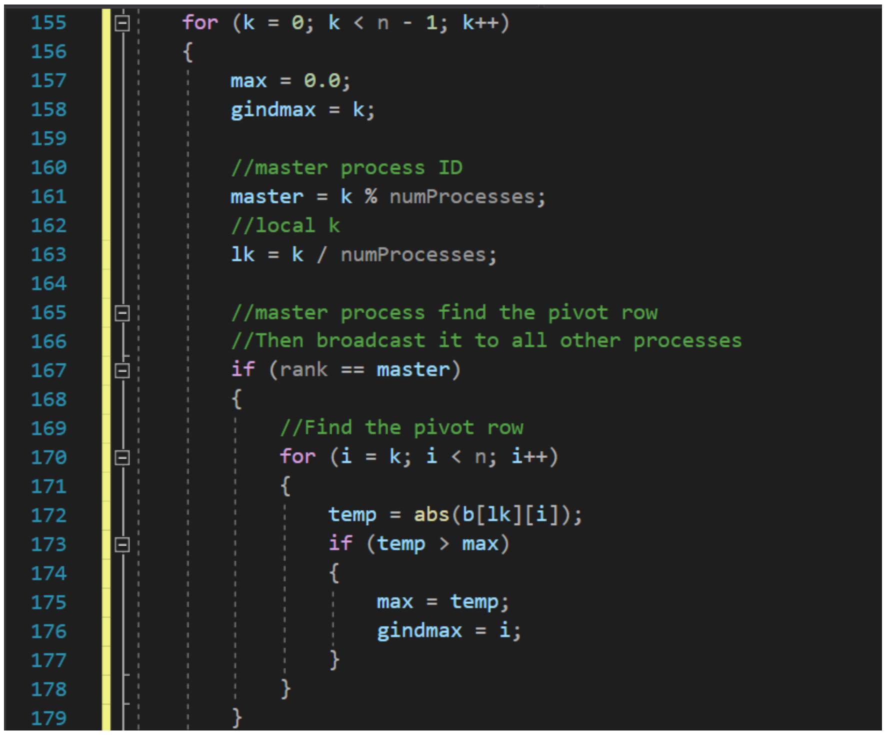
Command
The debug command to execute.


Microsoft Visual Studio Debug Console

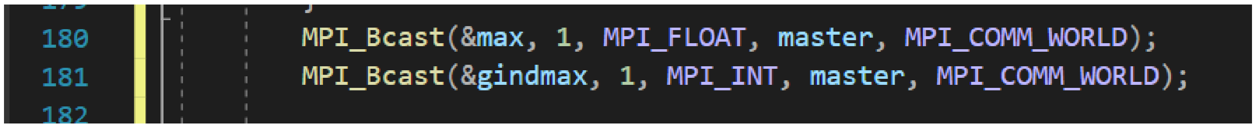
```
OpenMP LU decomposition
=========================

Generated : 3 x 3 Matrix
Row 1:  0.50    1.50    2.00
Row 2:  1.50    2.00    2.50
Row 3:  2.00    2.50    4.50

Lower Triangular Matrix:
Row 1:  1.00    0.00    0.00
Row 2:  3.00    1.00    0.00
Row 3:  4.00    1.40    1.00

Upper Triangular Matrix:
Row 1:  0.50    1.50    2.00
Row 2:  0.00    -2.50   -3.50
Row 3:  0.00    0.00    1.40


LU Decomposition take: 0.00210040 seconds

Total threads = 1

Matrix size  = 3

D:\DSPC\cudaLU\Debug\cudaLU.exe (process 24456) exited with code 0.
Press any key to close this window . . .
```

8.2 MPI

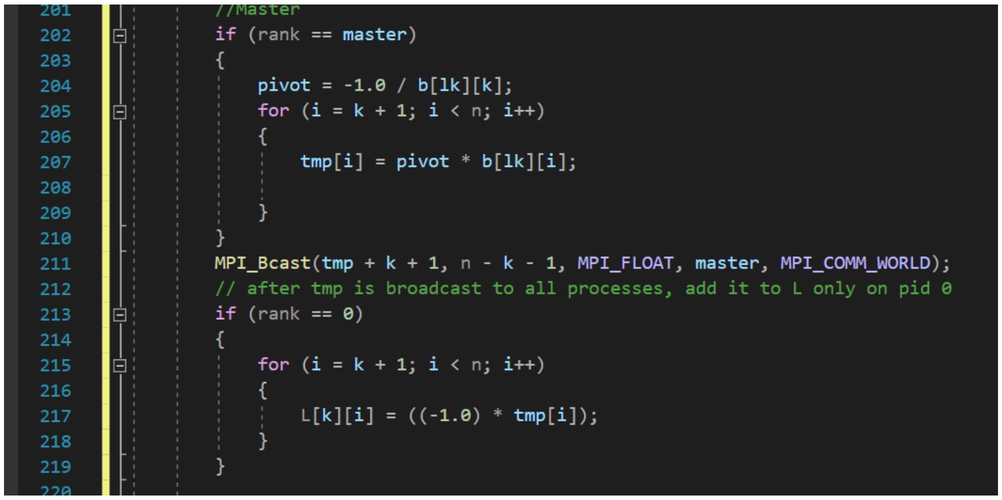

```
//process 0 send the data to compute nodes
if (rank == 0)
{
    //copy data part of each proccess from matrix a to local matrix b
    for (k = numProcesses - 1; k >= 0; k--)
    {
        for (j = 0; j < nCols; j++)
        {
            for (i = 0; i < n; i++)
            {
                b[j][i] = a[j * numProcesses + k][i];
            }
        }

        //send it to work proceess
        if (k != 0)
        {
            MPI_Send(b[0], n * nCols, MPI_FLOAT, k, 0, MPI_COMM_WORLD);
        }
    }
}
else
{
    MPI_Recv(b[0], n * nCols, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
}
```



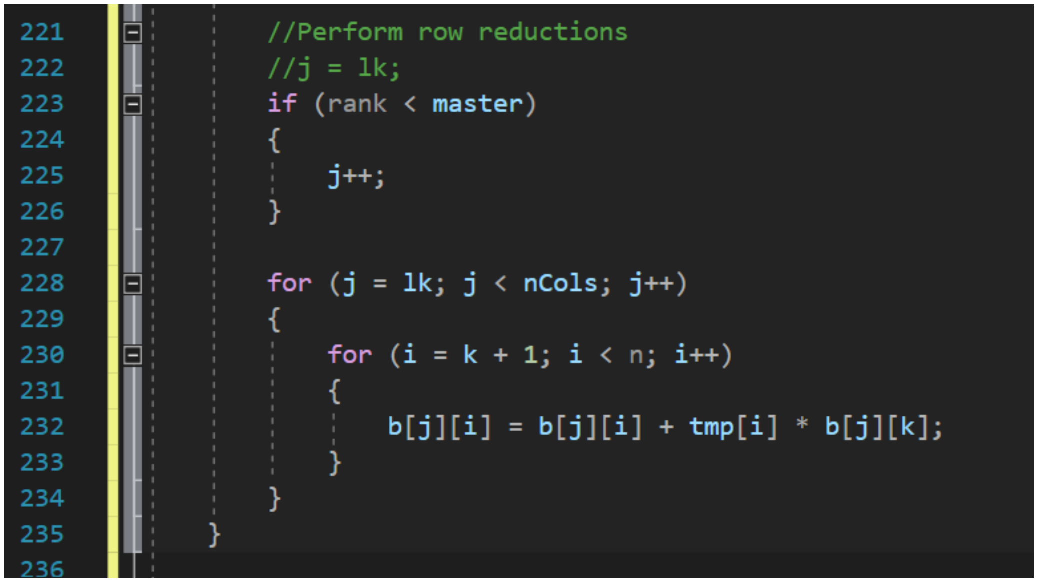
```
for (k = 0; k < n - 1; k++)
{
    max = 0.0;
    gindmax = k;

    //master process ID
    master = k % numProcesses;
    //local k
    lk = k / numProcesses;

    //master process find the pivot row
    //Then broadcast it to all other processes
    if (rank == master)
    {
        //Find the pivot row
        for (i = k; i < n; i++)
        {
            temp = abs(b[lk][i]);
            if (temp > max)
            {
                max = temp;
                gindmax = i;
            }
        }
    }
```



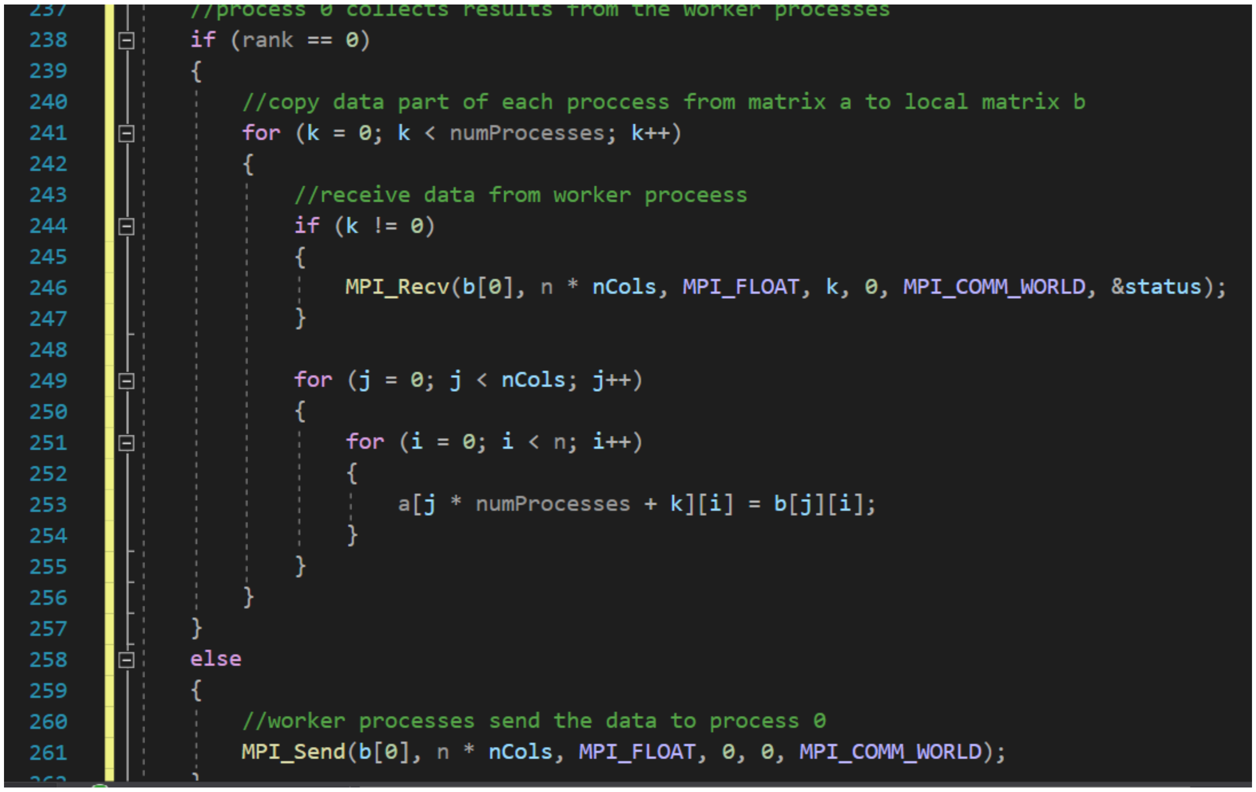
```
    MPI_Bcast(&max, 1, MPI_FLOAT, master, MPI_COMM_WORLD);
    MPI_Bcast(&gindmax, 1, MPI_INT, master, MPI_COMM_WORLD);
```



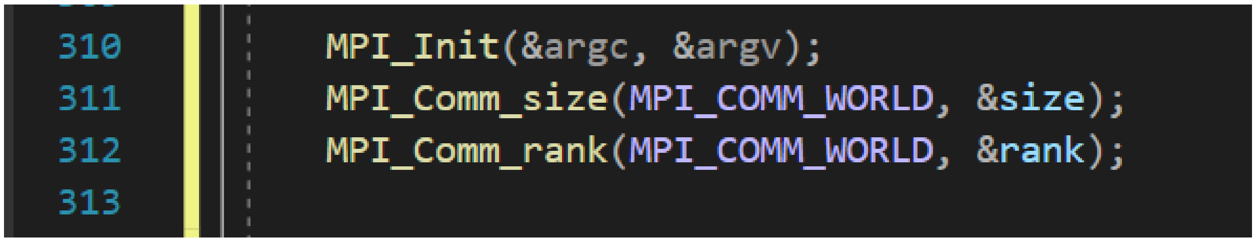
```
    //Master
    if (rank == master)
    {
        pivot = -1.0 / b[lk][k];
        for (i = k + 1; i < n; i++)
        {
            tmp[i] = pivot * b[lk][i];

        }
    }
    MPI_Bcast(tmp + k + 1, n - k - 1, MPI_FLOAT, master, MPI_COMM_WORLD);
    // after tmp is broadcast to all processes, add it to L only on pid 0
    if (rank == 0)
    {
        for (i = k + 1; i < n; i++)
        {
            L[k][i] = ((-1.0) * tmp[i]);
        }
    }
```



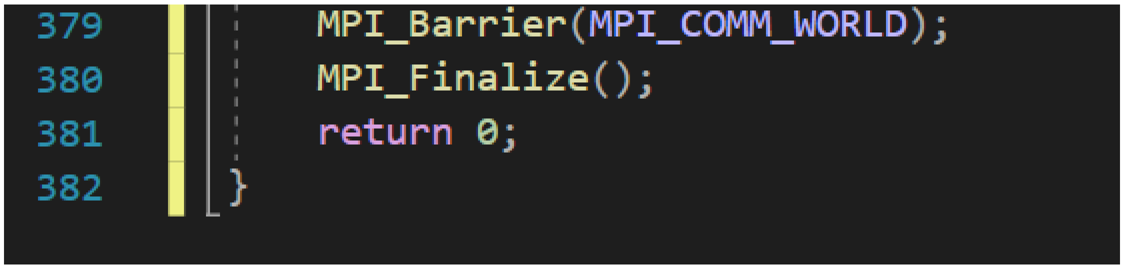
```
        //Perform row reductions
        //j = lk;
        if (rank < master)
        {
            j++;
        }

        for (j = lk; j < nCols; j++)
        {
            for (i = k + 1; i < n; i++)
            {
                b[j][i] = b[j][i] + tmp[i] * b[j][k];
            }
        }
    }
```



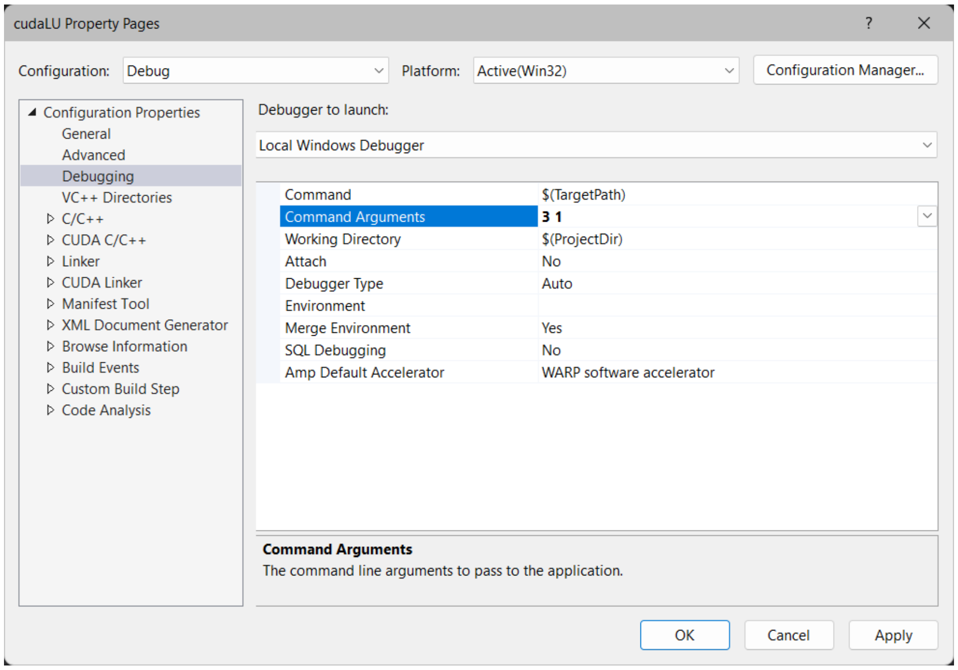
```
    //process 0 collects results from the worker processes
    if (rank == 0)
    {
        //copy data part of each proccess from matrix a to local matrix b
        for (k = 0; k < numProcesses; k++)
        {
            //receive data from worker proceess
            if (k != 0)
            {
                MPI_Recv(b[0], n * nCols, MPI_FLOAT, k, 0, MPI_COMM_WORLD, &status);
            }

            for (j = 0; j < nCols; j++)
            {
                for (i = 0; i < n; i++)
                {
                    a[j * numProcesses + k][i] = b[j][i];
                }
            }
        }
    }
    else
    {
        //worker processes send the data to process 0
        MPI_Send(b[0], n * nCols, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
```



```
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```



```
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
        return 0;
}
```



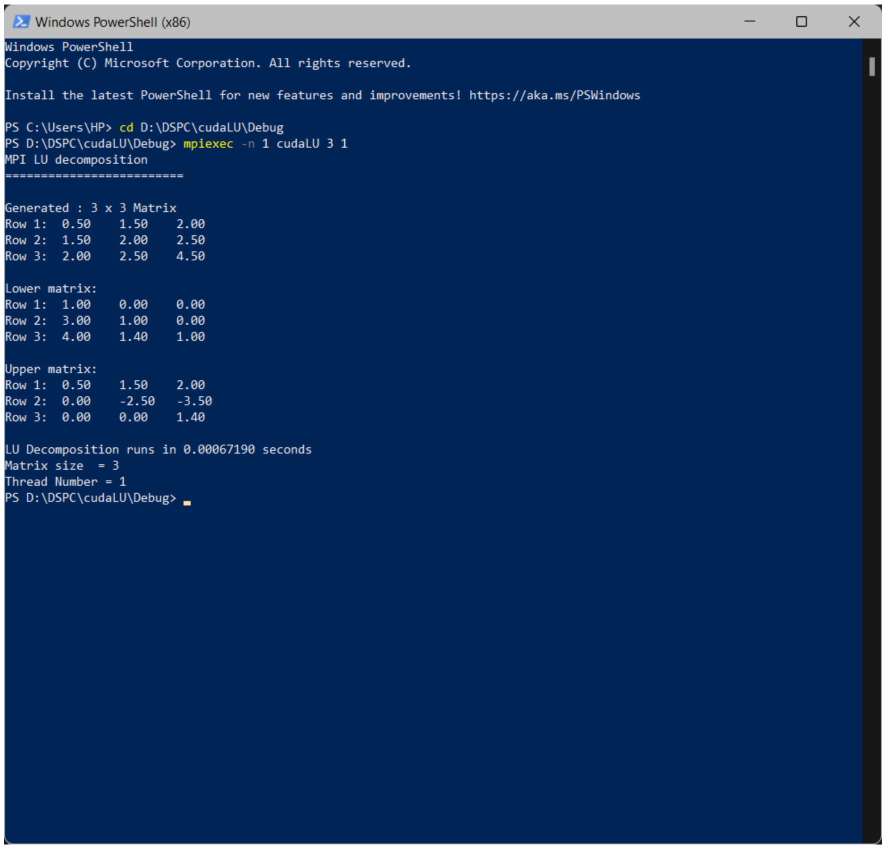
cudaLU Property Pages

Configuration: Debug   Platform: Active(Win32)

Debugger to launch: Local Windows Debugger

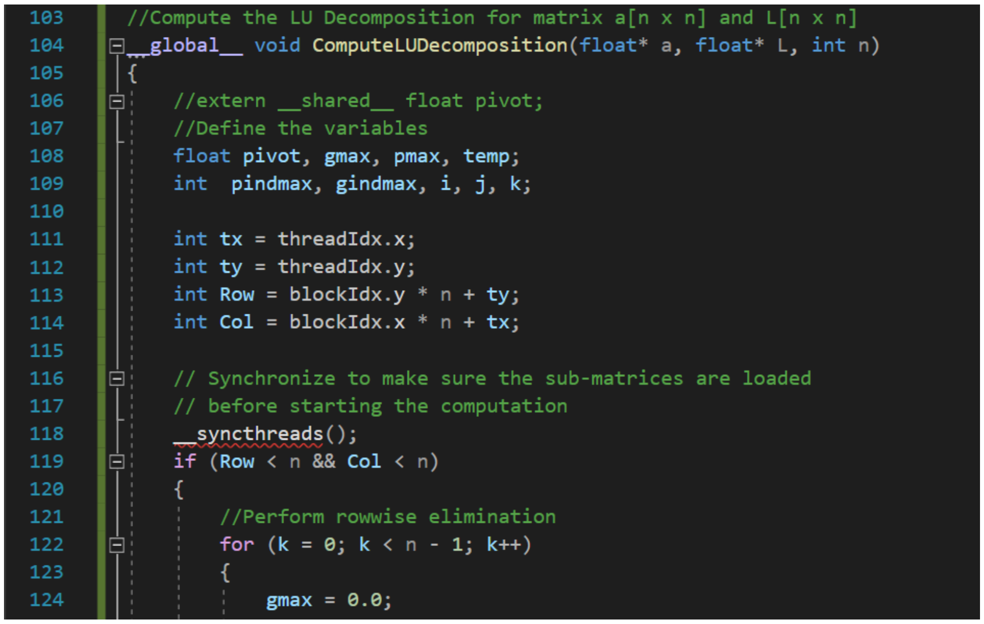
Command	$(TargetPath)
Command Arguments	3 1
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No
Amp Default Accelerator	WARP software accelerator

Command Arguments
The command line arguments to pass to the application.

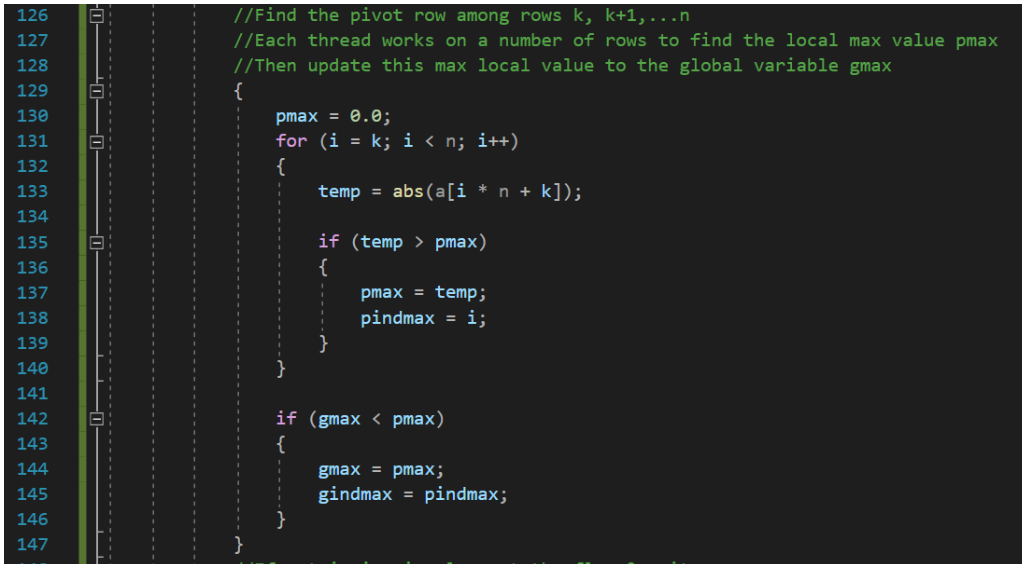

Windows PowerShell (x86)

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\HP> cd D:\DSPC\cudaLU\Debug
PS D:\DSPC\cudaLU\Debug> mpiexec -n 1 cudaLU 3 1
MPI LU decomposition
========================

Generated : 3 x 3 Matrix
Row 1:  0.50    1.50    2.00
Row 2:  1.50    2.00    2.50
Row 3:  2.00    2.50    4.50

Lower matrix:
Row 1:  1.00    0.00    0.00
Row 2:  3.00    1.00    0.00
Row 3:  4.00    1.40    1.00

Upper matrix:
Row 1:  0.50    1.50    2.00
Row 2:  0.00    -2.50   -3.50
Row 3:  0.00    0.00    1.40

LU Decomposition runs in 0.00067190 seconds
Matrix size  = 3
Thread Number = 1
PS D:\DSPC\cudaLU\Debug>
```

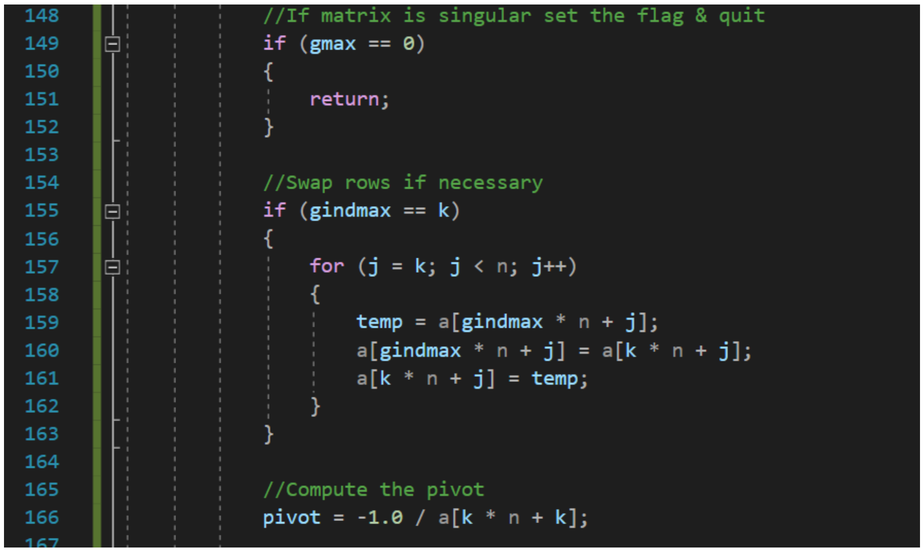
8.3 Cuda



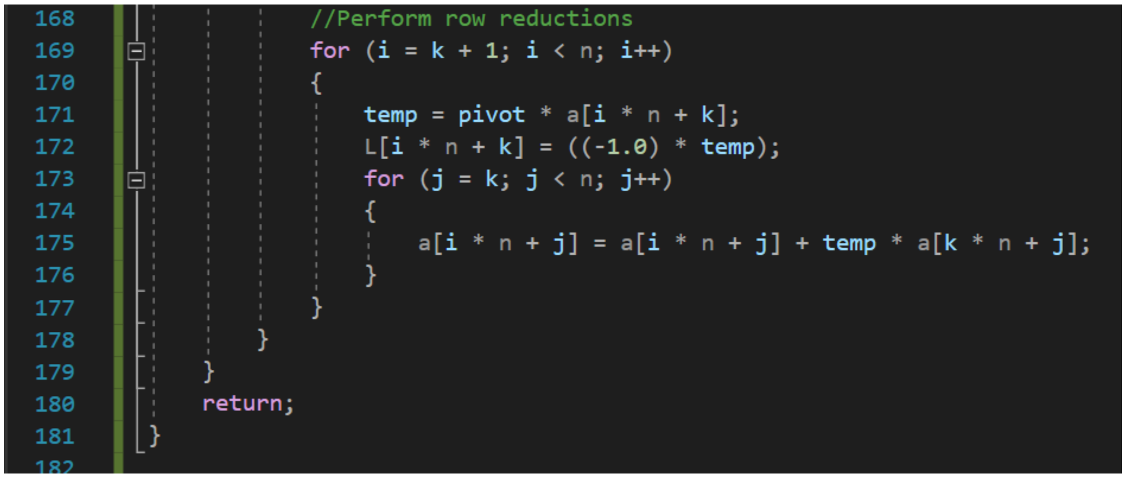
```
//Compute the LU Decomposition for matrix a[n x n] and L[n x n]
__global__ void ComputeLUDecomposition(float* a, float* L, int n)
{
    //extern __shared__ float pivot;
    //Define the variables
    float pivot, gmax, pmax, temp;
    int  pindmax, gindmax, i, j, k;

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int Row = blockIdx.y * n + ty;
    int Col = blockIdx.x * n + tx;

    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();
    if (Row < n && Col < n)
    {
        //Perform rowwise elimination
        for (k = 0; k < n - 1; k++)
        {
            gmax = 0.0;
```



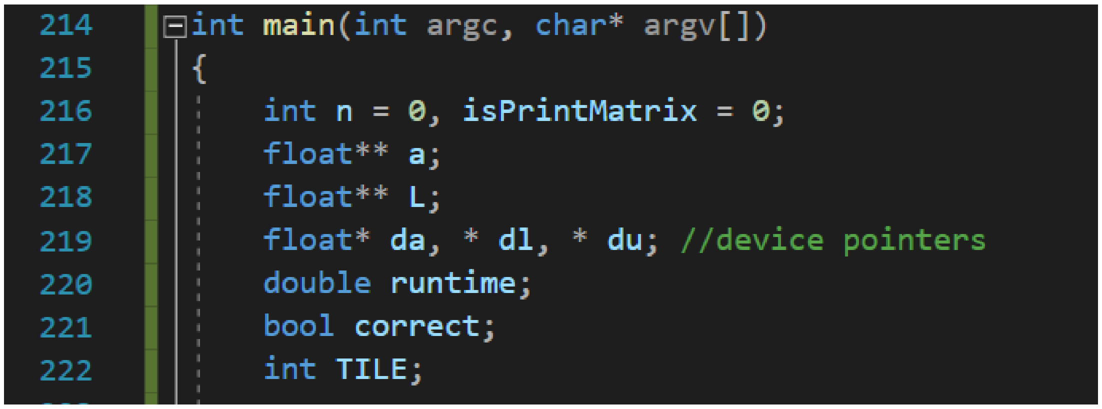
```
            //Find the pivot row among rows k, k+1,...n
            //Each thread works on a number of rows to find the local max value pmax
            //Then update this max local value to the global variable gmax
            {
                pmax = 0.0;
                for (i = k; i < n; i++)
                {
                    temp = abs(a[i * n + k]);

                    if (temp > pmax)
                    {
                        pmax = temp;
                        pindmax = i;
                    }
                }

                if (gmax < pmax)
                {
                    gmax = pmax;
                    gindmax = pindmax;
                }
            }
```



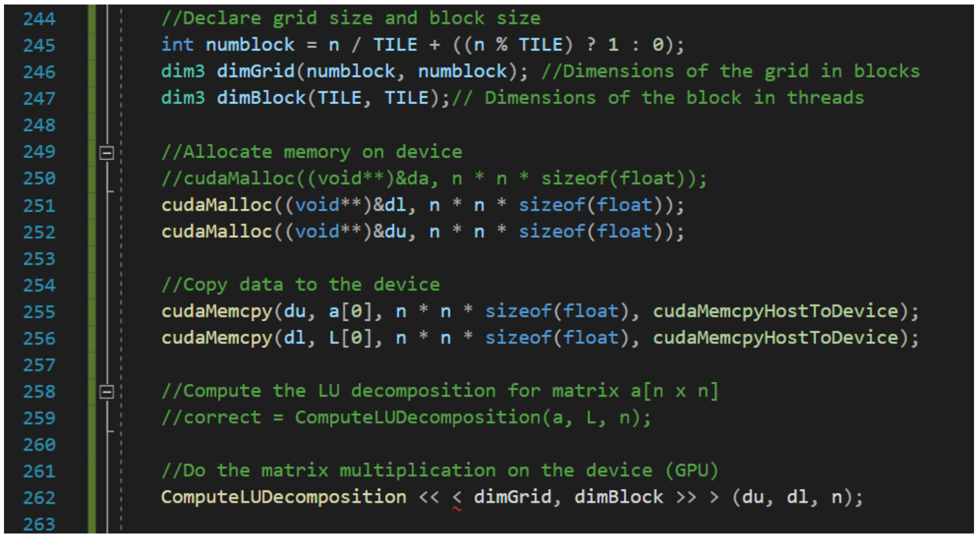
```
            //If matrix is singular set the flag & quit
            if (gmax == 0)
            {
                return;
            }

            //Swap rows if necessary
            if (gindmax == k)
            {
                for (j = k; j < n; j++)
                {
                    temp = a[gindmax * n + j];
                    a[gindmax * n + j] = a[k * n + j];
                    a[k * n + j] = temp;
                }
            }

            //Compute the pivot
            pivot = -1.0 / a[k * n + k];
```



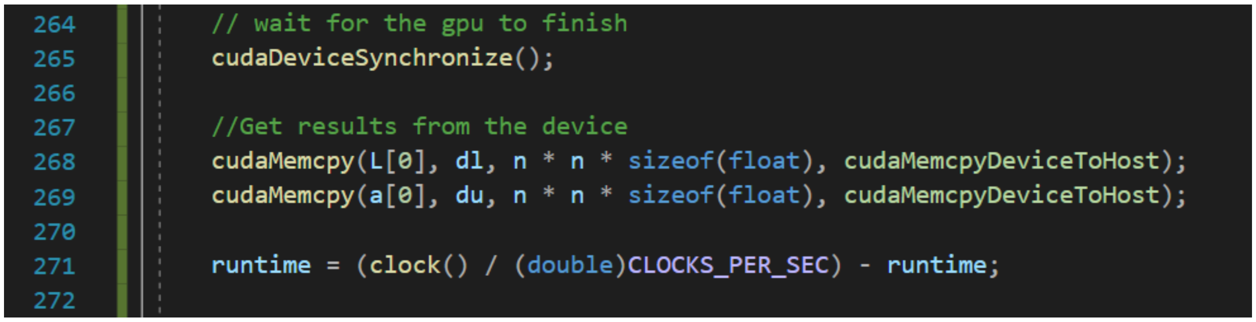
```
            //Perform row reductions
            for (i = k + 1; i < n; i++)
            {
                temp = pivot * a[i * n + k];
                L[i * n + k] = ((-1.0) * temp);
                for (j = k; j < n; j++)
                {
                    a[i * n + j] = a[i * n + j] + temp * a[k * n + j];
                }
            }
        }
    }
    return;
}
```



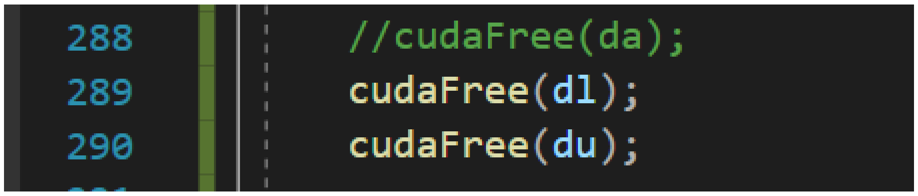
```
int main(int argc, char* argv[])
{
    int n = 0, isPrintMatrix = 0;
    float** a;
    float** L;
    float* da, * dl, * du; //device pointers
    double runtime;
    bool correct;
    int TILE;
```



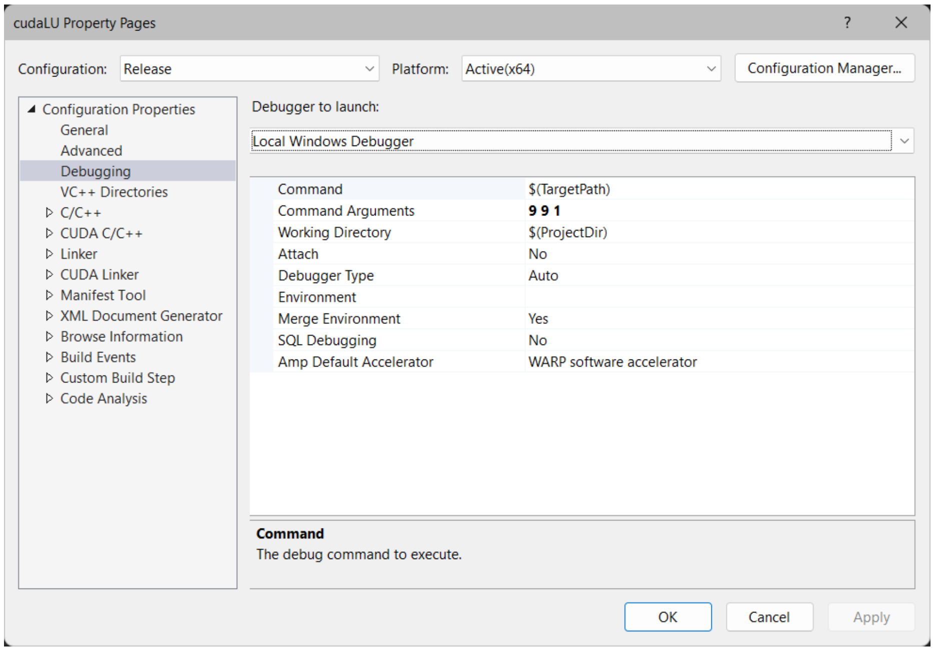
```
    //Declare grid size and block size
    int numblock = n / TILE + ((n % TILE) ? 1 : 0);
    dim3 dimGrid(numblock, numblock); //Dimensions of the grid in blocks
    dim3 dimBlock(TILE, TILE);// Dimensions of the block in threads

    //Allocate memory on device
    //cudaMalloc((void**)&da, n * n * sizeof(float));
    cudaMalloc((void**)&dl, n * n * sizeof(float));
    cudaMalloc((void**)&du, n * n * sizeof(float));

    //Copy data to the device
    cudaMemcpy(du, a[0], n * n * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dl, L[0], n * n * sizeof(float), cudaMemcpyHostToDevice);

    //Compute the LU decomposition for matrix a[n x n]
    //correct = ComputeLUDecomposition(a, L, n);

    //Do the matrix multiplication on the device (GPU)
    ComputeLUDecomposition << < dimGrid, dimBlock >> > (du, dl, n);

```



```
    // wait for the gpu to finish
    cudaDeviceSynchronize();

    //Get results from the device
    cudaMemcpy(L[0], dl, n * n * sizeof(float), cudaMemcpyDeviceToHost);
    cudaMemcpy(a[0], du, n * n * sizeof(float), cudaMemcpyDeviceToHost);

    runtime = (clock() / (double)CLOCKS_PER_SEC) - runtime;

```



```
    //cudaFree(da);
    cudaFree(dl);
    cudaFree(du);
```

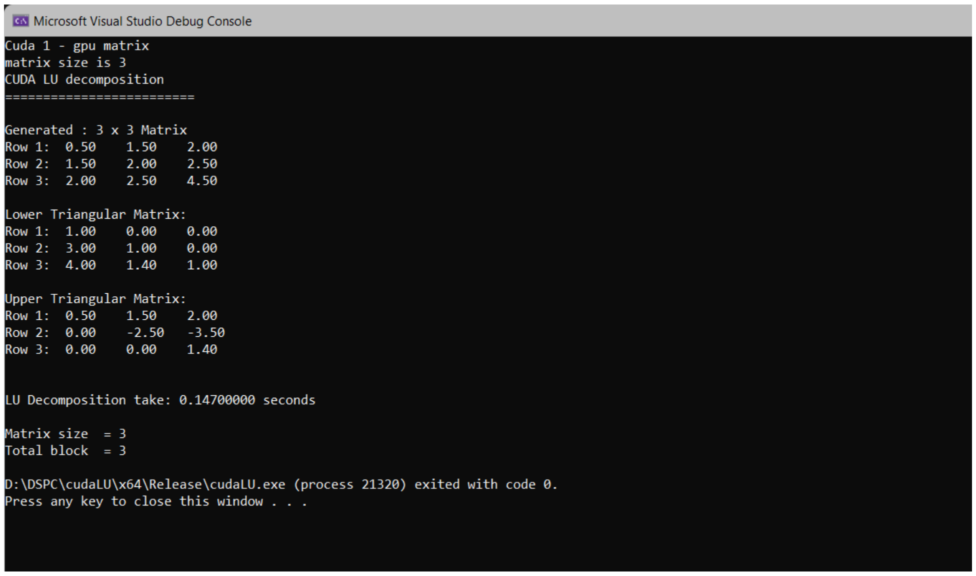


Configuration: Release   Platform: Active(x64)

Debugger to launch:
Local Windows Debugger

Command	$(TargetPath)
Command Arguments	**9 9 1**
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No
Amp Default Accelerator	WARP software accelerator

Command
The debug command to execute.



```
Cuda 1 - gpu matrix
matrix size is 3
CUDA LU decomposition
=========================

Generated : 3 x 3 Matrix
Row 1:  0.50    1.50    2.00
Row 2:  1.50    2.00    2.50
Row 3:  2.00    2.50    4.50

Lower Triangular Matrix:
Row 1:  1.00    0.00    0.00
Row 2:  3.00    1.00    0.00
Row 3:  4.00    1.40    1.00

Upper Triangular Matrix:
Row 1:  0.50    1.50    2.00
Row 2:  0.00    -2.50   -3.50
Row 3:  0.00    0.00    1.40


LU Decomposition take: 0.14700000 seconds

Matrix size  = 3
Total block  = 3

D:\DSPC\cudaLU\x64\Release\cudaLU.exe (process 21320) exited with code 0.
Press any key to close this window . . .
```