**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 | : | Hiew Long Shun / 20WMR09458 |
| Student's name/ ID Number | : | Chua Yee Chen / 20WMR09448 |
| Student's name/ ID Number | : | Cheah Su Ying/ 20WMR09430 |
| Programme | : | Bachelor of Computer Science (Honours) in Software Engineering |
| Tutorial Group | : | 4 |
| Date of Submission to Tutor | : | 19 - September - 2021 |

## Table of Content

# Parallel Process For Linear Algebra: LU Decomposition

Hiew Long Shun, Chua Yee Chen, Cheah Su Ying

## Abstract

*In the era of mathematics, Linear equations have become increasingly popular in the physical world, science, and our everyday applications for describing the relationships and processes between two variables, making predictions, calculating rates, making conversions, etc. Image reconstruction is one of the scientific areas that uses a system of linear equations to form a matrix and can be solved by using Lower-upper (LU) decomposition. However, the complexity of O(N3) computation time for LU decomposition will cause its performance to degrade and even be worse if the matrix is extremely large. This article attempts to parallelize the LU decomposition by using OpenMP, CUDA, and MPI to improve the performance and the computational speed of the LU decomposition. As a result of the implementation, OpenMP and CUDA had shown some success while MPI performs poorly than serial execution because of the distributed memory issue. When comparing OpenMP and CUDA, OpenMP performs better in smaller matrix size while CUDA performs better in larger size because of the usage of GPU and CPU.*

## 1 Introduction

According to the proposal, parallelization is required to enhance the processing performance of LU decomposition. The method proposed includes OpenMP with shared variables, loop construct and lock, Nvidia CUDA with 1D threads, and MPI with broadcasting. In LU decomposition, there are 2 regions for parallelization, the lower triangular matrix (L) and upper triangular matrix (U). In L, the value will be calculated column by column where U will be calculated row by row. Each of the columns and rows will depend on the previously calculated L and U values excluding the first column as shown in Figure 1.1. Even though there is a slight loop-carried dependency among the region, we attempt to allow the calculation of L and U to run concurrently by performing calculations in a block manner. The sequential formula is similar to the diagram below, take note that due to how the loop is structured, there will be minimal, if not none, loop-carried dependencies that require critical section or waits.

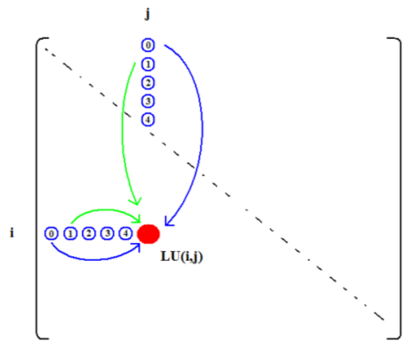


Figure 1.1: Crout's algorithms dependency (Virk 2010)

Each value on the matrix is generated from the random number within the range of 1 to 10. However, to ensure the L and U result is stable in the respect to the growth of round-off errors, a unique solution of L and U is produced without row and column interchanges by ensuring the matrix is strictly diagonal dominant (Fan 2016). In this case, the formula used to ensure a strictly diagonal dominant matrix is through gathering all the sum values of the particular column excluding the diagonal value of the particular column and sum with a random value ranging from 1 to 5. All of the methods suggested will apply with the random seed 1 when generating the matrix value to ensure more integrity and fair results.

This article is focusing on finding the best way to compute the LU decomposition. Each of the solutions will run for 5 different matrix sizes which are 64x64, 128x128, 256x256, 512x512, 1024x1024 and 2048x2048. The OpenMP is run for threads 2, 4, 8, 16, and 32 while the MPI is run for threads 2, 4, and 8, and CUDA is run for threads 32, 64, 128, 256, 512 and 1024. The computed result ( L and U) will then be validated by multiplying the L and U values together where it will form back the original matrix.

Each parallelization method will then select its best solution for comparison regardless of the number of threads for a fair comparison. The comparison among best solutions from each method is tested on the same testing environment which is Intel i5-8250U with 1.60GHz, 4 cores, 8 logical processors, and NVIDIA GeForce MX150.

## 2 Improvement of Chosen Method

The serial code proposed is unchanged from the proposal. From the serial code, 3 parallelism implementations are proposed for each of the methods.The main focus on our code will be parallelising the *void l_u_d(float** a, float** l, float** u, int size)* function that actually performs LU decomposition. The rest of the functions (such as printing) are not parallelized because they highly depend on implementation, and some are even non-parallelizable without causing out-of-order issues (e.g. sequential printing).

## 2.1 OpenMP

### 2.1.1 Memory Management

Firstly, the research applied memory management knowledge to implement shared data. In this case, we will be doing data sharing of the variable *a*, *l*, and *u*. These are the variables and their respective functions:

Table 2.1: List of shared variables

| Variable | Type | Function |
|---|---|---|
| a | float[][](2D float array) | Stores the full un-decomposed matrix |
| l | | Stores the l(ower) matrix from the decomposition |
| u | | Stores the u(pper) matrix from the decomposition |

The main reason for doing so is because firstly, *a* is a constant matrix value and thus it will waste memory by making them private, allocating, and copying them for each thread. Since all the operations do not modify *a*, there is no reason to do so. For the *l* and *u* 2D matrices, the matrix access is in a The second reason is because *l* and *u* are accessed by the line “l[j][i] = l[j][i] - l[j][k] * u[k][i];”. Therefore, the program needs access to the latest value available. This implementation is necessary for all of the 3 solutions. Due to the lack of conflicting dependencies (as long as nowait is not used), this does not actually affect the accuracy of the results.

### 2.1.2 Locks

Unlike the initial proposal, locks are not necessary in our implementation. Testing revealed that even on very large matrix sizes (1000x1000 and above) as well as on a high number of threads (40+ threads), accurate results are obtained. One reason is due to the "block-like" decomposition, where the LU decomposition first starts from the row and column nearest to the top-left corner, and then moves diagonally down, using the row and column in each position. Therefore, there are no dependencies between the values and the previous values that require critical sections. Thus, lock declarations had been excluded.

### 2.1.3 Solution 1: Instruction-level, loop parallelism (OMP-ILP)

The first solution proposed is instruction-level loop parallelism, or in OpenMP, "for-loop" parallelism (codenamed: OMP-ILP). Loop parallelism is a common type of parallelism in scientific codes, so OpenMP has an easy mechanism for it. OpenMP parallel loops are a first example of OpenMP 'work-sharing' constructs: constructs that take an amount of work and distribute it over the available threads in a parallel region, created with the parallel pragma. (Eijkhout 2020) There are 2 places to be parallelized. The first is the lower matrix, or "L" decomposition loop, and the second uses the upper matrix, or "U" decomposition loop. They are then enclosed in a *parallel for* parallel region enclosing all work-sharing for loops. This is because large parallel regions offer more opportunities for using data in cache and provide a bigger context for compiler optimizations (Satoh, Kusano & Sato 2001). The parallel regions are placed outside rather than inside the loops to avoid construction overheads (Satoh, Kusano & Sato 2001). Refer to Figure 8.2 in Appendix 8.1.1 for implementation.

### 2.1.4 Solution 2: Solution 1 + OpenMP Scheduling (OMP-DS)

The second solution is to add in dynamic scheduling. By default, OpenMP statically assigns loop iterations to threads (Sobral 2017). When the parallel for block is entered, it assigns each thread the set of loop iterations it is to execute (Sobral 2017). Therefore, assumption can be made that the results for static scheduling is simply OMP-ILP. To ensure this is the case, *pragma omp for schedule(static)* was tested, and there is negligible difference compared to no schedule specifier. A static schedule can be non-optimal, however. This is the case when the different iterations take different amounts of time. This is true with the LU decomposition, where:

1. Sometimes, instead of calculation, the program will assign 0 or 1 directly according to the algorithm. This is true for the diagonals.
2. For the inner *k* loop in the calculation branch, the number of loops is dependent on the location of the location in the matrix. The latter cells in the matrix will require more loops compared to previous loops.

Therefore, dynamic scheduling will be implemented and test different chunk sizes. For dynamic scheduling, a chunk size of 1,2,4,8,16, and 32 will be assigned. The chunk size is limited to 32 because that's exactly half the smallest matrix size that we will benchmark, which is 64x64. For the number of threads, we will use the number of threads that provide the best result from solution 1. The results are then documented. Refer to Figure 8.2 in Appendix 8.1.1 for implementation.

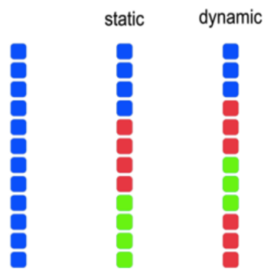


Figure 2.1: Static vs Dynamic Scheduling (Bosio 2017)

### 2.1.5 Solution 3: Solution 2 + data-level, SIMD parallelism (OMP-SIMD)

The third solution is to add in Single Instruction, Multiple Data (SIMD), or better known as data-leveling parallelism (SIMD). The *omp simd* directive is applied to a loop to indicate that multiple iterations of the loop can be executed concurrently by using SIMD instructions (IBM 2018). After declaring parallel-for in the outer loop to split the LU decomposition task to threads for multiple cores in CPU to handle them concurrently, our implementation then adds SIMD parallelism to the inner loop to take advantage of SIMD pipelines *within* individual cores of the CPU. To put it simply, this implementation transforms the individual data elements to an operation where a single instruction operates concurrently on multiple data elements (SIMD). Modern Intel processor cores have dedicated vector units supporting SIMD parallel data processing (Reinders & Jeffers 2016). Intel processors that support Intel® Advanced Vector Extensions (Intel® AVX) have one 256-bit vector unit per core. Thus, each core can process eight single-precision (e.g., 32-bit) floating point operations or four double-precision (e.g., 64-bit) floating point operations using a single instruction (Reinders & Jeffers 2016).

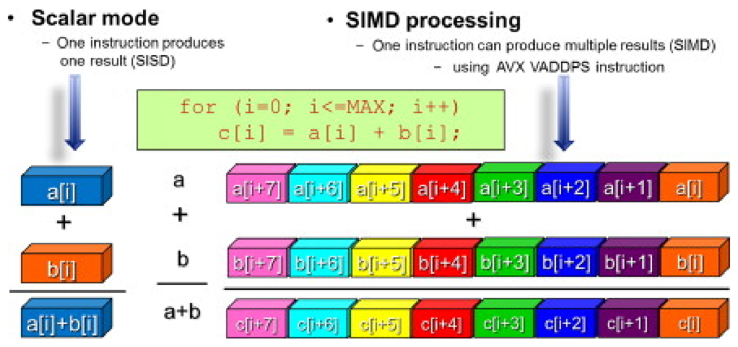

• Scalar mode
– One instruction produces one result (SISD)
• SIMD processing
– One instruction can produce multiple results (SIMD)
– using AVX VADDPS instruction
for (i=0; i<=MAX; i++)
c[i] = a[i] + b[i];
a[i]
+
b[i]
a[i]+b[i]
a
+
b
a+b
a[i+7] a[i+6] a[i+5] a[i+4] a[i+3] a[i+2] a[i+1] a[i]
+
b[i+7] b[i+6] b[i+5] b[i+4] b[i+3] b[i+2] b[i+1] b[i]
c[i+7] c[i+6] c[i+5] c[i+4] c[i+3] c[i+2] c[i+1] c[i]

Figure 2.2: Example of SIMD Processing. Multiple results are produced from 1 instruction. (Reinders & Jeffers 2016)

SIMD will be implemented in the LU decomposition for non-diagonals. The inner loop we have in Figure 8.3 in Appendix 8.1.1 is similar to the given example in the picture above. For testing purposes, the thread size, and scheduling algorithm that nets the best scheduling times from solution OMP-DS will be taken to test SIMD instructions.

## 2.2 Message Passing Interface (MPI)

In the proposal, MPI implementation is only proposed to utilize MPI_Bcast and MPI_Gather methods (Figure 8.8 in Appendix 8.2.1). However, after a thorough review of the MPI parallelization methods, two more ways are also chosen to be implemented for parallelizing the LU decompositions.

### 2.2.1 Solution 1: MPI_Allgather

The first chosen method is the MPI_Allgather method. The MPI_Allgather method is almost identical to the MPI_Gather method. However, by referring to the Figure 2.2, MPI_Allgather is used for many-to-many communication, but MPI_gather is used for many-to-one communication between the processes. This is because the MPI_gather method will gather elements from many processes and order them according to their rank of process from which they were received to one single process: the root process. For the MPI_Allgather method, the elements from many processes are collected but distributed to all the processes simultaneously instead of just gathering into the root process. Thus, by using the MPI_Allgather method, an argument "root" that is used to indicate the root process is no longer needed. Furthermore, the MPI_Bcast and MPI_Gather methods also can be used instead at the same time because the matrix elements do not need to broadcast from root process 0 to other processes again after the computation as they are already distributed during the gathering process (Figure 8.9 in Appendix 8.2.1). Apart from that, due to this project being proposed to solve square matrices, MPI_Allgather is chosen to be used instead of MPI_Allgatherv. This is because the matrix elements gathered from each process computation are the same size.

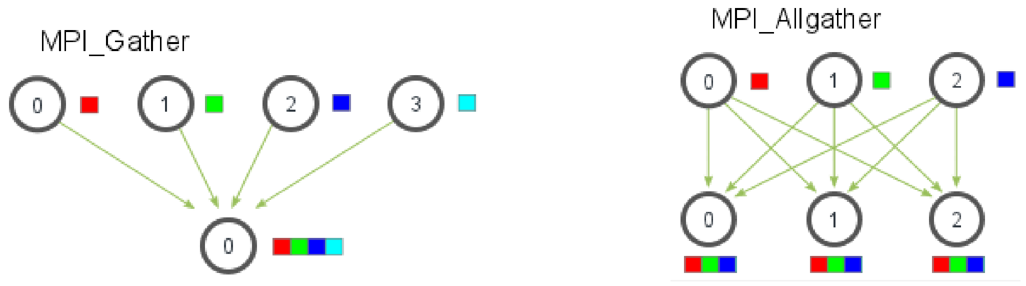


Figure 2.2 : Illustration for MPI_Gather parallelization method and MPI_Allgather parallelization method (Kendall 2019)

### 2.2.2 Solution 2: MPI_Send and MPI_Recv

The second chosen parallelization method is the combination of MPI_Send and MPI_Recv, where they are two foundational concepts for MPI. This method is totally different from the previous methods because previous methods are used for collective communication while this method is used for point-to-point communication as shown in Figure 2.3. Moreover, they operate in pairs for this project because they are used for one-to-one communication among the processes (Kendall 2020). For this method, the MPI_Bcast method is being replaced. The root process 0 will send the matrix after the random generation or computation to other processes one by one through placing the MPI_Send method in a for-loop. While other processes will receive the matrix from process 0 using the MPI_Recv method before processing the computation part (Figure 8.10 in Appendix 8.2.1). For MPI_Send and MPI_Recv, MPI_Barrier plays a crucial role in preventing the MPI_Send routines from continuing. It ensures that all the destination processes receive the message from process 0 before continuing to the following computation part. Thus, the computation process can be synchronized among all the processes and ensure all the processes receive the latest lower and upper matrix results. This is because the Crout algorithm that is used for lu decomposition has a data dependency where each of the columns and rows will depend on the previously calculated L and U values excluding the first column.

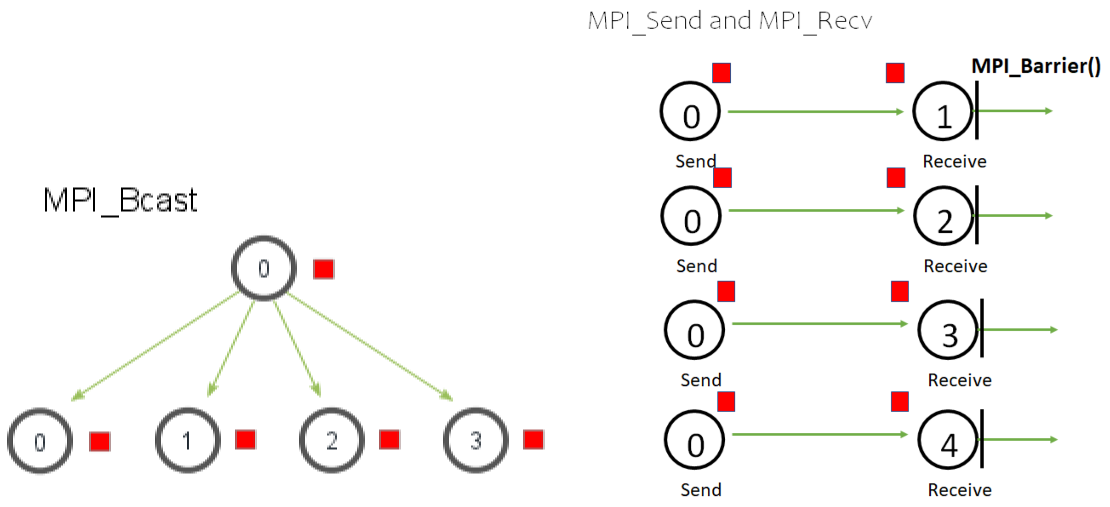


Figure 2.3 : Illustration for MPI_Bcast parallelization method, MPI_Send and MPI_Recv parallelization method (Kendall 2019)

### 2.2.3 Improvement for MPI Implementation

The original, lower, and upper matrices are formed in 2-dimensional arrays based on the sequential code. However, 2D arrays are having problems in working with the MPI communication functions. According to an article posted by Christal Karlsson, MPI implementations are often optimal for the single-dimensional array but only suboptimal for multidimensional arrays (Karlsson 2012). This is because the MPI communication for multidimensional arrays depends heavily on the hardware topology and overall physical configuration. For example, the CPU cores used for multidimensional arrays communication should be physically close to each other in the network grid to shorten communication time. More time and effort also needed to be spent to ensure the well communication between these neighboring cores. Moreover, the MPI communication for multidimensional arrays lacks optimization because each process rank can communicate along the row or column but never the both for the same time. Thus, this problem has a significant impact on performance.

In order to optimize the performance, the matrices have been implemented as one-dimensional arrays with the matrices stored in row-major order instead of two-dimensional arrays. This is because a 1D array offers better memory locality and less allocation and deallocation overhead if the dense matrix is used. Furthermore, 1D arrays also consume less memory than a 2D array. For this project, an equivalent matrix represented as a 2D array would have it's A[i][j] index correspond to A[i * size + j] in this single array implementation (Figure 8.11 and Figure 8.12 in Appendix 8.2.1).

## 2.3 CUDA

In the proposal, the CUDA implementation is proposed to utilize the shared memory. However, due to the limitation of entry-level graphic cards, shared memory is not implementable due to hardware compatibility. Moreover, it is also proposed to store the input matrix in the constant variable as the values will not change. However, there are two prerequisites of constant variable implementation that have been neglected. Firstly, there is only 64KB of constant memory to be used. As the CT scan image's matrix is larger than 64KB, multiple initialization and allocation are required to fulfill the 64KB of usage that will slow down the performance. Secondly, the constant variable is suitable for the unchanged value provided that all the threads are accessing the same constant value simultaneously. In the case of LU decomposition, each of the threads is required to access different values on the matrix. This causes the constant memory to be accessed sequentially where the performance deteriorates and the access time scales linearly according to the number of threads used for reading at a time. (Tutorialspoint, n.d.) Due to these consequences, only global memory will be utilized with local variables used to iterate the calculation for L and U by row or column (Figure 8.16 in Appendix 8.3.1). After The number of parallel threads and the size of the matrix is set, a matrix input with the matrix size will be initialized. The number of parallel threads is set to only accept multiple of 32 to better utilization of threads resources as NVIDIA GPU executes wraps of 32 parallel threads. This is also proved in Figure 8.19 and Figure 8.20 in Appendix 8.3.1 where threads 32 has a shorter execution time compared to threads 33 in computing matrix size 128. After the assessment of serial code, it is found out each calculated value under each row for U or column for L is independent of each other. Hence, parallelize this part in CUDA with each thread reads and writes each of the values where the thread id is mapping with the matrix index as shown in Figure 8.17 and Figure 8.18 in Appendix 8.3.1. Hence, there are two kernels that one calculates for the L value while the other calculates for the U value with the transfer of the initialized matrix from host to the device that will transfer back the calculated L and U value from device to host using cudaMemcpy(). The cudaDeviceSynchronize() in host code is applied to ensure the completion of pending GPU activity for accurate calculation and writing of results.

To optimize the parallelization of LU decomposition in CUDA, different memory optimization strategies have been performed for comparison to implement the best result. There are a total of five solutions provided which are pinned memory, unified memory, zero-copy host code, staged execution and texture memory.

### 2.3.1 Solution 1: Pinned Memory

Pinned memory a.k.a page-locked host memory is allocated using cudaHostAlloc() and the parameter of cudaHostAllocDefault. According to Harris (2012), the allocation of host data is pageable where the GPU cannot access it directly. Hence, it is required for the CUDA driver to be temporarily pinned to the host array by copying the host data to the pinned array as shown in the left hand side of the Figure 2.4. With the use of pinned memory, the transfer between pageable and pinned host arrays can be avoided by directly allocating the host data in the pinned array as shown in the right hand side of the Figure 2.4. With this approach, it is said to be able to enhance the data  transfer rate.

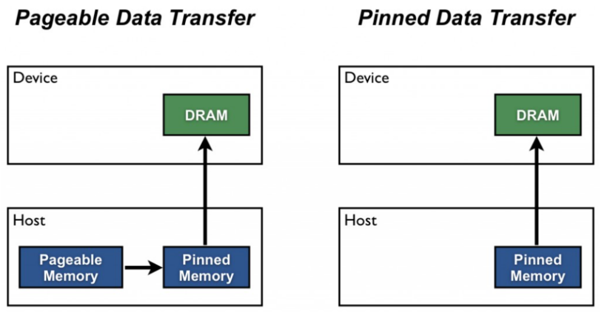


Figure 2.4: Pageable Data Transfer and Pinned Data Transfer (Harris 2012)

### 2.3.2 Solution 2: Unified Memory

Unified memory is a memory allocation technique that allocates a single point where it can be accessed either by the GPU or the CPU on demand as shown in Figure 2.5 through the use of cudaMallocManaged(). The data used will migrate automatically between the host and device where allocation of memory is only performed once without the requirements of copying the data from host to device and vice versa. (Harris 2013) In this project, performance tuning through explicit prefetching on unified memory is also performed using cudaMemPrefetchAsync() with the target GPU or CPU for accessing the data. Explicit memory hints are also provided to advise expected memory access behaviors during runtime through cudaMemAdvise() with the parameter of cudaMemAdviseSetReadMostly as the transferred data is mostly used for reading for calculation. (Crovella 2020)

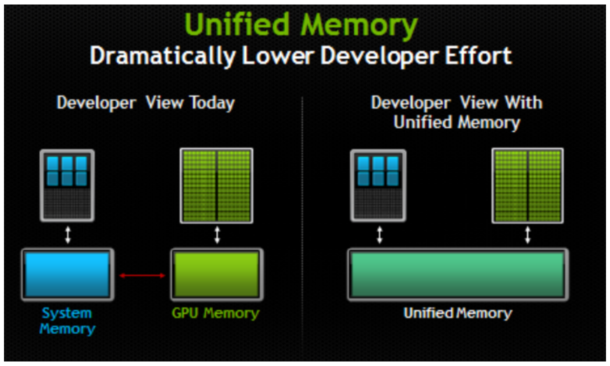


Figure 2.5: Program without and with Unified Memory (Harris 2013)

### 2.3.3 Solution 3: Zero-copy Host Code

The zero-copy host code is similar to the pinned memory where instead of cudaHostAllocDefault, the cudaHostAllocMapped is passed as parameter. In zero-copy host code, it has the same sense with pinned memory where the host memory is pinned. However, there will be no copy of data from host to device where it is accessed through the pointer using cudaHostGetDevicePointer(). Through this function, a valid GPU pointer will be available that it calls to cudaHostAlloc() to return the CPU pointer for the memory. (Sanders & Kandrot 2010)

### 2.3.4 Solution 4: Concurrent Kernel Execution

Concurrent kernel execution is the use of multiple CUDA streams to overlap the kernel execution. Through this process, it allows better utilization of multiprocessors to enhance the execution performance. (NVIDIA Corporation & affiliates 2021) In this case, the concurrent kernel execution is demonstrated with the launching of two streams where each stream is executing one kernel. As the computation of U kernel data depends on the L kernel, the cudaThreadSynchronize is used after execution of the L kernel to ensure the data from the L kernel is ready before the U kernel is executed.

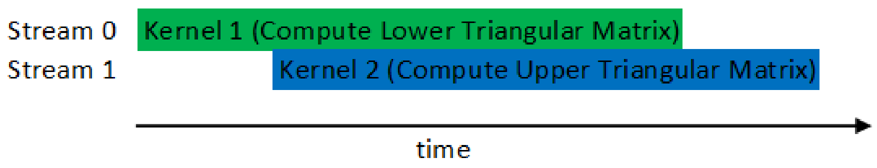

Stream 0
Stream 1
Kernel 1 (Compute Lower Triangular Matrix)
Kernel 2 (Compute Upper Triangular Matrix)
time

Figure 2.6: Concurrent Kernel Execution to compute LU decomposition

### 2.3.5 Solution 5: Texture Memory

According to Sanders & Kandrot (2010), texture memory is cache on the chip as shown in Figure 2.7. Texture memory is one of the read-only memory which can enhance performance by reducing the memory traffic during data reading with certain access patterns. Hence, it is implemented as the input matrix to be read to compute the L and U values. The applied texture memory is 1D texture memory where it is declared using texture<float,1> and the value is retrieved through tex1Dfetch.

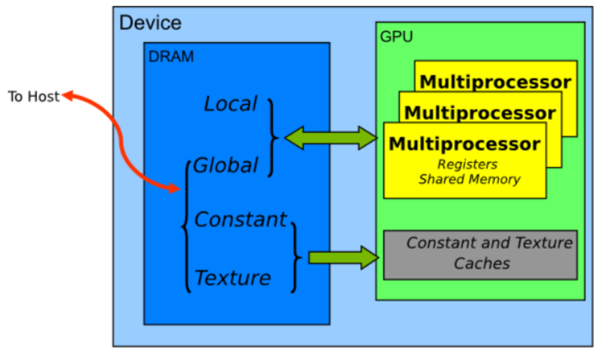


Figure 2.7: Memory spaces on a CUDA device (NVIDIA Corporation & affiliates 2021)

## 3 Results

The full raw averaged results are attached in the Appendix 8.1.3 for OpenMP, 8.2.3 for MPI, and 8.3.3 for CUDA, and “Execution Time”. Firstly, the program is run sequentially from matrix size 64 to 2048 in doubling increments. Afterwards, the same matrix size is used for OpenMP, MPI, and CUDA. For OpenMP benchmarks, the best setups will be cherry-picked for every matrix size. For scheduling, the overall best scheduler is used.

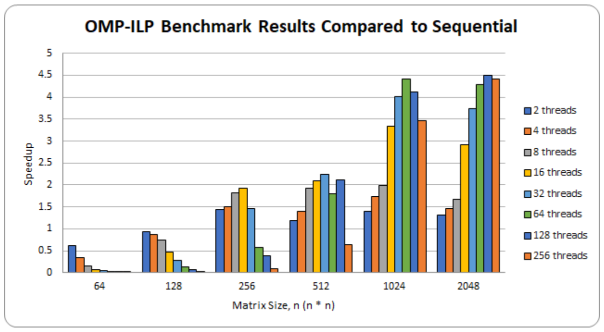
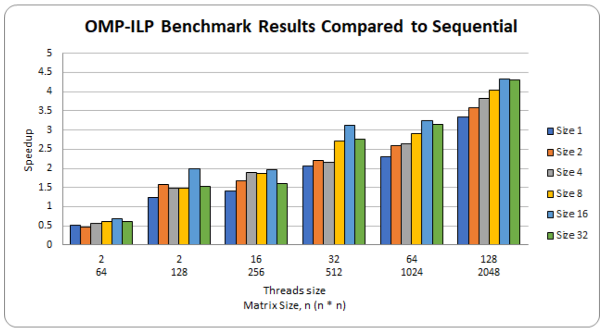
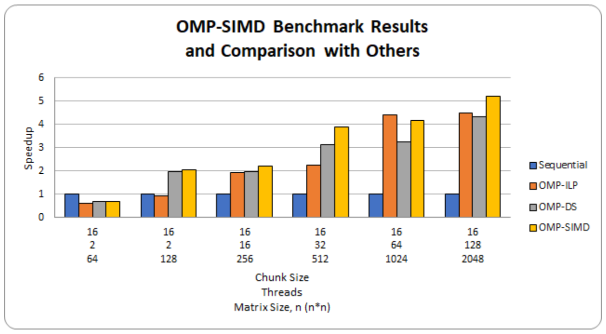
Each method is run 2 times to obtain the average. The averages of all methods in Appendix 8.1.3 for OpenMP, 8.2.3 for MPI, and 8.3.3 for CUDA, are recorded in seconds (s) for clarity and convenience, to allow for cross-comparison. Each method will be run on individual and distinct configurations to find the best method due to the current movement control order causing performance inconsistencies. However, for the final comparison, it will still run on the same computer.

## 3.1 OpenMP

Table 3.1: Testing Environment for OpenMP

| **Test Environment:** | CPU: | Intel(R) Core(TM) i7-4700MQ (2.4GHz - 3.4GHz, 4 cores, 8 threads, 256 KB Level 1 (L1) cache, 1MB Level 2 (L2) Cache, 6MB Level 3 (L3) cache) (Intel 2013) |
|---|---|---|
| | Motherboard: | Paris 5A8 |
| | Graphic Card: | Intel(R) HD Graphics 4600 |
| | RAM: | 5.00 GB |

Table 3.2: Benchmark Results Compared to Sequential for all OpenMP parallelization methods

| Solution 1: OMP-ILP (Figure 8.2, Appendix 8.1) | Solution 2: OMP-DS (Figure 8.3, Appendix 8.1) |
|---|---|
|  |  |
| Solution 3: OMP - SIMD (Figure 8.4, Appendix 8.1) | |
|  | |

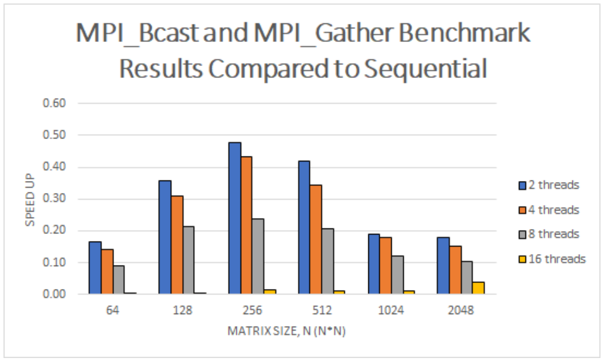
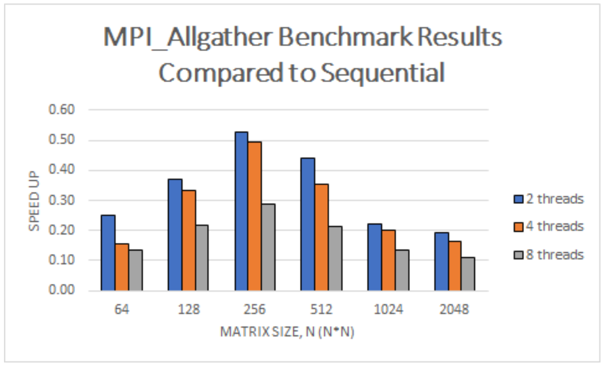
## 3.2 MPI

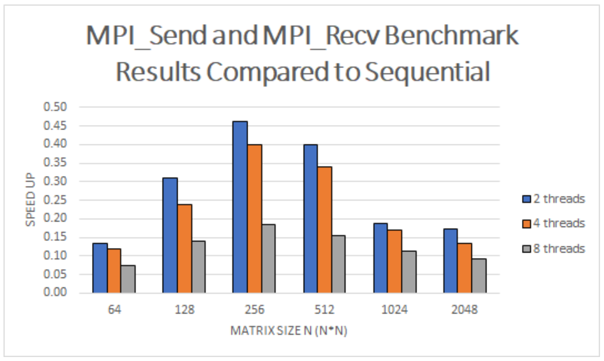
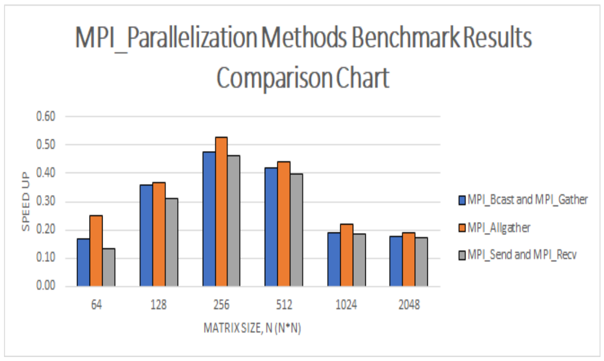
MPI applications generally run parallel computing on multiple processors, which communicate through the library calls. An MPI application is composed of one or more MPI processes, and each of the MPI processes has its local memory and environment. In order to share data explicitly, the processes need to pass the message between each other. The number of MPI processes is determined by mapping onto how many processors. According to Jeff Squyres, the MPI application will run at its peak performance when the number of processors is higher or equal to the number of MPI processes (Squyres 2005). This is because the application fully utilized or underutilized the processors in the operating system. Thus no issues regarding the context switching, cache misses, or virtual memory thrashing will be occurred by other local processes. Based on the number of logical processors for the testing environment is 8, the number of processors (2, 4, 8 and 16) will be implemented to evaluate the application's performance. The Table 3.4 shows the benchmark results for both three MPI_methods implementations on different numbers of processors. 16 processors will only be implemented for one of the methods to prove that overutilization will cause performance degradation.

Table 3.3: Testing Environment for MPI

| | | |
|---|---|---|
| **Test Environment:** | CPU: | Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz (4 Cores, 8 Logical Processors) 192 KB Level 1 (L1) cache, 2MB Level 2 (L2) Cache, 6MB Level 3 (L3) cache) (Hinum 2019) |
| | Motherboard: | Yebisu_IL |
| | Graphic Card: | NVIDIA GeForce MX250 (2GB, GDDR5)<br><br>Intel(R) UHD Graphics 620 |
| | RAM: | 8.00 GB |

Table 3.4: Benchmark Results Compared to Sequential for all MPI parallelization methods

| Solution 1: MPI_Bcast and MPI_Gather<br>(Table 8.11 in Appendix 8.2.4) | Solution 2: MPI_Allgather<br>(Table 8.12 in Appendix 8.2.4) |
|---|---|
|  |  |

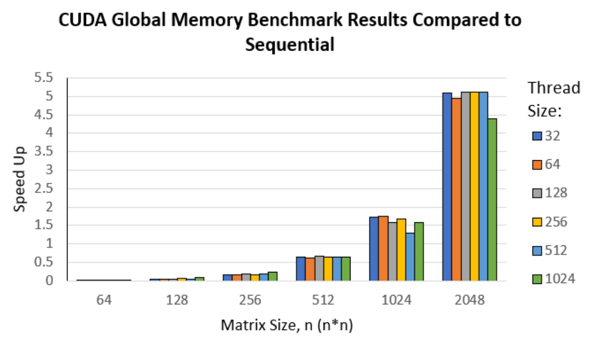
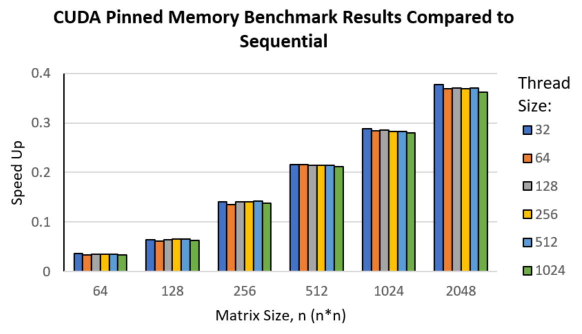
| Solution 3: MPI_Send and MPI_Recv (Table 8.13 in Appendix 8.2.4) | Comparison between three methods for 2 processors (Table 8.14 in Appendix 8.2.4) |
|---|---|
|  |  |

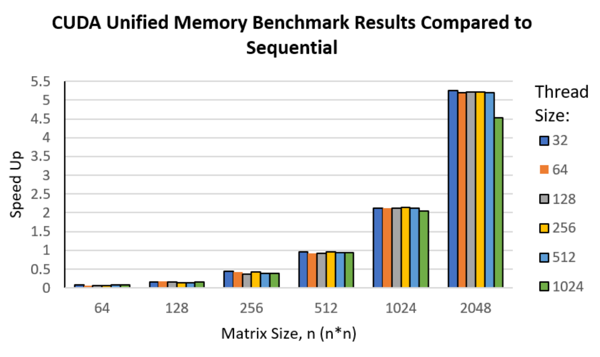
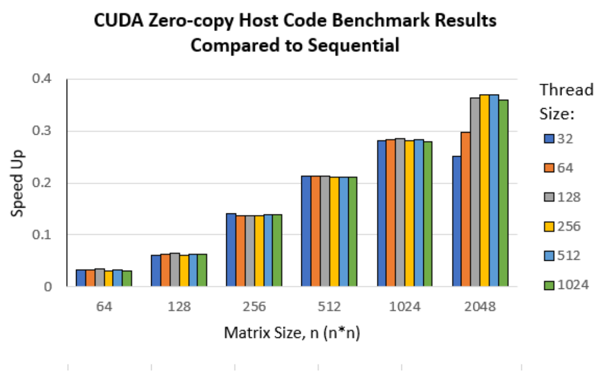
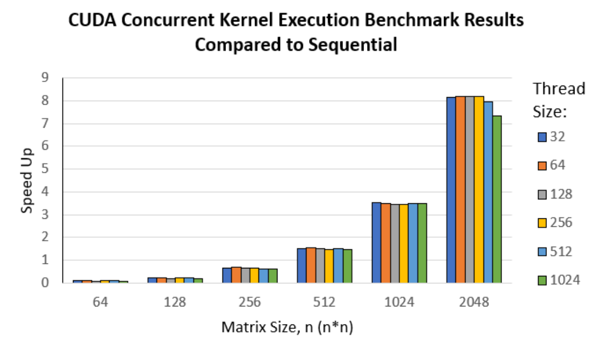
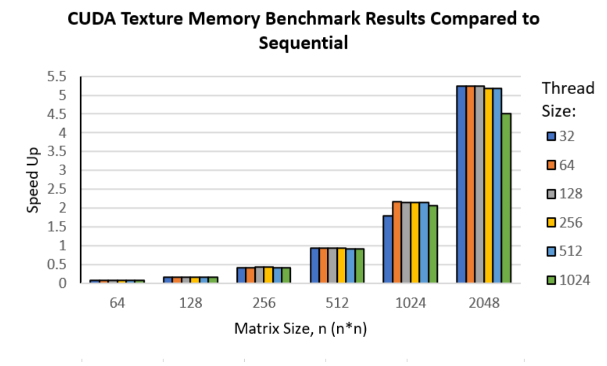
## 3.3 CUDA

Table 3.5: Testing Environment for CUDA

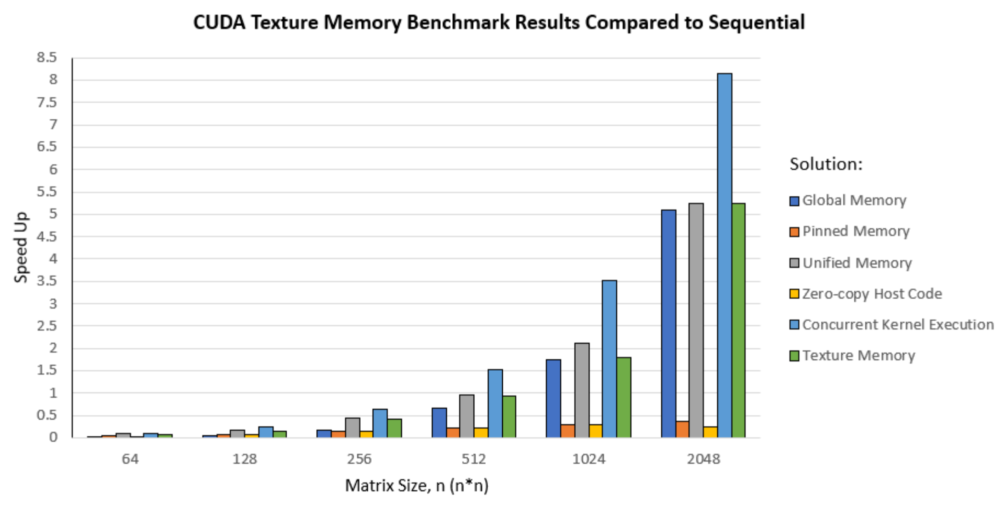
| **Test Environment:** | CPU: | Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz (4 Cores, 8 Logical Processors) 256 KB Level 1 (L1) cache, 1MB Level 2 (L2) Cache, 6MB Level 3 (L3) cache) (Hinum 2017) |
|---|---|---|
| | Motherboard: | LNVNB161216 |
| | Graphic Card: | NVIDIA GeForce MX150 (2GB, GDDR5)<br><br>Intel(R) UHD Graphics 620 |
| | RAM: | 12.00 GB |

Table 3.6: Benchmark Results Compared to Sequential for all CUDA parallelization methods

| Original Solution: Global Memory (Table 8.22 in Appendix 8.3.4) | Solution 1: Pinned Memory (Table 8.23 in Appendix 8.3.4) |
|---|---|
|  |  |

| Solution 2: Unified Memory (Table 8.24 in Appendix 8.3.4) | Solution 3: Zero-copy host code (Table 8.25 in Appendix 8.3.4) |
|---|---|
|  |  |
| Solution 4: Concurrent Kernel Execution (Table 8.26 in Appendix 8.3.4) | Solution 5: Texture memory (Table 8.27 in Appendix 8.3.4) |
|  |  |

Execution Time for global memory implementation and another five solutions with a number of threads of 32 (Table 8.28 in Appendix 8.3.4)



## 4 Analysis & Discussion

### 4.1 Serial Execution

The main reason a sequential algorithm is implemented is to provide a basis for the rest of our parallel algorithm, provide a baseline benchmark, as well as to verify that the answer is correctly computed after parallelization. Firstly, the sequential algorithm is run for 5 times at 1x1, 2x2, 3x3, 4x4, and 5x5 respectively. 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 sequential algorithm. Afterwards, only the 4x4 result is used to verify the accuracy of OpenMP, MPI, and CUDA implementations.

Table 4.1 : Execution Time (in seconds) for serial execution

| Matrix Size | Time (in seconds) |
|---|---|
| 64 | 0.001 |
| 128 | 0.007 |
| 256 | 0.063 |
| 512 | 0.522 |
| 1024 | 4.690 |
| 2048 | 46.216 |

### 4.2 OpenMP

#### 4.2.1 Implementation A - Loop parallelism (OMP-ILP)

The first row first column in Table 3.2 compares the performance of different matrix sizes under different numbers of threads assigned. First of all, speedups below a factor of 1 are considered to be a slowdown. The performance of the benchmarks follow a similar trend to the sequential benchmarks.

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, 2048x2048, as expected, with a speedup of 4.507 times compared to sequential 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 (OSToday 2021). 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 (). The main issue is that switching between the LU decomposition time slices requires saving the register & 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 (CodeGuru 2007). 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 (CodeGuru 2007). Too many software threads will cause cache fighting issues, hurting performance.

Another interesting point is that at size 128x128 and below for the matrix size, negative speedup is observed. The main reason is due to overhead from parallelization. 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. (Asiri 2018)

### 4.2.2 Implementation 2 - OMP-ILP + scheduling (OMP-DS)

The first row second column in Table 3.2 reuses the best matrix size & thread size combination, and varies the chunk size of the dynamic scheduling. The performance of the benchmarks follow a similar trend to the sequential benchmarks.

First of all, we observe a significant performance improvement over OMP-ILP. For matrix size 128x128, speedup was actually observed at a factor of 1.5, rather than negative speedup. Improvements can be seen across the board, with a few exceptions. First, we found out that the optimal chunk size is chunk size 16, with optimal results at every benchmark. However, performance actually degrades compared to OMP-ILP at matrix size 1024x1024 by about a quarter. Further testing may be required to reconfirm.

As expected, dynamic scheduling performs better than static scheduling when the execution time of each thread can vary because there are 2 branches in the processing of each cell in the matrix, and different iterations of calculation depending on the position of the matrix cell. Therefore, the system is able to utilize all the resources available better for the LU Decomposition.

### 4.2.3 Implementation 3 - Implementation 2 + SIMD (OMP-SIMD)

Based on the implementation and the second row in Table 3.2 the speedups obtained by OMP-SIMD are significant as well. According to the results obtained from running on an independent machine, the speedup ranges from 2.1 times at 128x128 to 5.2 times at 2048x2048. This makes OMP-SIMD by far the fastest implementation among all the OpenMP implementations.

The reason the speedup increases when the size of the matrix increases, is because efficient SIMD code requires massive data parallelism, where a sequence of operations is executed for a large number of inputs (Bikker 2017). At size 64, although SIMD gives performance improvements, the amount of data available for data parallelism is simply too small to make a big difference. On the other hand, once the matrix needs to iterate over thousands of values, the improvement becomes significant. talk about fine-grained parallelism

### 4.2.4 Summary for OpenMP

While discussing the OMP-SIMD, we would also want to summarize the results obtained from OpenMP so far. Solution C (OMP-SIMD) is the best performing overall because it took advantage of instruction level parallelism, dynamic scheduling, and SIMD, which is all the optimization that we have implemented. After OMP-SIMD, the runner-up is dependent on the size of the matrix, solution B (OMP-DS) is faster at lower sizes, while solution A (OMP-ILP) is faster at larger sizes. Favorable speedup is observed in all solutions, and therefore, since OMP-SIMD is the best performing, it will be used to compare with the other platform implementations, which are MPI and CUDA.

## 4.3 MPI

Based on the implementation, the MPI application is implementing data parallelism to improve the performance of the LU decomposition. Therefore, the matrices are partitioned and distributed equally among the processors so that each processor only needs to operate a portion of the data. This strategy is helpful as it can let each processor work independently on the parts of data which will increase the computation time. However, the processes also require exchanging the data periodically through communication because the data pieces needed for a process need to keep track to ensure the following calculation process can perform correctly. Therefore, the results showed in the Table 3.4 that two processors have greater speedup in this project than four, eight or sixteen processors because two processors require lesser communication time and thus lesser parallel run time is needed as well.

Based on the results showed in the Table 3.4, it can be seen that collective communication methods (MPI_Allgather, MPI_Bcast, and MPI_Gather) are performed better than the point-to-point communication method (MPI_Send and MPI_Recv) as they have greater speedup. This is because, in point-to-point communication, 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. Collective communication is the alternative way as the message can be sent to multiple receivers simultaneously without looping, which is more effective and easy to use (Cornell Virtual Workshop 2021). According to Alaa Ismail El-Nashar, he stated that the point-to-point pattern is simple to use but it is not suitable to use in large program size or complicated communication structure as it leads to negative performance (Ismail El-Nashar 2011). Thus, the execution time for collective communication methods is significantly shorter than point-to-point communication methods when the matrix size is increased.

By comparing both collective communication methods, MPI_Allgather has lesser execution time and greater speedup compared to MPI_Bcast and MPI_Gather as shown in the second row second column in Table 3.4. This is because, in MPI_Allgather, the message is gathered and distributed among the available processes simultaneously. However, in another method, the root process must collect the message from other processes before the message distribution process. Therefore, the communication time required for MPI_Allgather is lesser than MPI_Bcast and MPI_Gather.

In summary, the MPI_Allgather method that runs in 2 threads performs the best among other MPI parallelization methods as it has lesser execution time and more incredible speedup. MPI_Allgather is also one of the collective communication methods.

## 4.4 CUDA

At a glance on Table 3.6, the thread size of 32 (fine-grained parallel) provides better performance compared to the other thread sizes. Hence, it is selected for the overall comparison among all the solutions. According to NVIDIA Corporation (n.d.), it shows that threads number 32 performed better than threads 64 and 128 when the problem size is small. In this case, the LU decomposition is working with small problem size, hence the kernel size 32 is optimal to utilize. In other words, the fine-grained decomposition is performed when the LU decomposition problem is decomposed into a large number of tasks.

At a glance, most of the solutions are only able to achieve real speedup (speedup with value more than 1) where the parallelization is efficient when the matrix size is 1024x1024 and above. However, the pinned memory and zero-copy host code have inefficient parallelization (speedup with value less than 1) even though they have reached the highest matrix size analyzed (2048x2048). On the other hand, the concurrent kernel execution is efficient in parallelization where it can achieve actual speedup start from matrix size 512x512. The execution time of concurrent kernel execution is also the fastest among all the solutions. The parallelization performs badly in small matrix size because multi GPU approaches require extra operation or overheads for its operation. The overhead includes CPU wrapper overhead, memory overhead and gpu launch overhead. (Wilper, Knight & Cohen 2020)

For small matrix size of 64x64 and 128x128, the global memory is having the worst speedup range of between 0.0129 to 0.0285 and 0.040 to 0.0819 respectively. However, when the matrix size reached 256x256, it outperformed both pinned memory and zero-copy host code. The speedup performance of the pinned memory deteriorates due to the limitation of the testing environment (CPU and chipset) specification and the over-allocation of pinned memory to transfer the large matrix where it causes a trade-off in securing the access to memory (NVIDIA Corporation & affiliates 2021). However, the deterioration of zero-host code memory is due to the over-allocation of memory towards the available total memory (Santos, Eler & Garcia 2016). According to the research, it claimed that zero-copy memory should only be used on data that undergoes less amount of access during execution. Through this, it proves the reason why a zero-copy host is only performed well with a small matrix size.

The unified memory and texture memory have roughly the same speedup for all the matrix sizes, where they are the second most efficient parallelization speedup compared to the other solutions.

For unified memory, it can achieve high performance due to the performance tuning. This is because unified memory intends to simplify the code where there will be some trade-off in the performance. With the performance tuning, it prevents trashing situations by preventing the migration of sparsely accessed pages to save the bandwidth. (Sakharnykh 2017) Hence, improving the execution performance. According to Knap & Czarnul (2019), only the unified memory with performance tuning is performed better compared to explicit memory copying. The unified memory without the performance tuning is said to be performed worse than explicit memory copying.

On the other side, texture memory can perform better compared to global memory even though there is an overhead in creating texture memory compared to global memory. The texture memory is acting as a cache for the global memory hence it can read faster compared to global memory. This results in better performance with the use of texture memory. According to Galoppo et al. (2005), texture memory can cache and provide bandwidth-efficient execution. However, there is some limitation of texture memory where when the matrix size becomes larger the improvement in the processing time becomes insignificant which can be seen through analyzed results where the differences between the global memory execution time and texture memory become closer.

The concurrent kernel execution has the best parallelization efficiency or speedup compared to all the other solutions. Furthermore, the time for cudaMalloc is performed more than a hundred times better than the other approaches as shown in Table 8.33 in Appendix 8.3.5. Even though there is a large amount of overhead in creating multiple streams to launch the kernel as shown in Figure 8.27 in Appendix 8.3.2 where the cudaStreamCreate allocates 118.65 ms with a matrix size of 2048 and 32 threads. The concurrent kernel execution still outperformed the second-fastest execution (unified memory) about 2 times. This is because the kernel is running parallel with 2 different streams. According to Fang et al. (2016), it shows that the overlapped and non-overlapped application has an improvement of around 24% through the implementation of concurrent kernel execution.

Lastly, all the speedup efficiency increases when the matrix size increases. This is because of the efficient utilization of parallel execution when the matrix size is large which makes the overhead negligible.

In summary, the best approach in CUDA is the concurrent kernel execution as it has doubled the performance of the other solutions.

## 4.5 Comparison between All Models

Due to our analysis and discussion is based on benchmark values obtained from testing on different configurations, this section is dedicated to test for the performance using the best implementation for each platform.

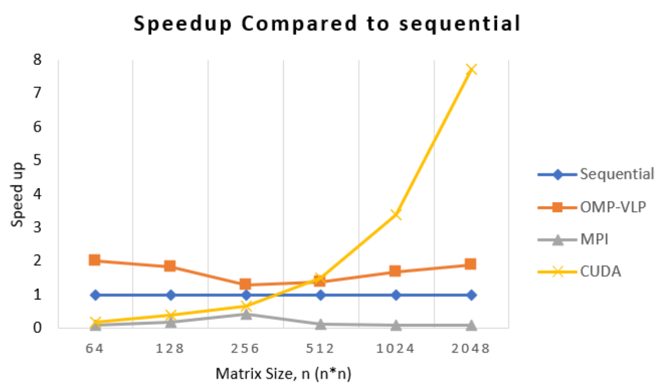


Figure 4.1: Speedup results among all models compared to sequential (Table 8.36 in Appendix 8.4.1)

The performance shown by OMP-SIMD is better than sequential, MPI, and CUDA at lower speed, with a speedup of at least 1.29, and up to matrix size of 2. However, this is quite different from the results obtained in analysis and discussion. One reason is due to the different core speed. Depending on thermals, the platform may only allow up to 1.00GHz on the processor, which is significantly slower than 2.40GHz. However, for smaller matrix sizes, it is surprising to see a positive speedup, and a significant one as well. This is due to greater efficiency of the architecture in newer processors, and the short burst of extra speed provided through Intel Turbo Boost of up to 3.4GHz when not all cores are utilized. Therefore, we should see a performance bump when the computer is not thermally constrained.

Based on Figure 4.1, all the MPI parallelization methods perform poorly than sequential communication, OMP-SIMD and CUDA. The speedups for all the ways are below a factor of 1, which is considered a slowdown. Moreover, it can also be seen that the speedup for the MPI parallel execution starts to degrade gradually when the matrix size is over 256*256. This is because MPI is usually run in parallel across multiple computers connected by the network to carry out distributed computing (Aveuh 2018). Therefore when the testing environment for MPI in this project is just run in parallel in a single computer, the implementations of MPI do not take full advantage of the distributed memory for communication and computation. According to an article published in ResearchGate, it also concluded that the single node's configuration or capacity is not enough to use for the extensive matrix computation (Sampath, Sagar & Nanjesh 2013). Thus more than one node is supposed to be used for the MPI parallelization method to perform better than serial execution and other parallelization methods.

According to Figure 4,1 shown above, the speedup of CUDA parallelization is worse than MPI and sequential methods for matrix size 256x256 and below. This is due to the penalty associated with the GPU parallelization where it requires transferring of the data to GPU over the PCI express bus before parallelization can be performed where MPI and serial execution are not required. (Sims 2009) Furthermore, according to Caulfield (2009), CPU consists of a few cores with lots of cache memory compared to GPU with many cores that handle multiple threads concurrently. The cache memory allows faster execution, however, due to the limitation of the cores available when problem size increases, it causes the performance to decrease. Meanwhile, GPUs, with hundreds to thousands of cores available, scales much better when the problem size increases significantly.

## 5 Amdahl’s Law

Table: 5.1: Amdahl’s law formula

$$Speedup\ =\ 1\frac{1}{(1-fe)+f_i}\ \text{- (1)}$$

$$fe = \frac{T_1-T}{T_1}\ \text{- (2)}$$

Where $f_e$ refers to fraction of enhancement, $f_i$ refers to factor of improvement which represent number of threads used, $T_1$ refers to serial runtime, and $T_N$ refers to the parallel runtime. (Tuomanen 2018)

To sum it up, Amdahl's Law is a simple formula that allows us to roughly (very roughly) estimate potential speedup for a program that can be at least partially parallelized. This can provide a general idea as to whether it will be worthwhile to write a parallel version of a particular serial program, provided we know what proportion of the code we can parallelize (p), and how many cores we can run our parallelized code on (N).

Two uses are attempted for this equation. First, we compare the predicted speedup with the actual speedup. This is done by substituting the fraction of enhancement and the number of threads used as the factor of improvement into the equation. Secondly, we tried to predict the factor of improvement needed to achieve the actual speedup we have obtained.

### 5.1 Configuration

OpenMP: Use OMP-SIMD, set the thread size, matrix size, chunk size, according to Table 8.3 (Appendix 8.1.3)
MPI: Use MPI_Allgather, set the thread size to 2
CUDA: Use concurrent kernel execution, set thread size to 32

Note that for Amdahl’s Law, different test environments are used from the analysis and discussion, but the same for all 3 configurations. Therefore, expected results will differ due to different CPU and GPU specifications.

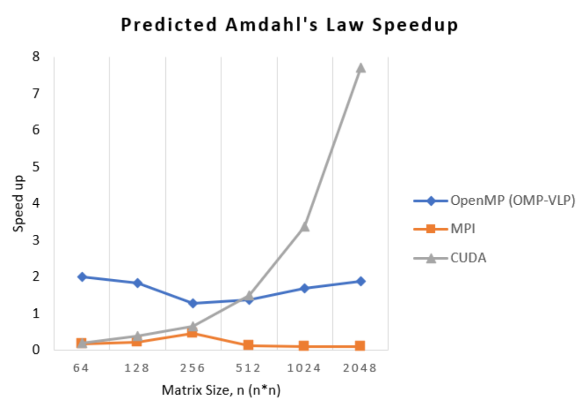
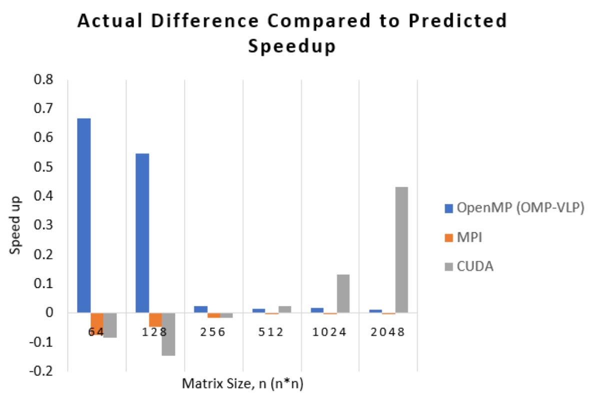
Table 5.2 : Testing Environment for all model to carry out final performance evaluation

| | | |
|---|---|---|
| **Test Environment:** | CPU: | Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz (4 Cores, 8 Logical Processors) 256 KB Level 1 (L1) cache, 1MB Level 2 (L2) Cache, 6MB Level 3 (L3) cache) (Hinum 2017) |
| | Motherboard: | LNVNB161216 |

| | | |
|---|---|---|
| | Graphic Card: | NVIDIA GeForce MX150 (2GB, GDDR5)<br><br>Intel(R) UHD Graphics 620 |
| | RAM: | 12.00 GB |

## 5.2 Comparison Graph

Table 5.3: Amdahl's law comparison results among OpenMP, CUDA and MPI

| Predicted Amdahl's Law Speedup (Table 8.37 in Appendix 8.4.2) | Actual Difference Compared to Predicted Speedup (Table 8.38 in Appendix 8.4.3) |
|---|---|
|  |  |

## 5.3 Discussion for Amdahl's Law

The first column in Table 5.3 shows the result of predicted speedup using Amdahl's Law, while the second column in Table 5.3 shows the difference between the predicted and the actual speedup.

The actual speedup for OMP-SIMD initially deviates from Amdahl's Law by a significant margin. However, from matrix size 256x256 onwards, it follows Amdahl's law prediction quite closely with a deviation of not more than 0.03, and only goes closer to the prediction as the size of the matrix increases, to 0.013 at matrix size of 2048. Most comparisons have shown positive deviation in OpenMP, suggesting that the actual speedups are mostly higher than the predicted speedup.

The main reason for the deviation is due to the fact that Amdahl's Law does not consider cache memory and virtual memory (Wolfe 2015). A smaller matrix size will be more likely to fit into the processor's cache memory. Fitting the matrix into the cache is a step function, if it fits, it fits, and is accessed at ten times or greater speed than main memory (Wolfe 2015). Otherwise, the matrix is accessed at memory speeds. Therefore, when the matrix size is small, the matrix has a greater chance of fitting into the L3, L2 or even the L1 cache, which is extremely fast compared to main memory. Amdahl's law doesn't allow for this, and only considers processors. This is expected because the first cache memory system was the IBM System 360 Model 85 in 1968, a year after Amdahl's original paper was published (Wolfe 2015).

Based on Table 5.3, it can be seen that the MPI parallelization method obeys Amdahl's Law as the range for the differences between the actual speedup and predicted speedup is not more than ± 0.1. However, the real speedup is slightly less than the expected speedup because MPI includes some I/O operation for the user to enter matrix size. Furthermore, some loops in the MPI implementation are split into parts, one per processor, which causes loop overhead. At the same time, Amdahl's law usually ignores the parallelization overhead (Texas Advanced Computing Center 2021). Thus, the speedup obtained is lower than the predicted speed.

For CUDA parallelization, only half of the method obeys Amdahl's Law with the difference between 0.01 to 0.08 for matrix size 64, 256 and 512. This indicates that the number of threads used is utilized efficiently according to the input matrix size. For matrix size 128, it disobeyed Amdahl's law and having the actual speedup is lower than predicted speedup , this indicates that the number of threads used is not fully utilized for parallelization. On the other hand, for matrix size 1024 and 2048, they also disobey Amdahl's law while having actual speedup is higher than the expected speedup, it indicates that the number of threads utilized is performed much better than the expected. (Tuomanen 2018)

## **6 Conclusion**

In conclusion, parallelizing the LU decomposition to improve the performance by using the parallelization method seems successful except for MPI implementation. OpenMP and CUDA prove that the computation time and the speedup for carrying out the LU decomposition are lesser and greater than the serial execution, especially when the matrix size gets larger. These results are significant because enormous matrix sizes and massive computation are required for whole image reconstruction. In terms of speed-up, MPI implementation obeys Amdahl's law throughout this project while in CUDA and OpenMP only half of the method obeys Amdahl's Law while another half of the method deviates from Amdahl's Law. This is mainly due to factors that are not considered in Amdahl's Law, because it only considers processors.

For OpenMP implementation, OMP-SIMD is the best performing overall because it leverages instruction level parallelism, dynamic scheduling, and SIMD simultaneously. After OMP-SIMD, solution B (OMP-DS) is faster at lower sizes, while solution A (OMP-ILP) is faster at larger sizes. The key limitation of OMP-SIMD is that as the problem size grows, the time required increases more compared to CUDA, which leads to poor scaling.

For CUDA implementation, the concurrent kernel execution which utilizes multiple streams achieved the best performance. This is because the multiple streams allows the multiple execution to be carried out at the same time that it better utilizes the multiprocessor. The limitation of concurrent kernel execution is that the execution run in parallel must be dependent with each other to prevent false sharing which causes the result to be inaccurate.

For MPI implementation, MPI_Allgather can have the fastest execution time and most excellent speedup at two threads compared to other parallelization methods and other numbers of threads. This is because the MPI_Allgather method with two threads does not require much communication time between the processes as only two processes are involved and can do the data sending-receiving communication simultaneously. Results also suggested that MPI should be implemented for distributed computing rather than run in a single node as run in the single node will negatively affect the parallelization performance.

Last but not least, the matrix size and the computation environment play an important role in determining the suitable parallelization method to use to maximize the calculation performance. If the matrix size is extremely large but the number of nodes is limited, CUDA is the best choice, as the GPU has more cores than CPU in a single node for the program to be handled by multiple threads simultaneously. If the matrix size is medium and the number of nodes are limited, OpenMP will be better because the cache memory in CPU allows faster execution than GPU although it has a lesser number of cores. If the matrix size is extremely large and the number of nodes is not limited, MPI is the best choice because distributed computing can be carried out for the computation which will improve the performance.

In the future, more parallel computing models should be implemented, such as OpenACC, OpenCL, and the hybrid model, so that more accurate results can be produced. By comparing the performance of the parallel computing model, a better decision can be made to choose the most efficient model for parallelizing the LU decomposition process. Furthermore, the testing environment should be maintained when carrying out the parallelization process for OpenMP, CUDA, and MPI so that the results are consistent and avoid bias. Last but not least, for OpenMP, there are many more OpenMP directive customization that can be applied to potentially improve performance.

## 7 References

Asiri, S 2018, *Is Concurrency Really Increase the Performance?*, Medium, viewed 19 September 2021, <https://towardsdatascience.com/is-concurrency-really-increases-the-performance-8cd06dd762f6>.

Aveuh 2018, *Coding Games and Programming Challenges to Code Better*, CodinGame, viewed 19 September 2021, <https://www.codingame.com/playgrounds/349/introduction-to-mpi/introduction-to-distributed-computing>.

Bikker, J 2017, *Practical SIMD Programming*, viewed 19 September 2021, <http://www.cs.uu.nl/docs/vakken/magr/2017-2018/files/SIMD%20Tutorial.pdf>.

Bosio, A 2017, *MPSoC Architectures OpenMP*, viewed 19 September 2021, <http://www.lirmm.fr/~bosio/USTH/mpsoc/04-openmp.pdf>.

Caulfield, B 2009, *Difference Between a CPU and a GPU? | The Official NVIDIA Blog*, The Official NVIDIA Blog, viewed 19 September 2021, <https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/>.

CodeGuru 2007, *Why Too Many Threads Hurts Performance, and What to do About It*, CodeGuru, viewed 19 September 2021, <https://www.codeguru.com/cplusplus/why-too-many-threads-hurts-performance-and-what-to-do-about-it/>.

Cornell Virtual Workshop 2021, *Cornell Virtual Workshop: Collective Communication*, cvw.cac.cornell.edu, viewed 12 September 2021, <https://cvw.cac.cornell.edu/parallel/collect>.

Crovella, B 2020, *CUDA UNIFIED MEMORY*, viewed 8 September 2021, <https://www.olcf.ornl.gov/wp-content/uploads/2019/06/06_Managed_Memory.pdf>.

Eijkhout, V 2020, *OpenMP topic: Loop parallelism*, Utexas.edu, viewed 19 September 2021, <https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html>.

Fan, H-Y 2016, *Direct Methods for Solving Linear Systems*, Department of Mathematics, National Taiwan Normal University, Taiwan, viewed 8 September 2021, <http://math.ntnu.edu.tw/~hyfan/linked_files/courses/NumerAnal/Course_slides/CH06.pdf>.

Fang, J, Zhang, P, Li, Z, Tang, T, Chen, X, Chen, C & Yang, C 2016, ‘Evaluating Multiple Streams on Heterogeneous Platforms’, *Parallel Processing Letters*, vol. 26, no. 04, p. 1640002, viewed 4 May 2020, <https://arxiv.org/pdf/1603.08619.pdf>.

Galoppo, N, Govindaraju, N, Henson, M & Manocha, D 2005, *LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware* *, viewed 8 September 2021, <http://gamma.cs.unc.edu/LU-GPU/lugpu05.pdf>.

Harris, M 2012, *How to Optimize Data Transfers in CUDA C/C++*, NVIDIA Developer Blog, viewed 8 September 2021, <https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/>.

Harris, M 2013, *Unified Memory in CUDA 6*, NVIDIA Developer Blog, viewed 8 September 2021, <https://developer.nvidia.com/blog/unified-memory-in-cuda-6/>.

Hinum, K 2017, *Intel Core i5-8250U SoC - Benchmarks and Specs*, Notebookcheck, viewed 19 September 2021, <https://www.notebookcheck.net/Intel-Core-i5-8250U-SoC-Benchmarks-and-Specs.242172.0.html>.

Hinum, K 2019, *Intel Core i5-1035G1 Laptop Processor (Ice Lake)*, Notebookcheck, viewed 19 September 2021, <https://www.notebookcheck.net/Intel-Core-i5-1035G1-Laptop-Processor-Ice-Lake.423867.0.html>.

IBM 2018, *#pragma omp simd*, www.ibm.com, International Business Machines Corporation, viewed 19 September 2021, <https://www.ibm.com/docs/en/xl-c-and-cpp-linux/13.1.6?topic=pdop-pragma-omp-simd>.

Intel 2013, *Intel® Core$^{TM}$ i7-4700MQ Processor Product Specifications*, www.intel.com, viewed 19 September 2021, <https://ark.intel.com/content/www/us/en/ark/products/75117/intel-core-i7-4700mq-processor-6m-cache-up-to-3-40-ghz.html>.

Ismail El-Nashar, A 2011, ‘To Parallelize or Not to Parallelize, Speed Up Issue’, *International Journal of Distributed and Parallel systems*, vol. 2, no. 2, pp. 14–28.

Karlsson, C 2012, *OPTIMIZING MULTI-DIMENSIONAL MPI COMMUNICATIONS ON MULTI-CORE ARCHITECTURES by*, viewed 12 September 2021, <https://mountainscholar.org/bitstream/handle/11124/70678/Karlsson_mines_0052E_10005.pdf?sequence=1>.

Kendall, W 2019, *MPI Scatter, Gather, and Allgather · MPI Tutorial*, mpitutorial.com, viewed 12 September 2021, <https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/#:~:text=MPI_Gather%20is%20the%20inverse%20of>.

Kendall, W 2020, *MPI Send and Receive · MPI Tutorial*, mpitutorial.com, viewed 19 September 2021, <https://mpitutorial.com/tutorials/mpi-send-and-receive/>.

Knap, M & Czarnul, P 2019, ‘Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs’, *The Journal of Supercomputing*, vol. 75, no. 11, pp. 7625–7645.

NVIDIA Corporation n.d., *CUDA OPTIMIZATION, PART 1 NVIDIA Corporation*, viewed 8 September 2021, <https://www.olcf.ornl.gov/wp-content/uploads/2019/12/03-CUDA-Fundamental-Optimization-Part-1.pdf>.

NVIDIA Corporation & affiliates 2021, *CUDA C Best Practices Guide*, Nvidia.com, viewed 8 September 2021, <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>.

OSToday 2021, *Which scheduling algorithm is used in Windows 10?*, OS Today, viewed 19 September 2021, <https://ostoday.org/windows/frequent-question-which-scheduling-algorithm-is-used-in-windows-10.html>.

Reinders, J & Jeffers, J 2016, *High Performance Parallelism Pearls | ScienceDirect*, www.sciencedirect.com, viewed 19 September 2021, <https://www.sciencedirect.com/book/9780128038192/high-performance-parallelism-pearls>.

Sakharnykh, N 2017, *Maximizing Unified Memory Performance in CUDA*, NVIDIA Developer Blog, viewed 8 September 2021, <https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/>.

Sampath, S, Sagar, BB & Nanjesh, BR 2013, 'Performance evaluation and comparison of MPI and PVM using a cluster based parallel computing architecture', *2013 International Conference on Circuits, Power and Computing Technologies (ICCPCT)*.

Sanders, J & Kandrot, E 2010, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, J Dongarra (ed.), viewed 8 September 2021, <http://www.mat.unimi.it/users/sansotte/cuda/CUDA_by_Example.pdf>.

Santos, RS, Eler, DM & Garcia, RE 2016, 'Performance Evaluation of Data Migration Methods Between the Host and the Device in CUDA-Based Programming', *Advances in Intelligent Systems and Computing*, pp. 689–700, viewed 15 November 2019, <https://rd.springer.com/chapter/10.1007/978-3-319-32467-8_60>.

Satoh, S, Kusano, K & Sato, M 2001, 'Compiler Optimization Techniques for OpenMP Programs', *Scientific Programming*, vol. 9, no. 2-3, pp. 131–142, viewed 25 October 2020, <https://core.ac.uk/download/pdf/205999472.pdf>.

Sims, GD 2009, *LSU Digital Commons Parallel cloth simulation using OpenMp and CUDA*, viewed 19 September 2021, <https://digitalcommons.lsu.edu/cgi/viewcontent.cgi?article=4066&context=gradschool_dissertations>.

Sobral, B 2017, *OpenMP Scheduling*, Universidade Federal de Santa Catarina, viewed 19 September 2021, <http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf>.

Squyres, J 2005, *MPI: Processes, Processors, and MPI, Oh My! | MPI | Columns*, www.clustermonkey.net, viewed 19 September 2021, <https://www.clustermonkey.net/MPI/processes-processors-and-mpi-oh-my.html>.

Texas Advanced Computing Center 2021, *Intro to High Performance Scientific Computing | Victor Eijkhout's homepage*, www.tacc.utexas.edu, viewed 19 September 2021, <Https://Www.Tacc.Utexas.Edu/~Eijkhout/Istc/Istc.Html>.

Tuomanen, B 2018, *Hands-on GPU programming with Python and CUDA: explore high-performance parallel computing with CUDA*, *Open WorldCat*, Packt, viewed 19 September 2021, <https://www.worldcat.org/title/hands-on-gpu-programming-with-python-and-cuda-explore-high-performance-parallel-computing-with-cuda/oclc/1078552287>.

Tutorialspoint n.d., *CUDA - Memories - Tutorialspoint*, www.tutorialspoint.com, viewed 8 September 2021, <https://www.tutorialspoint.com/cuda/cuda_memories.htm>.

Virk, B 2010, *Implementing Method of Moments on a GPGPU using Nvidia CUDA*, viewed 8 September 2021, <https://smartech.gatech.edu/bitstream/handle/1853/33980/virk_bikram_j_201005_mast.pdf>.

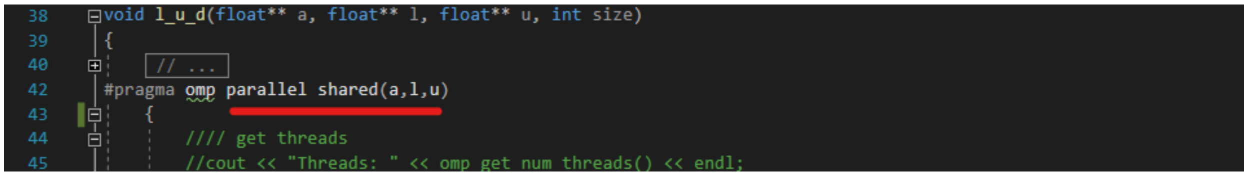
Wilper, H, Knight, R & Cohen, J 2020, *Understanding the Visualization of Overhead and Latency in NVIDIA Nsight Systems*, NVIDIA Developer Blog, viewed 28 June 2021, <https://developer.nvidia.com/blog/understanding-the-visualization-of-overhead-and-latency-in-nsight-systems/>.

Wolfe, M 2015, *Compilers and More: Is Amdahl's Law Still Relevant?*, HPCwire, viewed 19 September 2021, <https://www.hpcwire.com/2015/01/22/compilers-amdahls-law-still-relevant/>.

## 8 Appendix

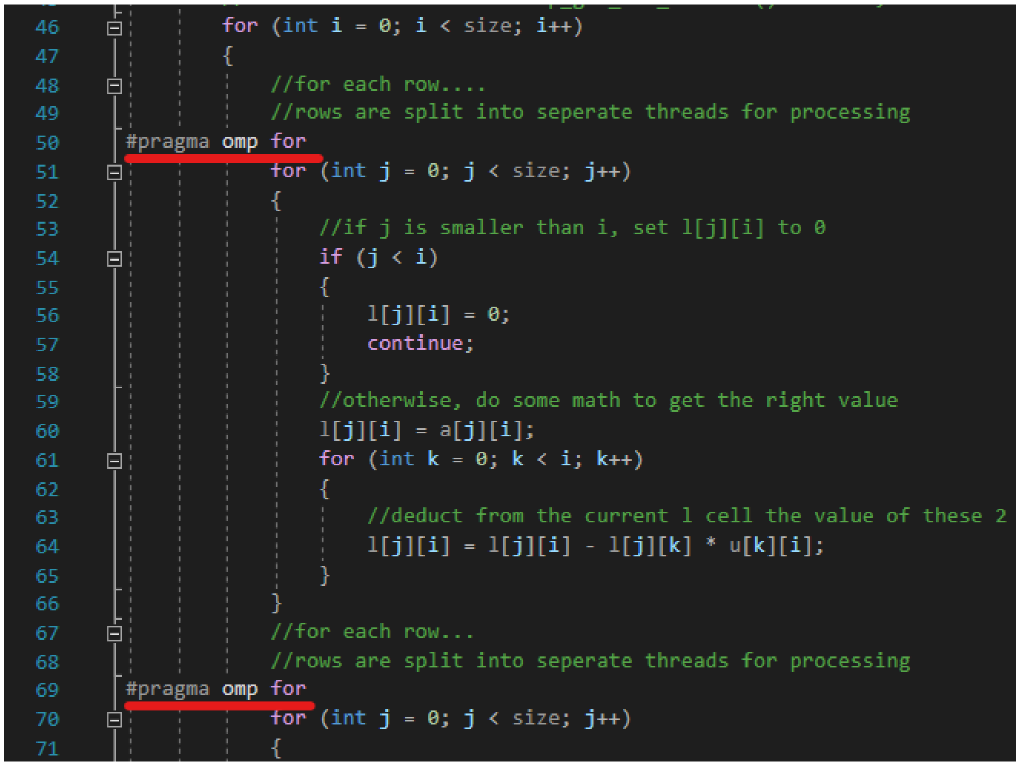
## 8.1 OpenMP

### 8.1.1 Implementation



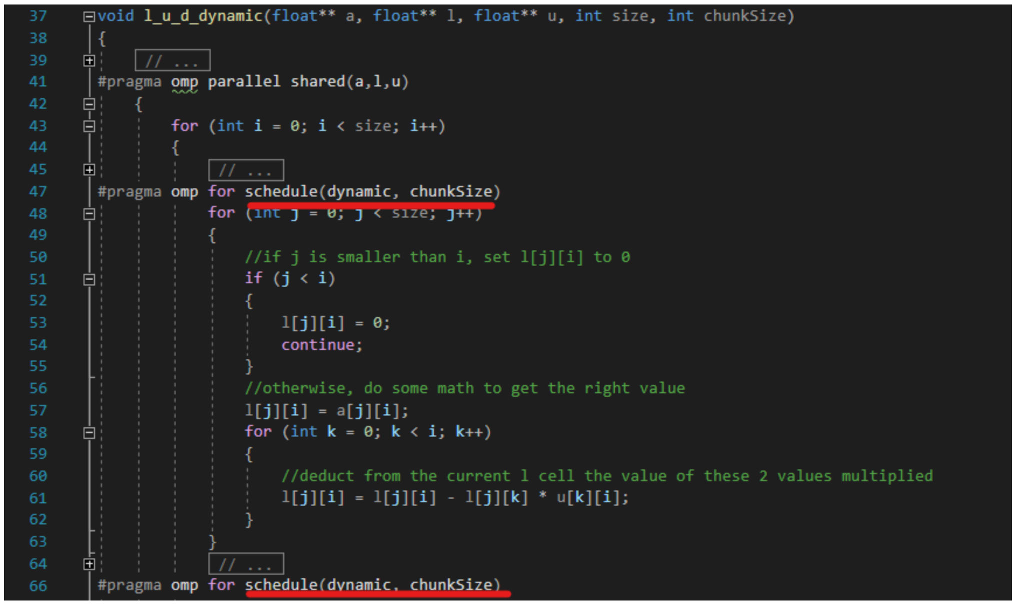
```
void l_u_d(float** a, float** l, float** u, int size)
{
    // ...
#pragma omp parallel shared(a,l,u)
    {
        //// get threads
        //cout << "Threads: " << omp_get_num_threads() << endl;
```

Figure 8.1:Use of shared variable for omp parallel block



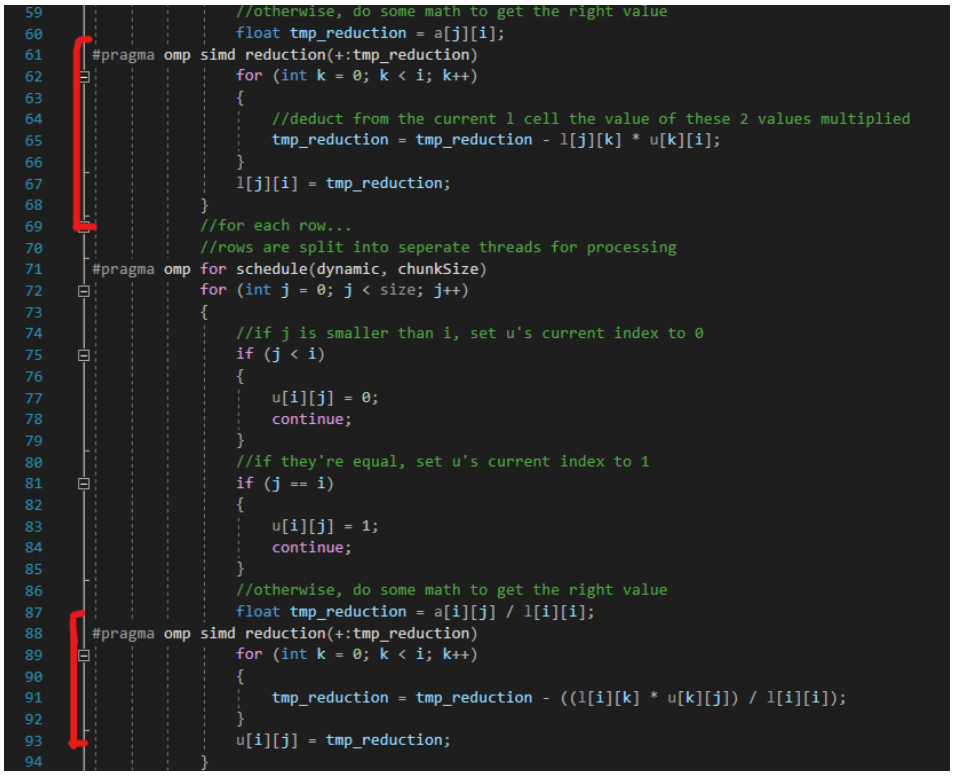
```
        for (int i = 0; i < size; i++)
        {
            //for each row....
            //rows are split into seperate threads for processing
#pragma omp for
            for (int j = 0; j < size; j++)
            {
                //if j is smaller than i, set l[j][i] to 0
                if (j < i)
                {
                    l[j][i] = 0;
                    continue;
                }
                //otherwise, do some math to get the right value
                l[j][i] = a[j][i];
                for (int k = 0; k < i; k++)
                {
                    //deduct from the current l cell the value of these 2
                    l[j][i] = l[j][i] - l[j][k] * u[k][i];
                }
            }
            //for each row...
            //rows are split into seperate threads for processing
#pragma omp for
            for (int j = 0; j < size; j++)
            {
```

Figure 8.2: Use of “for-loop”, instruction-level parallelism in  LU decomposition (OMP-ILP, OMP-DS, and OMP-SIMD)



```
void l_u_d_dynamic(float** a, float** l, float** u, int size, int chunkSize)
{
    // ...
#pragma omp parallel shared(a,l,u)
    {
        for (int i = 0; i < size; i++)
        {
            // ...
#pragma omp for schedule(dynamic, chunkSize)
            for (int j = 0; j < size; j++)
            {
                //if j is smaller than i, set l[j][i] to 0
                if (j < i)
                {
                    l[j][i] = 0;
                    continue;
                }
                //otherwise, do some math to get the right value
                l[j][i] = a[j][i];
                for (int k = 0; k < i; k++)
                {
                    //deduct from the current l cell the value of these 2 values multiplied
                    l[j][i] = l[j][i] - l[j][k] * u[k][i];
                }
            }
            // ...
#pragma omp for schedule(dynamic, chunkSize)
```

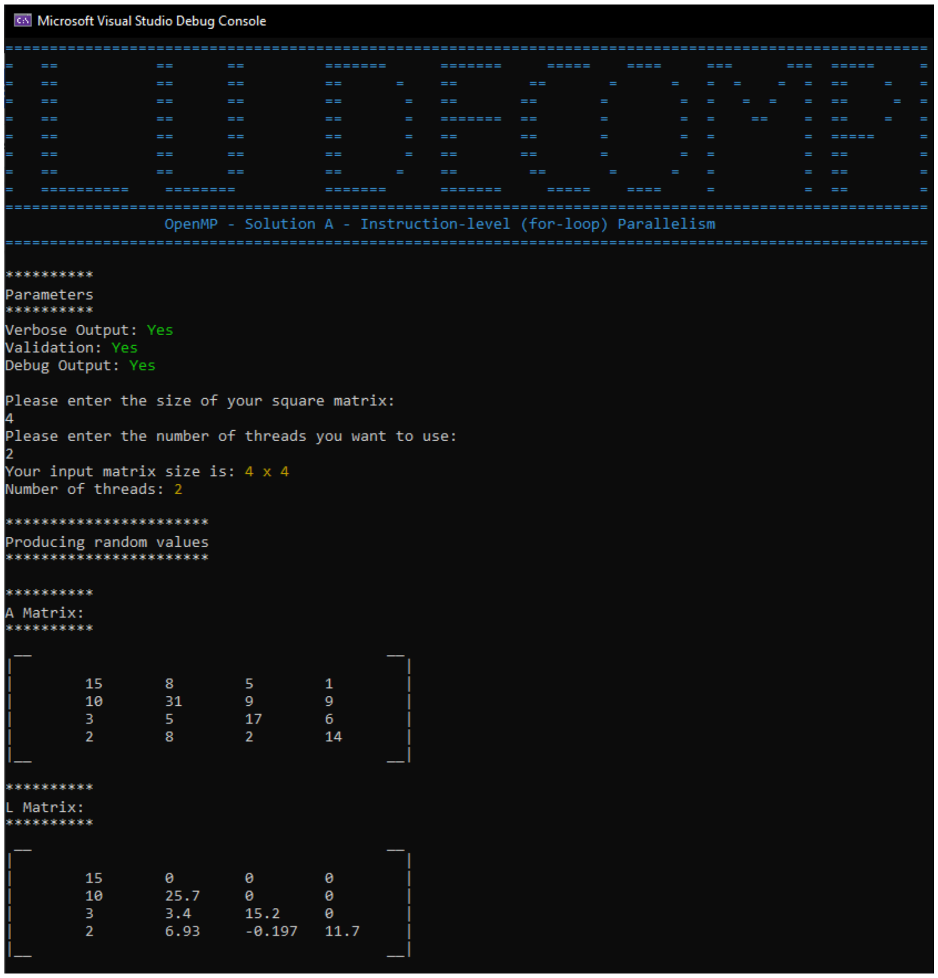
Figure 8.3:Use of dynamic scheduling in LU decomposition with varying chunk size. (OMP-DS & OMP-SIMD)



```
                //otherwise, do some math to get the right value
                float tmp_reduction = a[j][i];
#pragma omp simd reduction(+:tmp_reduction)
                for (int k = 0; k < i; k++)
                {
                    //deduct from the current l cell the value of these 2 values multiplied
                    tmp_reduction = tmp_reduction - l[j][k] * u[k][i];
                }
                l[j][i] = tmp_reduction;
            }
            //for each row...
            //rows are split into seperate threads for processing
#pragma omp for schedule(dynamic, chunkSize)
            for (int j = 0; j < size; j++)
            {
                //if j is smaller than i, set u's current index to 0
                if (j < i)
                {
                    u[i][j] = 0;
                    continue;
                }
                //if they're equal, set u's current index to 1
                if (j == i)
                {
                    u[i][j] = 1;
                    continue;
                }
                //otherwise, do some math to get the right value
                float tmp_reduction = a[i][j] / l[i][i];
#pragma omp simd reduction(+:tmp_reduction)
                for (int k = 0; k < i; k++)
                {
                    tmp_reduction = tmp_reduction - ((l[i][k] * u[k][j]) / l[i][i]);
                }
                u[i][j] = tmp_reduction;
            }
```

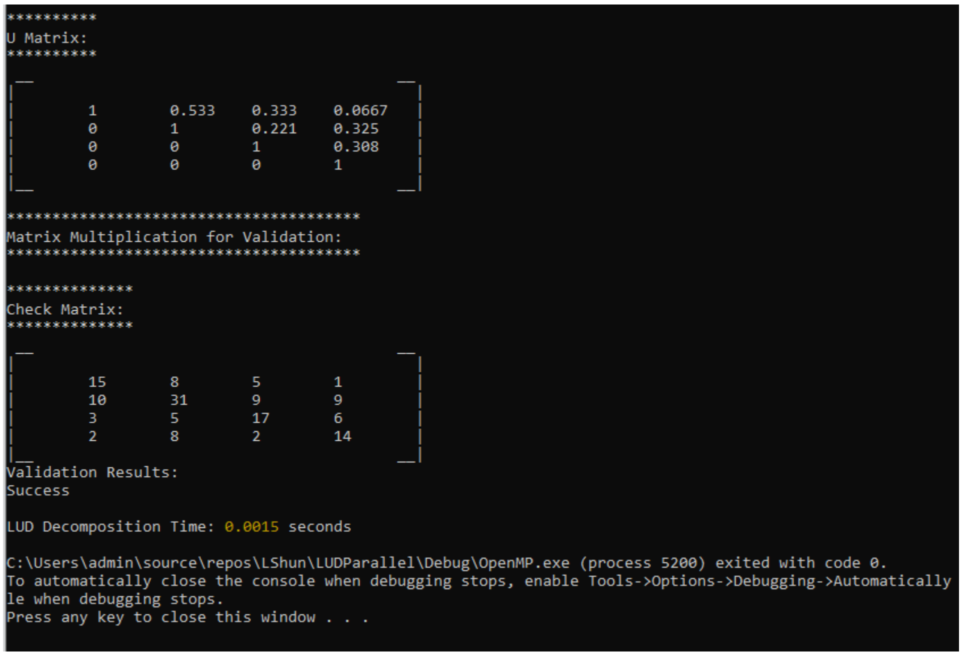
Figure 8.4: Use of SIMD, data-level parallelism (SIMD) in LU decomposition with varying chunk size. (OMP-SIMD)

## 8.1.2 Validation Results



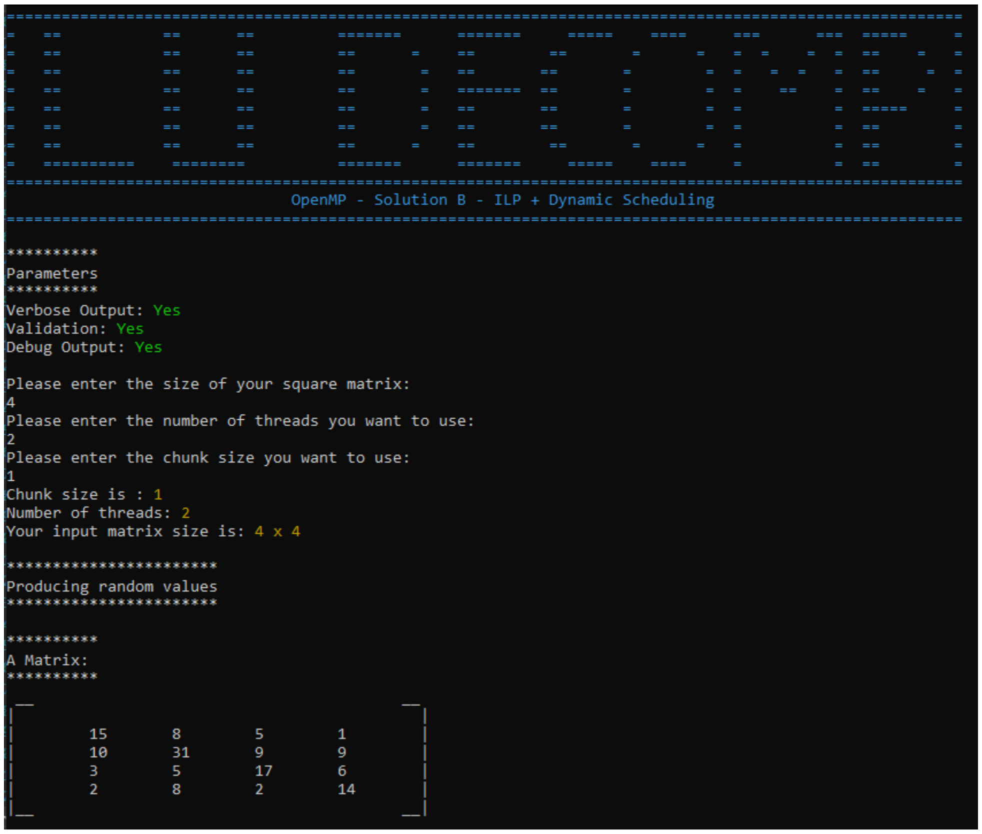
Microsoft Visual Studio Debug Console

```
=========================================================================================================
=   ==           ==      ==         =======      =======     =====     ====     ===      ===  =====     =
=   ==           ==      ==         ==      =    ==         ==        =      =   =  =    =  =  ==    =   =
=   ==           ==      ==         ==        =  ==       ==        =        =  =   =  =   =  ==     =  =
=   ==           ==      ==         ==        =  =======  ==        =        =  =    ==    =  ==    =   =
=   ==           ==      ==         ==        =  ==       ==        =        =  =          =  =====     =
=   ==           ==      ==         ==        =  ==       ==        =        =  =          =  ==        =
=   ==           ==      ==         ==      =    ==         ==        =      =   =         =  ==        =
=   ==========    ========          =======      =======     =====     ====     =          =  ==        =
=========================================================================================================
                  OpenMP - Solution A - Instruction-level (for-loop) Parallelism
=========================================================================================================

**********
Parameters
**********
Verbose Output: Yes
Validation: Yes
Debug Output: Yes

Please enter the size of your square matrix:
4
Please enter the number of threads you want to use:
2
Your input matrix size is: 4 x 4
Number of threads: 2

***********************
Producing random values
***********************

**********
A Matrix:
**********
 __                                           __
|                                               |
|        15       8        5        1           |
|        10       31       9        9           |
|        3        5        17       6           |
|        2        8        2        14          |
|__                                           __|

**********
L Matrix:
**********
 __                                           __
|                                               |
|        15       0        0        0           |
|        10       25.7     0        0           |
|        3        3.4      15.2     0           |
|        2        6.93     -0.197   11.7        |
|__                                           __|
```

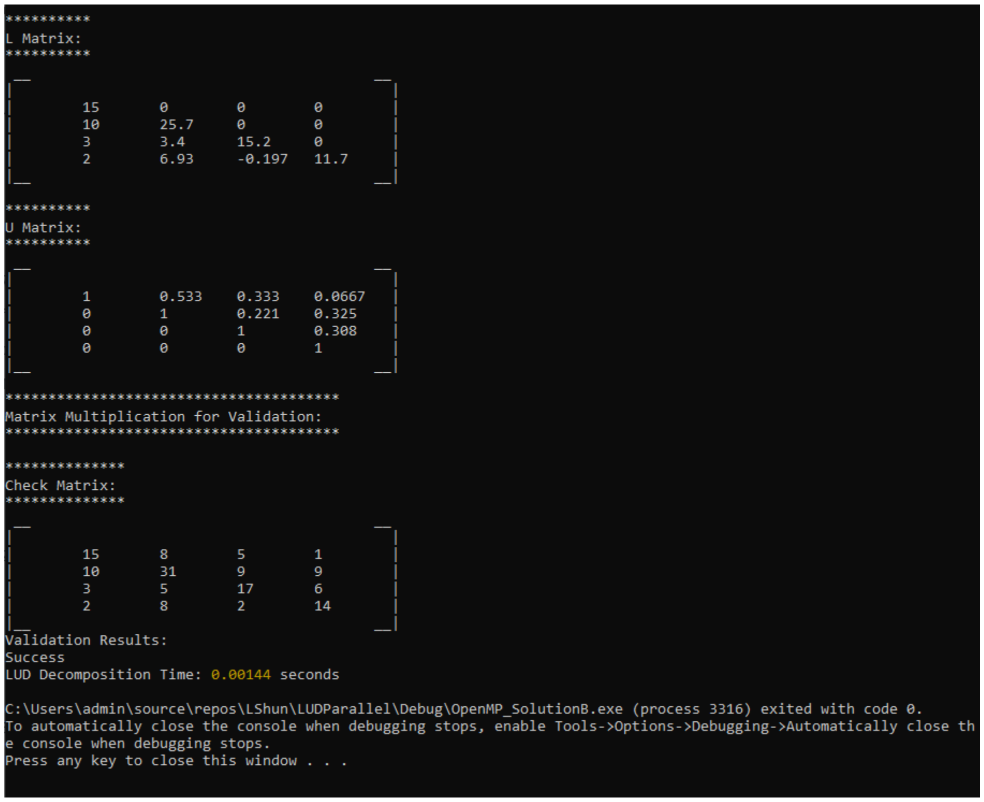


```
**********
U Matrix:
**********
 __                                           __
|                                               |
|        1        0.533    0.333    0.0667      |
|        0        1        0.221    0.325       |
|        0        0        1        0.308       |
|        0        0        0        1           |
|__                                           __|

****************************************
Matrix Multiplication for Validation:
****************************************

**************
Check Matrix:
**************
 __                                           __
|                                               |
|        15       8        5        1           |
|        10       31       9        9           |
|        3        5        17       6           |
|        2        8        2        14          |
|__                                           __|
Validation Results:
Success

LUD Decomposition Time: 0.0015 seconds

C:\Users\admin\source\repos\LShun\LUDParallel\Debug\OpenMP.exe (process 5200) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically
le when debugging stops.
Press any key to close this window . . .
```

Figure 8.5: Validation Result for OMP-ILP method

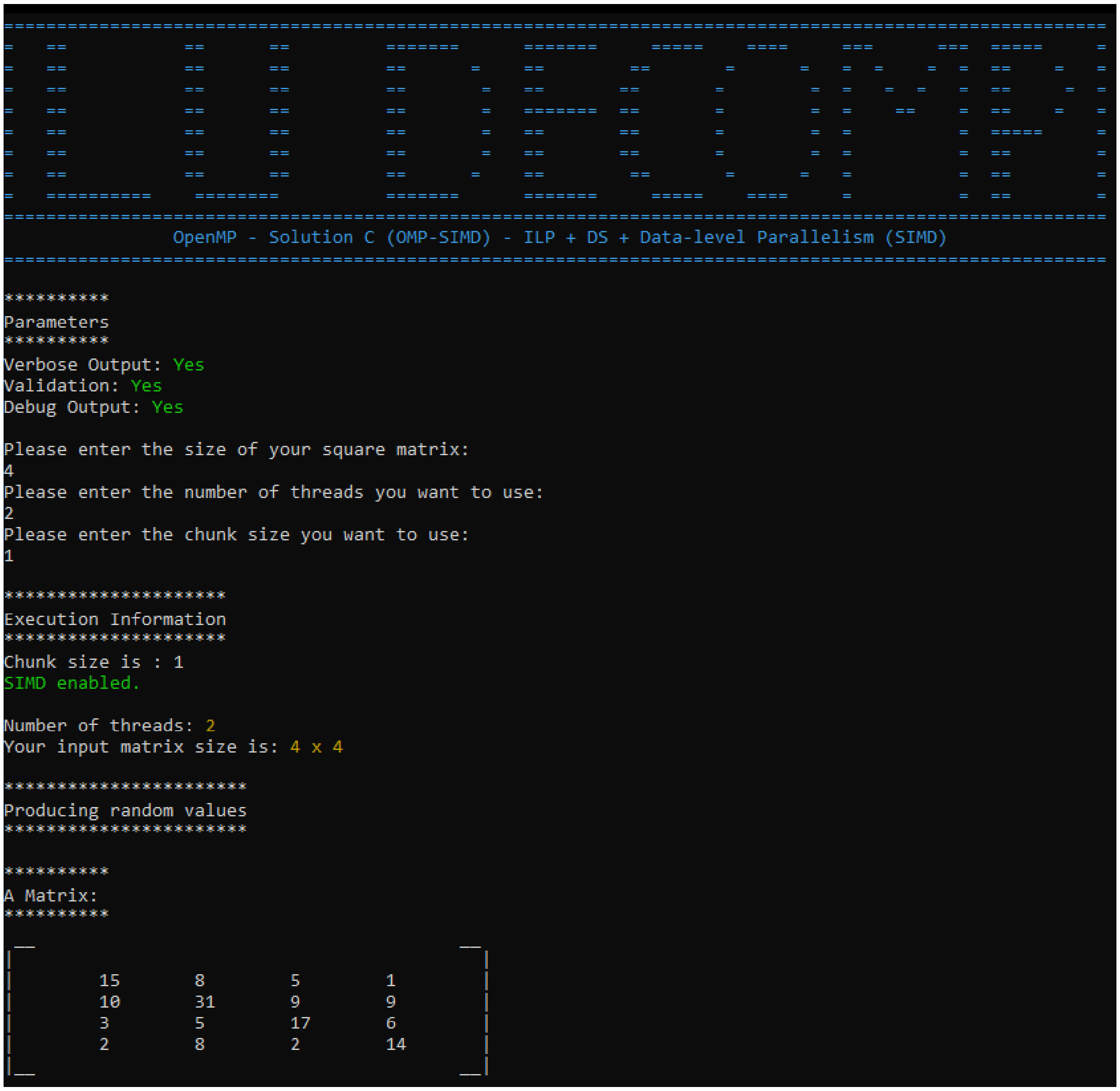


```
=========================================================================================================
=   ==           ==      ==         =======      =======     =====    ====      ===      ===  =====     =
=   ==           ==      ==         ==       =   ==         ==       =      =   =  = =   =  =  ==    =   =
=   ==           ==      ==         ==        =  ==        ==       =        =  =   =  =   =  ==     =  =
=   ==           ==      ==         ==        =  =======   ==       =        =  =    ==    =  ==    =   =
=   ==           ==      ==         ==        =  ==        ==       =        =  =          =  =====     =
=   ==           ==      ==         ==        =  ==        ==       =        =  =          =  ==        =
=   ==           ==      ==         ==       =   ==         ==       =      =   =          =  ==        =
=   =========     ========          =======      =======     =====    ====      =          =  ==        =
=========================================================================================================
                               OpenMP - Solution B - ILP + Dynamic Scheduling
=========================================================================================================

**********
Parameters
**********
Verbose Output: Yes
Validation: Yes
Debug Output: Yes

Please enter the size of your square matrix:
4
Please enter the number of threads you want to use:
2
Please enter the chunk size you want to use:
1
Chunk size is : 1
Number of threads: 2
Your input matrix size is: 4 x 4

***********************
Producing random values
***********************

**********
A Matrix:
**********
 __                                          __
|                                              |
|        15       8        5        1          |
|        10       31       9        9          |
|        3        5        17       6          |
|        2        8        2        14         |
|__                                          __|
```

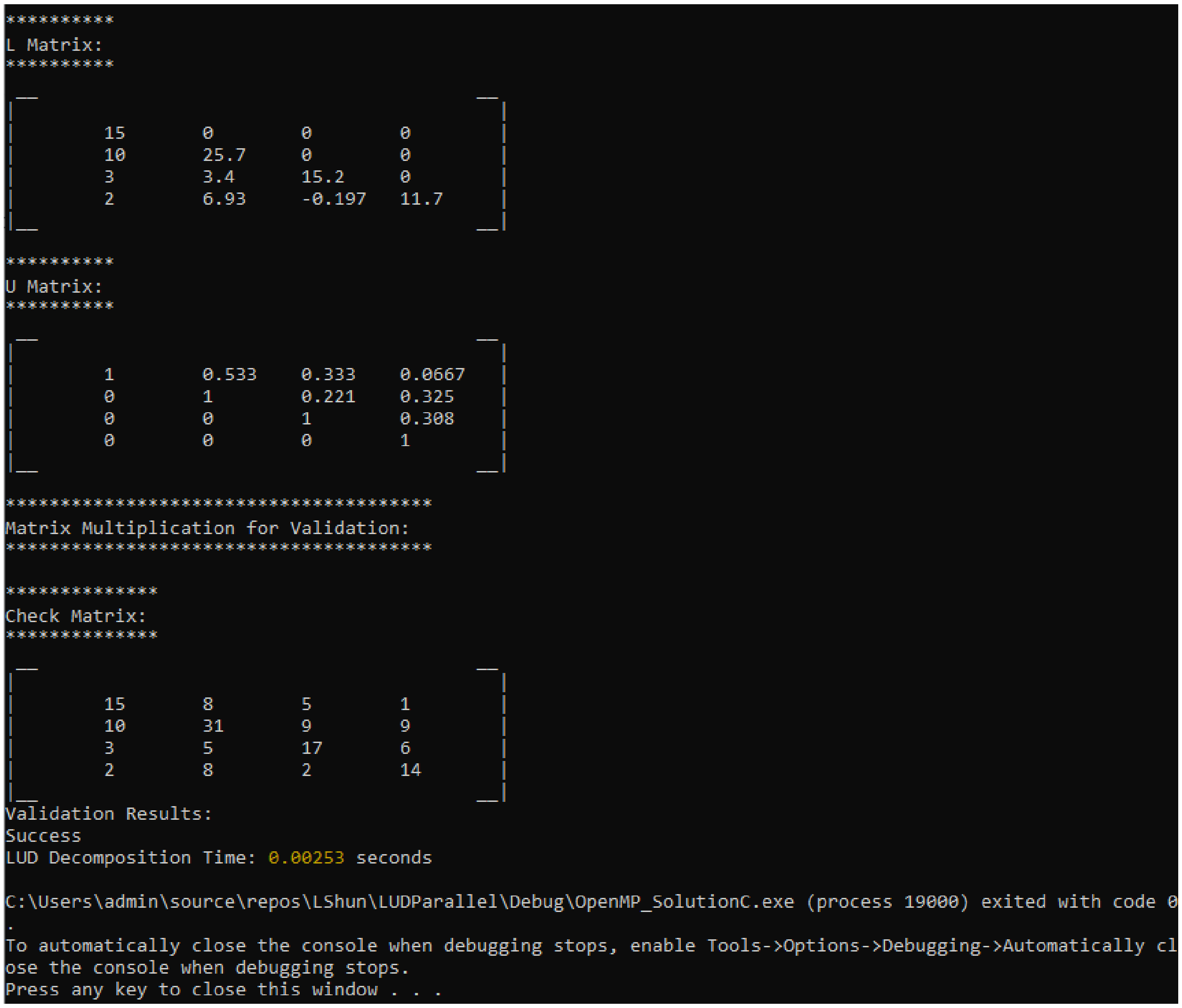


```
**********
L Matrix:
**********
 __                                       __
|                                           |
|        15       0        0        0       |
|        10       25.7     0        0       |
|        3        3.4      15.2     0       |
|        2        6.93     -0.197   11.7    |
|__                                       __|

**********
U Matrix:
**********
 __                                       __
|                                           |
|        1        0.533    0.333    0.0667  |
|        0        1        0.221    0.325   |
|        0        0        1        0.308   |
|        0        0        0        1       |
|__                                       __|

****************************************
Matrix Multiplication for Validation:
****************************************

**************
Check Matrix:
**************
 __                                       __
|                                           |
|        15       8        5        1       |
|        10       31       9        9       |
|        3        5        17       6       |
|        2        8        2        14      |
|__                                       __|
Validation Results:
Success
LUD Decomposition Time: 0.00144 seconds

C:\Users\admin\source\repos\LShun\LUDParallel\Debug\OpenMP_SolutionB.exe (process 3316) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close th
e console when debugging stops.
Press any key to close this window . . .
```

Figure 8.6: Validation Result for OMP-DS method (ILP + Dynamic Scheduling)



```
========================================================================================================
=   ==           ==      ==         =======      =======     =====    ====     ===      ===  =====     =
=   ==           ==      ==         ==      =    ==        ==       =      =   =  =    =  =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =   =  =   =  ==     =  =
=   ==           ==      ==         ==       =   =======  ==       =        =  =    ==    =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  =====     =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  ==        =
=   ==           ==      ==         ==      =    ==        ==       =      =   =          =  ==        =
=   ==========    ========          =======      =======     =====    ====     =          =  ==        =
========================================================================================================
                OpenMP - Solution C (OMP-SIMD) - ILP + DS + Data-level Parallelism (SIMD)
========================================================================================================

**********
Parameters
**********
Verbose Output: Yes
Validation: Yes
Debug Output: Yes

Please enter the size of your square matrix:
4
Please enter the number of threads you want to use:
2
Please enter the chunk size you want to use:
1

*********************
Execution Information
*********************
Chunk size is : 1
SIMD enabled.

Number of threads: 2
Your input matrix size is: 4 x 4

***********************
Producing random values
***********************

**********
A Matrix:
**********

 __                                        __
|                                            |
|        15        8         5         1     |
|        10        31        9         9     |
|        3         5         17        6     |
|        2         8         2         14    |
|__                                        __|
```



```
**********
L Matrix:
**********
 __                                      __
|                                          |
|       15       0        0        0       |
|       10       25.7     0        0       |
|       3        3.4      15.2     0       |
|       2        6.93     -0.197   11.7    |
|__                                      __|

**********
U Matrix:
**********
 __                                      __
|                                          |
|       1        0.533    0.333    0.0667  |
|       0        1        0.221    0.325   |
|       0        0        1        0.308   |
|       0        0        0        1       |
|__                                      __|

***************************************
Matrix Multiplication for Validation:
***************************************

**************
Check Matrix:
**************
 __                                      __
|                                          |
|       15       8        5        1       |
|       10       31       9        9       |
|       3        5        17       6       |
|       2        8        2        14      |
|__                                      __|
Validation Results:
Success
LUD Decomposition Time: 0.00253 seconds

C:\Users\admin\source\repos\LShun\LUDParallel\Debug\OpenMP_SolutionC.exe (process 19000) exited with code 0
.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically cl
ose the console when debugging stops.
Press any key to close this window . . .
```

Figure 8.7: Validation Result for OMP-SIMD method (ILP + DS + data-level parallelism, SIMD)

### 8.1.3 Execution Time

Table 8.1: Execution Time for (in seconds) for OMP-ILP (OpenMP Instruction-Level Parallelism)

| Matrix Size / Number of Threads | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| 2 | **0.0016** | **0.0076** | 0.0435 | 0.4417 | 3.3588 | 35.452 |
| 4 | 0.0029 | 0.008 | 0.0416 | 0.3718 | 2.7128 | 31.7816 |
| 8 | 0.0061 | 0.0095 | 0.0345 | 0.2705 | 2.3517 | 27.6914 |
| 16 | 0.0132 | 0.0150 | **0.0328** | 0.2491 | 1.404 | 15.8041 |
| 32 | 0.0249 | 0.0259 | 0.0432 | **0.2332** | 1.168 | 12.3498 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 64 | 0.0466 | 0.0494 | 0.1100 | 0.2914 | **1.0636** | 10.7682 |
| 128 | 0.0982 | 0.0994 | 0.1647 | 0.2467 | 1.1407 | **10.2528** |
| 256 | 0.5038 | 0.5116 | 0.7309 | 0.8098 | 1.3523 | 10.4641 |

Table 8.2: Execution Time for (in seconds) for OMP-DS (OpenMP Instruction-Level Parallelism + Dynamic Scheduling)

| Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Threads | 2 | 2 | 16 | 32 | 64 | 128 |
| Chunk Size | | | | | | |
| 1 | 0.0019 | 0.0056 | 0.0450 | 0.2542 | 2.0424 | 13.7990 |
| 2 | 0.0021 | 0.0044 | 0.0377 | 0.2360 | 1.8162 | 12.9012 |
| 4 | 0.0017 | 0.0047 | 0.0335 | 0.2410 | 1.7775 | 12.0686 |
| 8 | 0.0016 | 0.0047 | 0.0338 | 0.1917 | 1.6159 | 11.4159 |
| 16 | **0.0015** | **0.0035** | **0.0321** | **0.1666** | **1.4466** | **10.6580** |
| 32 | 0.0016 | 0.0046 | 0.0396 | 0.1899 | 1.4947 | 10.718 |

Table 8.3: Execution Time for (in seconds) for OMP-SIMD (OpenMP ILP + DS + Data-level Parallelism (SIMD))

| Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Threads | 2 | 2 | 16 | 32 | 64 | 128 |
| Chunk Size | 16 | 16 | 16 | 16 | 16 | 16 |
| Sequential | 0.0010 | 0.0070 | 0.0630 | 0.5220 | 4.690 | 46.2160 |
| OMP-ILP | 0.0016 | 0.0076 | 0.0328 | 0.2332 | 1.0636 | 10.2528 |
| OMP-DS | 0.0015 | 0.0035 | 0.0321 | 0.1666 | 1.4466 | 10.6580 |
| OMP-SIMD | **0.0015** | **0.0034** | **0.0283** | **0.1345** | **1.1284** | **8.8929** |

### 8.1.4 Speedup

Speedup = Serial Runtime / Parallel Runtime

Table 8.4: Speedup for OMP-ILP (OpenMP Instruction-Level Parallelism)

| Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Threads | 2 | 2 | 16 | 32 | 64 | 128 |
| Chunk Size | | | | | | |
| 2 | **0.6208** | 0.9261 | 1.4482 | 1.1819 | 1.3964 | 1.3036 |
| 4 | 0.3481 | **0.8773** | 1.5127 | 1.4039 | 1.7288 | 1.4542 |
| 8 | 0.1641 | 0.7382 | 1.8277 | 1.9299 | 1.9943 | 1.6690 |
| 16 | 0.0760 | 0.4669 | **1.9204** | 2.0955 | 3.3405 | 2.9243 |
| 32 | 0.0402 | 0.2707 | 1.4586 | **2.2387** | 4.0156 | 3.7423 |
| 64 | 0.0215 | 0.1418 | 0.5729 | 1.7911 | **4.4095** | 4.2919 |
| 128 | 0.0102 | 0.0704 | 0.3826 | 2.1162 | 4.1115 | **4.5076** |
| 256 | 0.0020 | 0.0137 | 0.0862 | 0.6446 | 3.4682 | 4.4166 |

Table 8.5: Speedup for OMP-DS (OpenMP Instruction-Level Parallelism + Dynamic Scheduling)

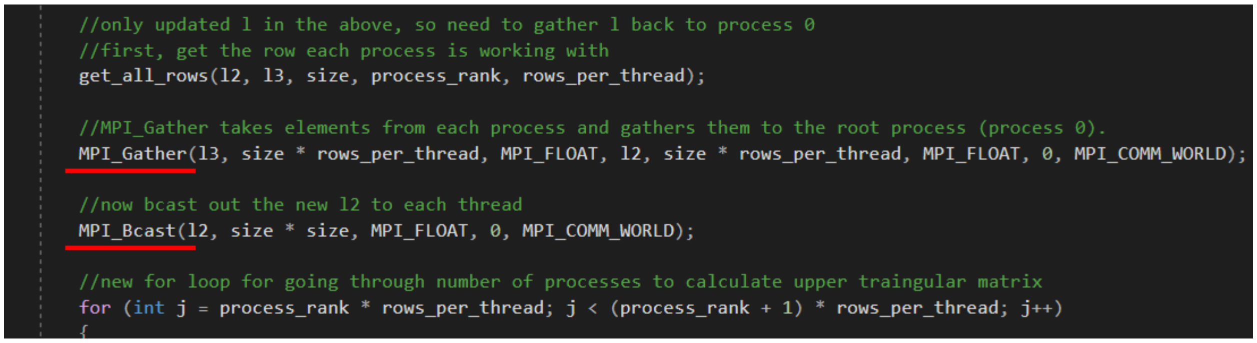
| Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Threads | 2 | 2 | 16 | 32 | 64 | 128 |
| Chunk Size | | | | | | |
| 1 | 0.5216 | 1.2395 | 1.3986 | 2.0538 | 2.2963 | 3.3492 |
| 2 | 0.4764 | 1.5747 | 1.6717 | 2.2120 | 2.5823 | 3.5823 |
| 4 | 0.5759 | 1.4746 | 1.8818 | 2.1662 | 2.6386 | 3.8294 |
| 8 | 0.6136 | 1.4759 | 1.8658 | 2.7225 | 2.9024 | 4.0484 |
| 16 | **0.6796** | **1.9842** | **1.9628** | **3.1333** | **3.2421** | **4.3363** |
| 32 | 0.6197 | 1.5289 | 1.5919 | 2.7484 | 3.1378 | 4.312 |

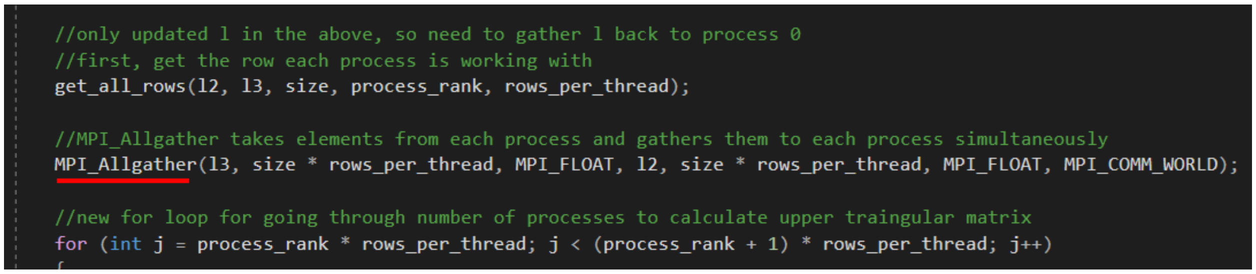
Table 8.6: Speedup for OMP-SIMD (OpenMP ILP + DS + Data-level Parallelism (SIMD))

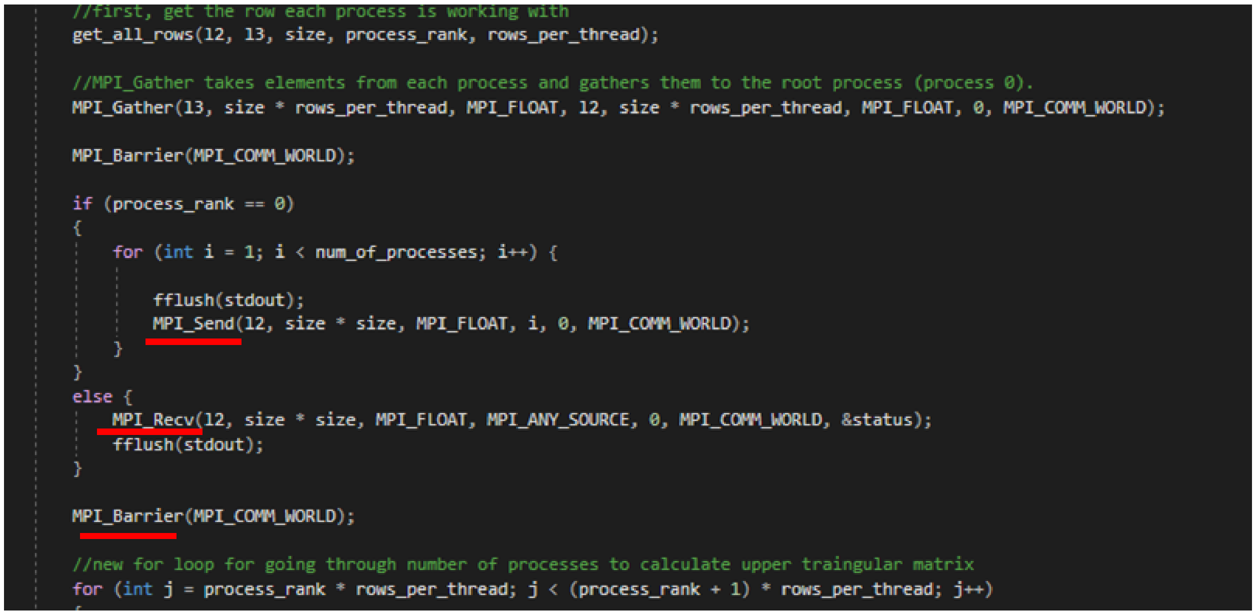
| Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Threads | 2 | 2 | 16 | 32 | 64 | 128 |
| Chunk Size | 16 | 16 | 16 | 16 | 16 | 16 |
| Sequential | 1 | 1 | 1 | 1 | 1 | 1 |
| OMP-ILP | 0.6208 | 0.9261 | 1.9204 | 2.2387 | 4.4095 | 4.5076 |
| OMP-DS | 0.6796 | 1.9842 | 1.9628 | 3.1333 | 3.2421 | 4.3363 |
| OMP-SIMD | **0.6817** | **2.0732** | **2.2277** | **3.8816** | **4.1564** | **5.197** |

## **<u>8.2 MPI</u>**

### **8.2.1 Implementation**

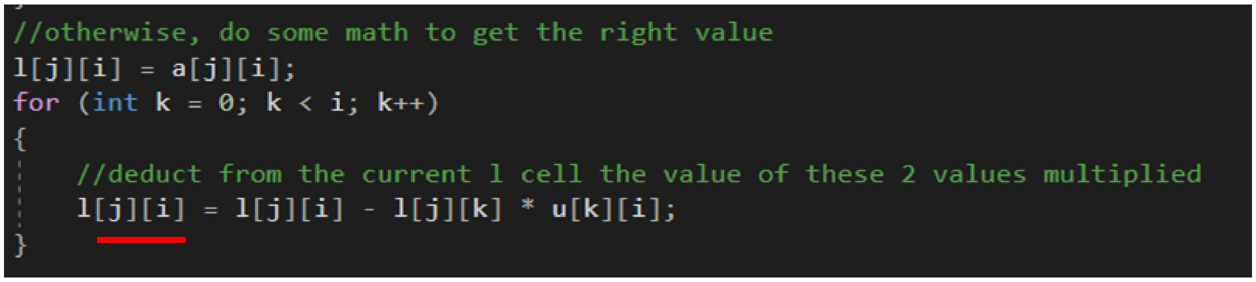

```
    //only updated l in the above, so need to gather l back to process 0
    //first, get the row each process is working with
    get_all_rows(l2, l3, size, process_rank, rows_per_thread);

    //MPI_Gather takes elements from each process and gathers them to the root process (process 0).
    MPI_Gather(l3, size * rows_per_thread, MPI_FLOAT, l2, size * rows_per_thread, MPI_FLOAT, 0, MPI_COMM_WORLD);

    //now bcast out the new l2 to each thread
    MPI_Bcast(l2, size * size, MPI_FLOAT, 0, MPI_COMM_WORLD);

    //new for loop for going through number of processes to calculate upper traingular matrix
    for (int j = process_rank * rows_per_thread; j < (process_rank + 1) * rows_per_thread; j++)
    {
```
Figure 8.8: Use of MPI_Bcast and MPI_Gather in LU decomposition


```
    //only updated l in the above, so need to gather l back to process 0
    //first, get the row each process is working with
    get_all_rows(l2, l3, size, process_rank, rows_per_thread);

    //MPI_Allgather takes elements from each process and gathers them to each process simultaneously
    MPI_Allgather(l3, size * rows_per_thread, MPI_FLOAT, l2, size * rows_per_thread, MPI_FLOAT, MPI_COMM_WORLD);

    //new for loop for going through number of processes to calculate upper traingular matrix
    for (int j = process_rank * rows_per_thread; j < (process_rank + 1) * rows_per_thread; j++)
    {
```
Figure 8.9: Use of MPI_Allgather in LU decomposition



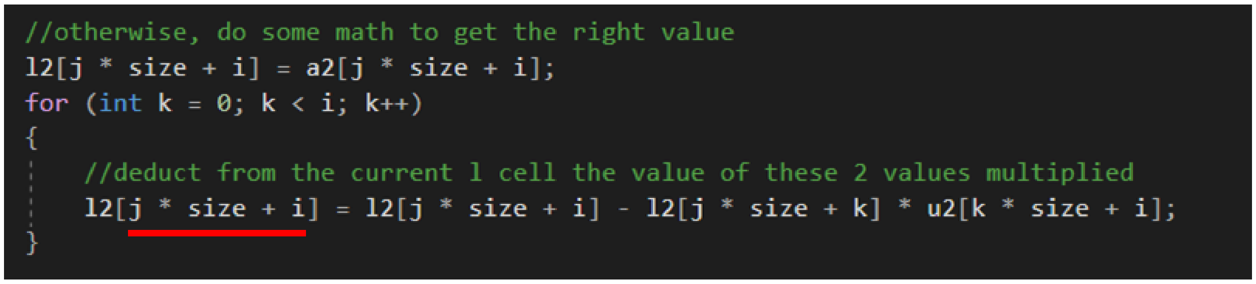
```
//first, get the row each process is working with
get_all_rows(l2, l3, size, process_rank, rows_per_thread);

//MPI_Gather takes elements from each process and gathers them to the root process (process 0).
MPI_Gather(l3, size * rows_per_thread, MPI_FLOAT, l2, size * rows_per_thread, MPI_FLOAT, 0, MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);

if (process_rank == 0)
{
    for (int i = 1; i < num_of_processes; i++) {

        fflush(stdout);
        MPI_Send(l2, size * size, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
    }
}
else {
    MPI_Recv(l2, size * size, MPI_FLOAT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    fflush(stdout);
}

MPI_Barrier(MPI_COMM_WORLD);

//new for loop for going through number of processes to calculate upper traingular matrix
for (int j = process_rank * rows_per_thread; j < (process_rank + 1) * rows_per_thread; j++)
```

Figure 8.10: Use of MPI_Send, MPI_Recv and MPI_Barrier in LU decomposition



```
//otherwise, do some math to get the right value
l[j][i] = a[j][i];
for (int k = 0; k < i; k++)
{
    //deduct from the current l cell the value of these 2 values multiplied
    l[j][i] = l[j][i] - l[j][k] * u[k][i];
}
```

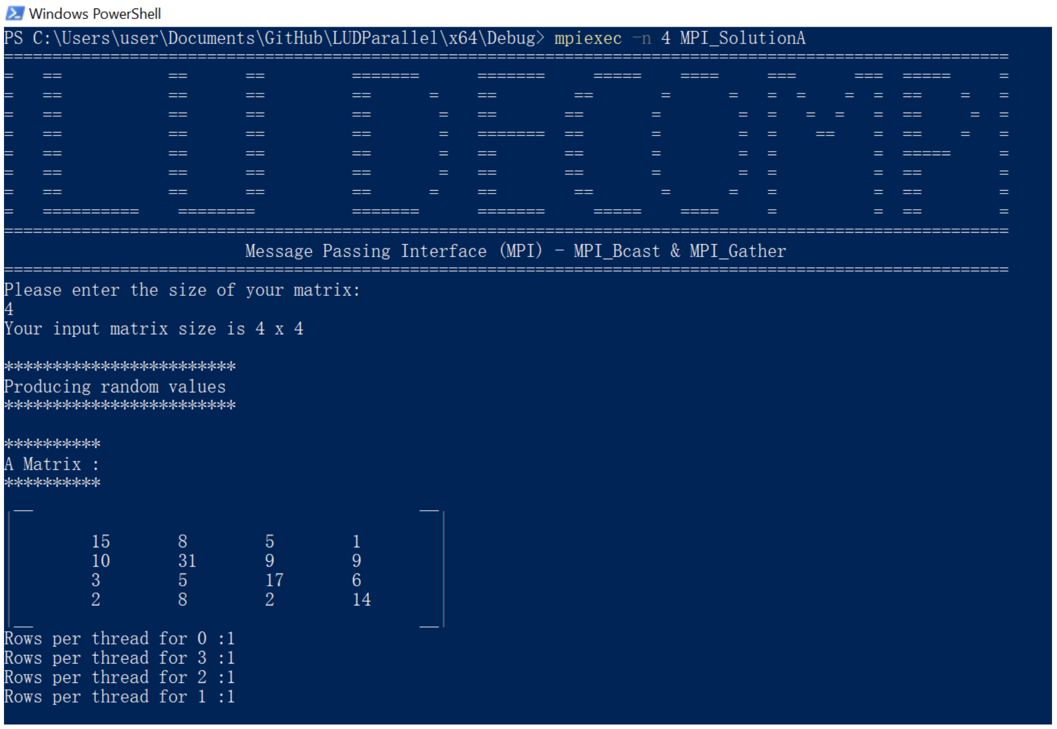
Figure 8.11: Use of 2D array in Sequential Code



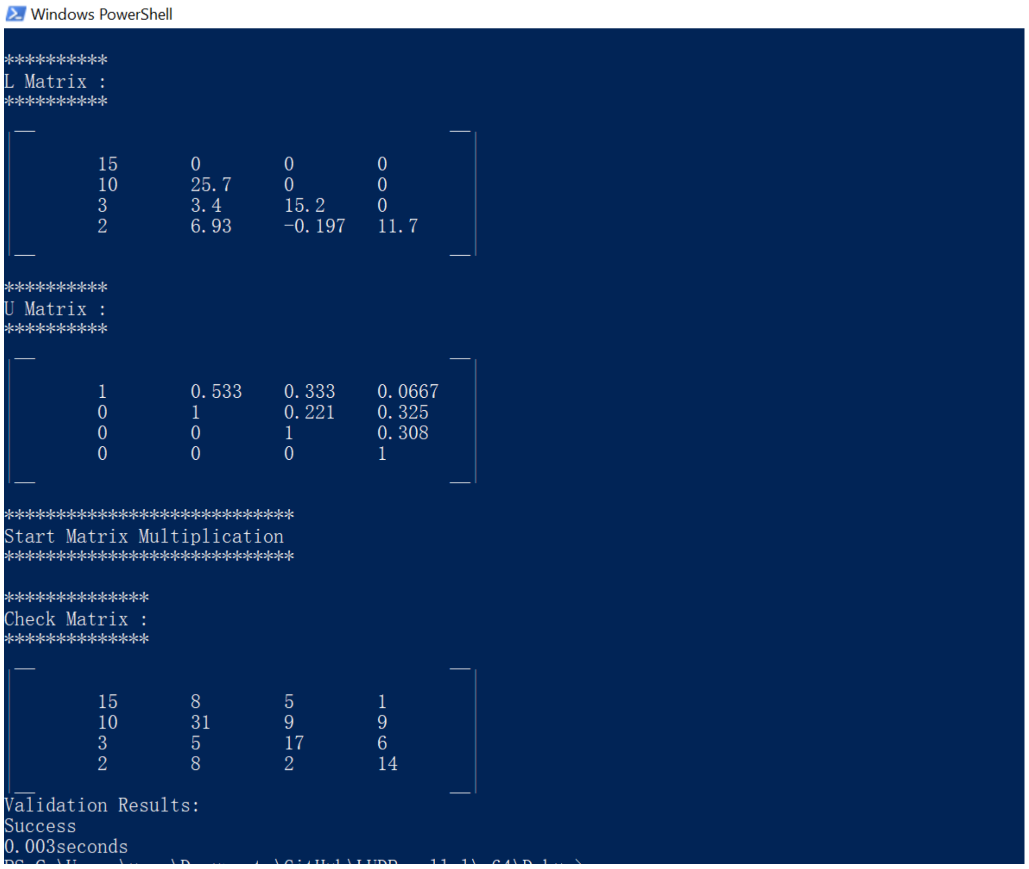
```
//otherwise, do some math to get the right value
l2[j * size + i] = a2[j * size + i];
for (int k = 0; k < i; k++)
{
    //deduct from the current l cell the value of these 2 values multiplied
    l2[j * size + i] = l2[j * size + i] - l2[j * size + k] * u2[k * size + i];
}
```

Figure 8.12: Use of 1D array in MPI Code

### 8.2.2 Validation Results

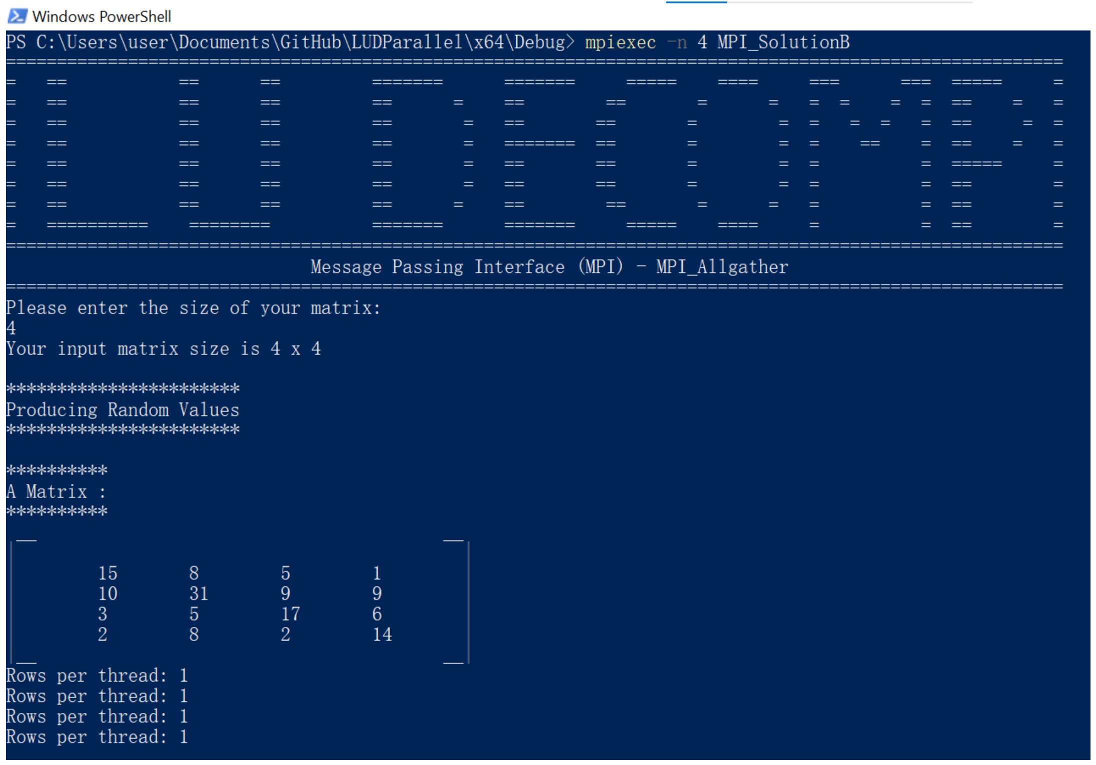


```
PS C:\Users\user\Documents\GitHub\LUDParallel\x64\Debug> mpiexec -n 4 MPI_SolutionA
========================================================================================================
=   ==           ==      ==         =======      =======     =====    ====     ===      ===  =====     =
=   ==           ==      ==         ==      =    ==        ==       =      =   =  =    =  =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =   =  =   =  ==     =  =
=   ==           ==      ==         ==       =   =======  ==       =        =  =    ==    =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  =====     =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  ==        =
=   ==           ==      ==         ==      =    ==        ==       =      =   =          =  ==        =
=   ==========    ========          =======      =======     =====    ====     =          =  ==        =
========================================================================================================
                         Message Passing Interface (MPI) - MPI_Bcast & MPI_Gather
========================================================================================================
Please enter the size of your matrix:
4
Your input matrix size is 4 x 4

************************
Producing random values
************************

**********
A Matrix :
**********
 __                                         __
|                                             |
|        15       8        5        1         |
|        10       31       9        9         |
|        3        5        17       6         |
|        2        8        2        14        |
|__                                         __|
Rows per thread for 0 :1
Rows per thread for 3 :1
Rows per thread for 2 :1
Rows per thread for 1 :1
```



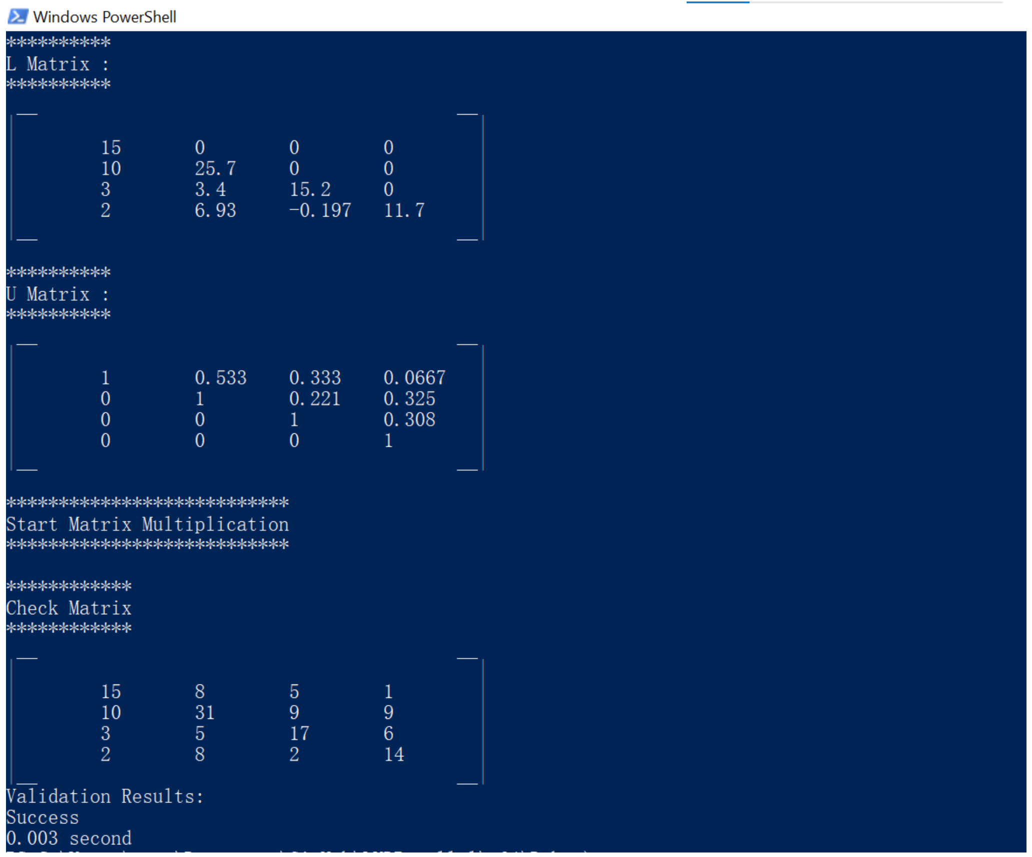
```

**********
L Matrix :
**********
 __                                             __
|                                                 |
|        15       0        0        0             |
|        10       25.7     0        0             |
|        3        3.4      15.2     0             |
|        2        6.93     -0.197   11.7          |
|__                                             __|

**********
U Matrix :
**********
 __                                             __
|                                                 |
|        1        0.533    0.333    0.0667        |
|        0        1        0.221    0.325         |
|        0        0        1        0.308         |
|        0        0        0        1             |
|__                                             __|

****************************
Start Matrix Multiplication
****************************

**************
Check Matrix :
**************
 __                                             __
|                                                 |
|        15       8        5        1             |
|        10       31       9        9             |
|        3        5        17       6             |
|        2        8        2        14            |
|__                                             __|
Validation Results:
Success
0.003seconds
```

Figure 8.13: Validation Result for MPI_Bcast and MPI_Gather method



Windows PowerShell

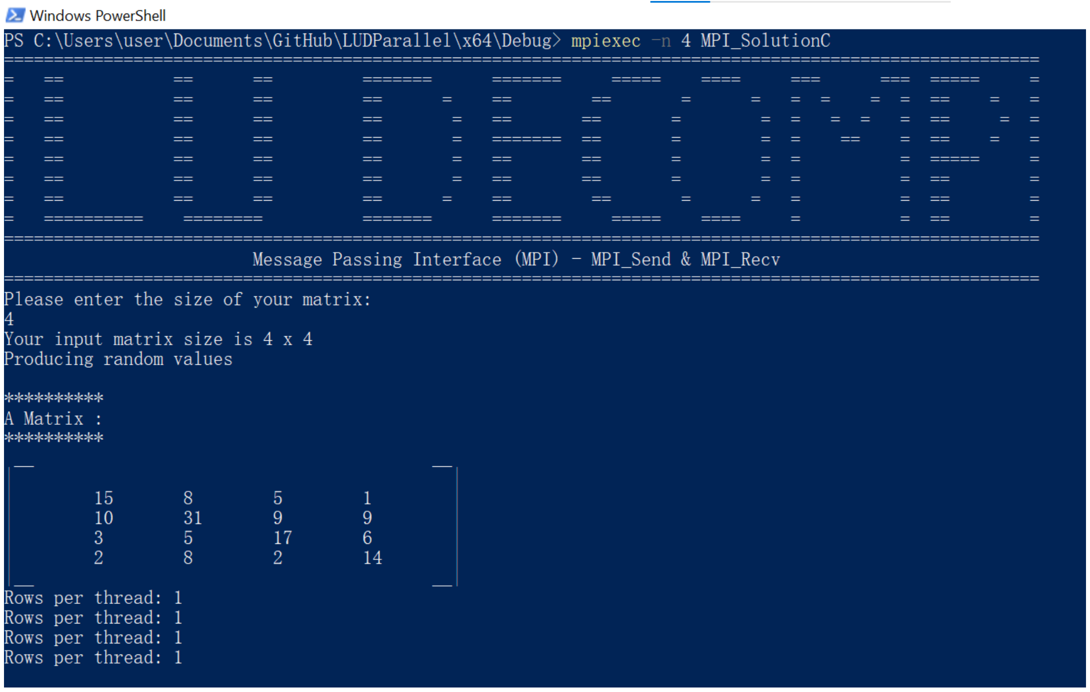
```
PS C:\Users\user\Documents\GitHub\LUDParallel\x64\Debug> mpiexec -n 4 MPI_SolutionB
========================================================================================================
=   ==           ==      ==         =======      =======     =====    ====     ===      ===  =====     =
=   ==           ==      ==         ==      =    ==        ==       =      =   =  =    =  =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =   =  =   =  ==     =  =
=   ==           ==      ==         ==       =   =======  ==       =        =  =    ==    =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  =====     =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  ==        =
=   ==           ==      ==         ==      =    ==        ==       =      =   =          =  ==        =
=   ==========    ========          =======      =======     =====    ====     =          =  ==        =
========================================================================================================
                              Message Passing Interface (MPI) - MPI_Allgather
========================================================================================================
Please enter the size of your matrix:
4
Your input matrix size is 4 x 4

***********************
Producing Random Values
***********************

**********
A Matrix :
**********

 __                                       __
|                                           |
        15       8        5        1
        10       31       9        9
        3        5        17       6
        2        8        2        14
|__                                       __|
Rows per thread: 1
Rows per thread: 1
Rows per thread: 1
Rows per thread: 1
```



Windows PowerShell

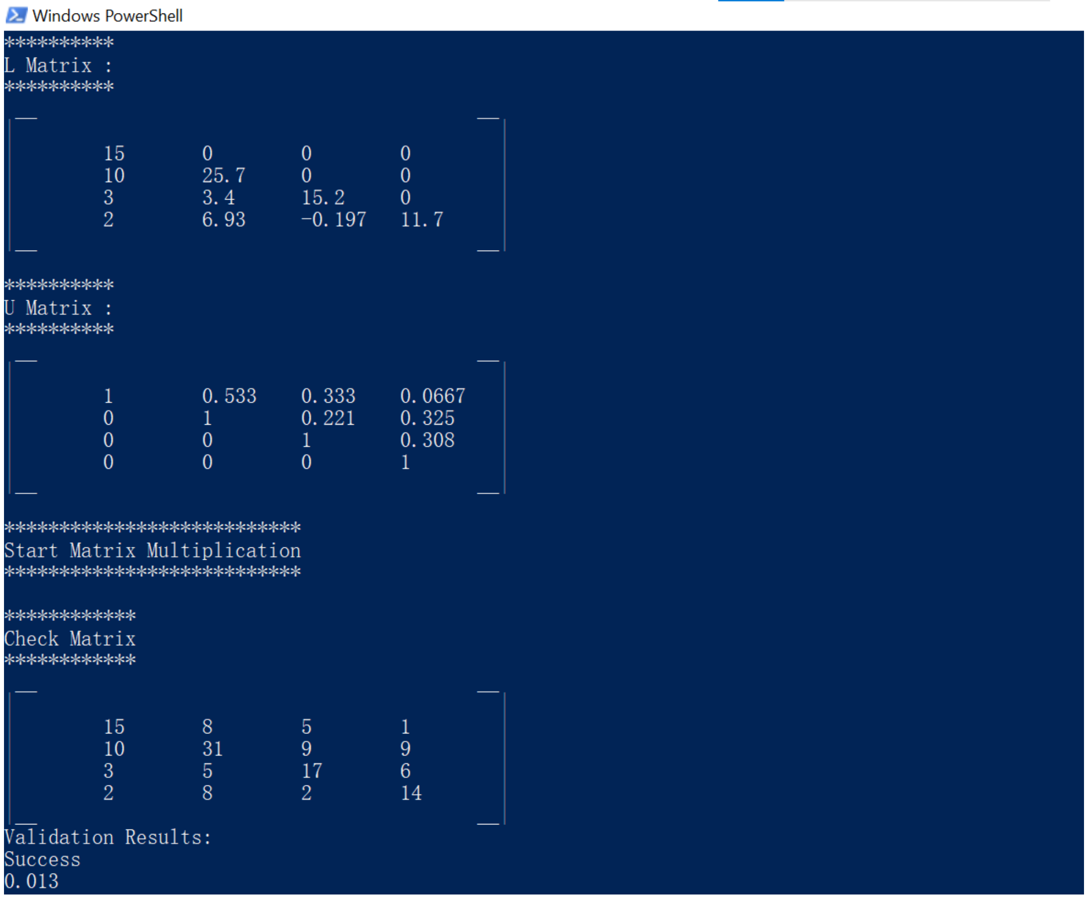
```
**********
L Matrix :
**********

 __                                       __
|                                           |
        15       0        0        0
        10       25.7     0        0
        3        3.4      15.2     0
        2        6.93     -0.197   11.7
|__                                       __|

**********
U Matrix :
**********

 __                                       __
|                                           |
        1        0.533    0.333    0.0667
        0        1        0.221    0.325
        0        0        1        0.308
        0        0        0        1
|__                                       __|

***************************
Start Matrix Multiplication
***************************

************
Check Matrix
************

 __                                       __
|                                           |
        15       8        5        1
        10       31       9        9
        3        5        17       6
        2        8        2        14
|__                                       __|
Validation Results:
Success
0.003 second
```

Figure 8.14: Validation Result for MPI_Allgather method



Windows PowerShell

```
PS C:\Users\user\Documents\GitHub\LUDParallel\x64\Debug> mpiexec -n 4 MPI_SolutionC
===================================================================================================
=   ==          ==      ==        =======     =======     =====   ====     ===     ===  =====     =
=   ==          ==      ==        ==      =   ==        ==       =     =   =  =   =  =  ==    =   =
=   ==          ==      ==        ==       =  ==       ==       =       =  =   =  =  =  ==     =  =
=   ==          ==      ==        ==       =  =======  ==       =       =  =    ==   =  ==    =   =
=   ==          ==      ==        ==       =  ==       ==       =       =  =         =  =====     =
=   ==          ==      ==        ==       =  ==       ==       =       =  =         =  ==        =
=   ==          ==      ==        ==      =   ==        ==       =     =   =         =  ==        =
=   ==========   ========         =======     =======     =====   ====     =         =  ==        =
===================================================================================================
                        Message Passing Interface (MPI) - MPI_Send & MPI_Recv
===================================================================================================
Please enter the size of your matrix:
4
Your input matrix size is 4 x 4
Producing random values

**********
A Matrix :
**********
 __                                         __
|                                             |
         15       8        5        1
         10       31       9        9
         3        5        17       6
         2        8        2        14
|__                                         __|
Rows per thread: 1
Rows per thread: 1
Rows per thread: 1
Rows per thread: 1
```



Windows PowerShell

```
**********
L Matrix :
**********
 __                                         __
|                                             |
         15       0        0        0
         10       25.7     0        0
         3        3.4      15.2     0
         2        6.93     -0.197   11.7
|__                                         __|

**********
U Matrix :
**********
 __                                         __
|                                             |
         1        0.533    0.333    0.0667
         0        1        0.221    0.325
         0        0        1        0.308
         0        0        0        1
|__                                         __|

***************************
Start Matrix Multiplication
***************************

************
Check Matrix
************
 __                                         __
|                                             |
         15       8        5        1
         10       31       9        9
         3        5        17       6
         2        8        2        14
|__                                         __|
Validation Results:
Success
0.013
```

Figure 8.15: Validation Result for MPI_Send and MPI_Recv method

### 8.2.3 Execution Time

Table 8.7: Execution Time for (in second) for MPI_Bcast and MPI_Gather

| Number of processors / Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| **2** | **0.0060** | **0.0195** | **0.1320** | **1.2445** | **24.8245** | **258.7760** |
| 4 | 0.0070 | 0.0225 | 0.1455 | 1.5145 | 26.3105 | 305.7160 |
| 8 | 0.0110 | 0.0330 | 0.2670 | 2.5375 | 38.5530 | 441.7060 |
| 16 | 0.4445 | 2.0825 | 4.6635 | 51.2320 | 369.8185 | 1227.9450 |

Table 8.8: Execution Time for (in second) for MPI_Allgather

| Number of processors / Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| **2** | **0.0040** | **0.0190** | **0.1190** | **1.1845** | **21.2965** | **242.3895** |
| 4 | 0.0065 | 0.0210 | 0.1275 | 1.4810 | 23.4535 | 279.4570 |
| 8 | 0.0075 | 0.0325 | 0.2195 | 2.4680 | 34.6175 | 427.6685 |

Table 8.9: Execution Time for (in second) for MPI_Send and MPI_Recv

| Number of processors / Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| **2** | **0.0075** | **0.0225** | **0.1360** | **1.3055** | **24.9715** | **269.2025** |
| 4 | 0.0085 | 0.0295 | 0.1580 | 1.5420 | 27.6955 | 344.3585 |
| 8 | 0.0135 | 0.0500 | 0.3430 | 3.3685 | 41.9970 | 507.5590 |

Table 8.10: Execution Time for (in second) for all MPI parallelization method with two threads

| MPI methods / Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| MPI_Bcast and MPI_Gather | 0.0060 | 0.0195 | 0.1320 | 1.2445 | 24.8245 | 258.7760 |

| | | | | | | |
|---|---|---|---|---|---|---|
| **MPI_Allgather** | **0.0040** | **0.0190** | **0.1190** | **1.1845** | **21.2965** | **242.3895** |
| MPI_Send and MPI_Recv | 0.0075 | 0.0225 | 0.1360 | 1.3055 | 24.9715 | 269.2025 |

### 8.2.4 Speedup

Speedup = Serial Runtime / Parallel Runtime

Table 8.11: Speedup for MPI_Bcast and MPI_Gather

| Number of processors / Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| **2** | **0.1667** | **0.3590** | **0.4773** | **0.4194** | **0.1889** | **0.1786** |
| 4 | 0.1429 | 0.3111 | 0.4330 | 0.3447 | 0.1783 | 0.1512 |
| 8 | 0.0909 | 0.2121 | 0.2360 | 0.2057 | 0.1217 | 0.1046 |
| 16 | 0.0022 | 0.0034 | 0.0135 | 0.0102 | 0.0127 | 0.0376 |

Table 8.12: Speedup for MPI_Allgather

| Number of processors / Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| **2** | **0.2500** | **0.3684** | **0.5294** | **0.4407** | **0.2202** | **0.1907** |
| 4 | 0.1538 | 0.3333 | 0.4941 | 0.3525 | 0.2000 | 0.1654 |
| 8 | 0.1333 | 0.2154 | 0.2870 | 0.2115 | 0.1355 | 0.1081 |

Table 8.13: Speedup for MPI_Send and MPI_Recv

| Number of processors / Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| **2** | **0.1333** | **0.3111** | **0.4632** | **0.3998** | **0.1878** | **0.1717** |
| 4 | 0.1176 | 0.2373 | 0.3987 | 0.3385 | 0.1693 | 0.1342 |

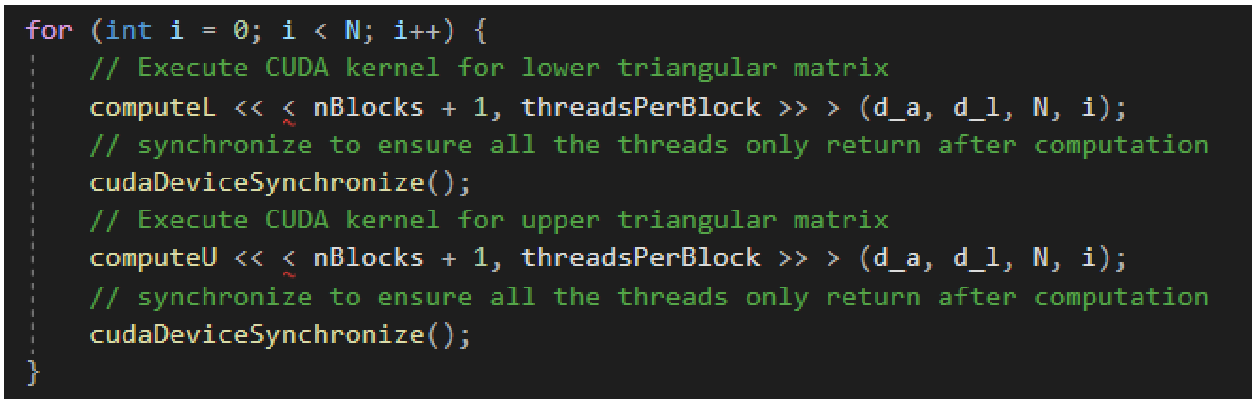
| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | 0.0741 | 0.1400 | 0.1837 | 0.1550 | 0.1117 | 0.0911 |

Table 8.14: Speedup for all MPI parallelization methods with two threads

| MPI methods / Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| MPI_Bcast and MPI_Gather | 0.1667 | 0.3590 | 0.4773 | 0.4194 | 0.1889 | 0.1786 |
| **MPI_Allgather** | **0.2500** | **0.3684** | **0.5294** | **0.4407** | **0.2202** | **0.1907** |
| MPI_Send and MPI_Recv | 0.1333 | 0.3111 | 0.4632 | 0.3998 | 0.1878 | 0.1717 |

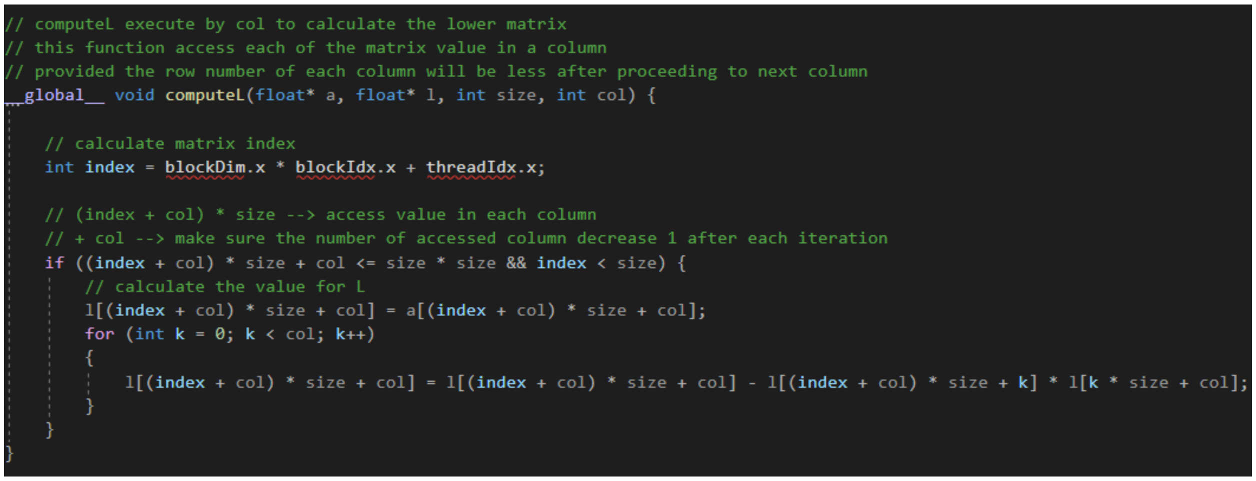
## 8.3 CUDA

### 8.3.1 Implementation



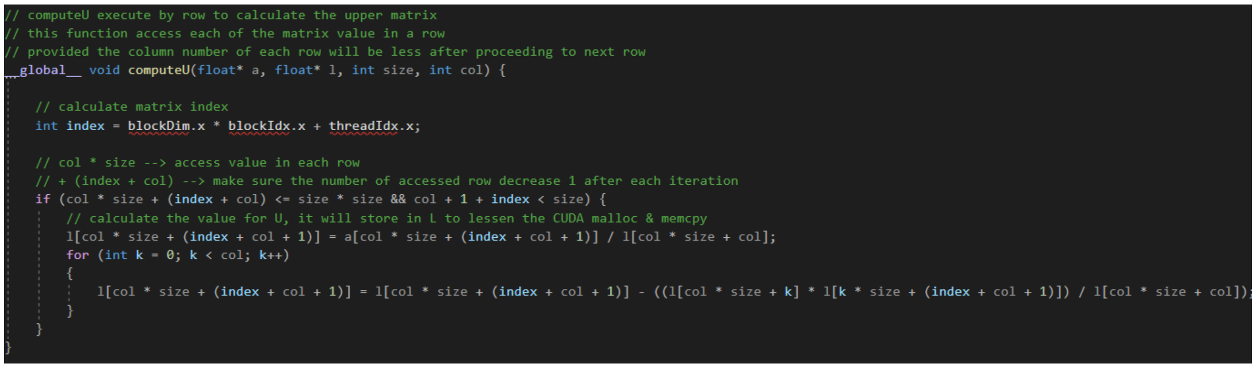
```
for (int i = 0; i < N; i++) {
    // Execute CUDA kernel for lower triangular matrix
    computeL << < nBlocks + 1, threadsPerBlock >> > (d_a, d_l, N, i);
    // synchronize to ensure all the threads only return after computation
    cudaDeviceSynchronize();
    // Execute CUDA kernel for upper triangular matrix
    computeU << < nBlocks + 1, threadsPerBlock >> > (d_a, d_l, N, i);
    // synchronize to ensure all the threads only return after computation
    cudaDeviceSynchronize();
}
```

Figure 8.16: Use of Local Variable to Iterate L and U Kernel Execution



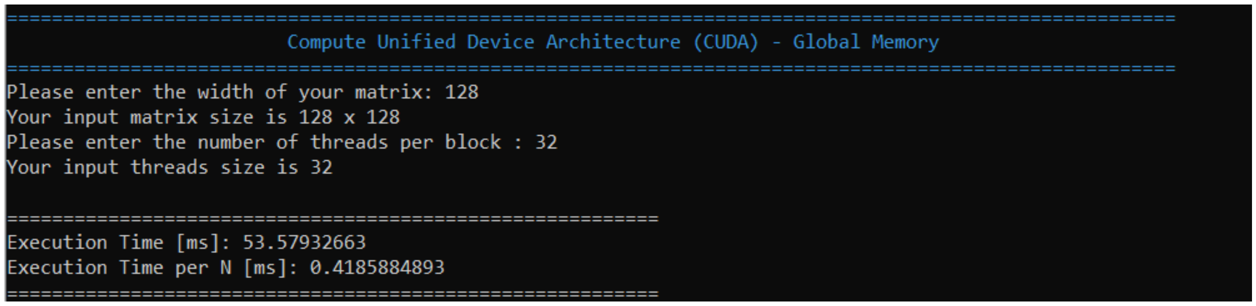
```
// computeL execute by col to calculate the lower matrix
// this function access each of the matrix value in a column
// provided the row number of each column will be less after proceeding to next column
__global__ void computeL(float* a, float* l, int size, int col) {

    // calculate matrix index
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    // (index + col) * size --> access value in each column
    // + col --> make sure the number of accessed column decrease 1 after each iteration
    if ((index + col) * size + col <= size * size && index < size) {
        // calculate the value for L
        l[(index + col) * size + col] = a[(index + col) * size + col];
        for (int k = 0; k < col; k++)
        {
            l[(index + col) * size + col] = l[(index + col) * size + col] - l[(index + col) * size + k] * l[k * size + col];
        }
    }
}
```

Figure 8.17: Each Matrix Index is Mapped with Thread’s Index in Executing L Value



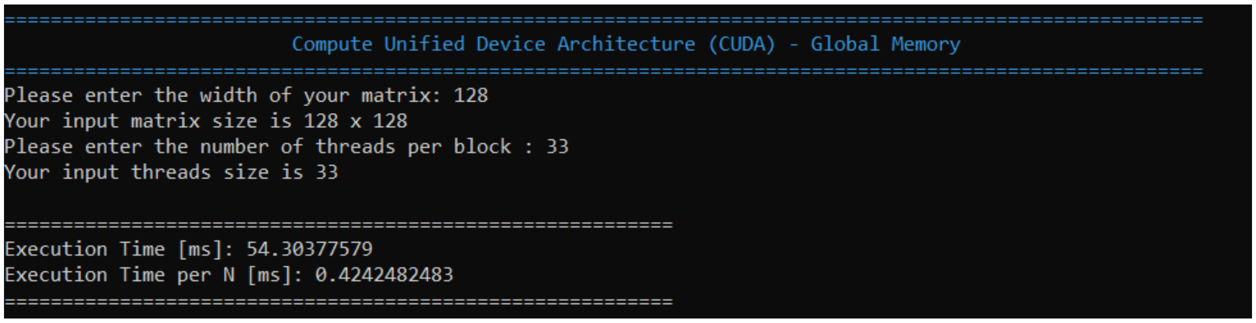
```
// computeU execute by row to calculate the upper matrix
// this function access each of the matrix value in a row
// provided the column number of each row will be less after proceeding to next row
__global__ void computeU(float* a, float* l, int size, int col) {

    // calculate matrix index
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    // col * size --> access value in each row
    // + (index + col) --> make sure the number of accessed row decrease 1 after each iteration
    if (col * size + (index + col) <= size * size && col + 1 + index < size) {
        // calculate the value for U, it will store in L to lessen the CUDA malloc & memcpy
        l[col * size + (index + col + 1)] = a[col * size + (index + col + 1)] / l[col * size + col];
        for (int k = 0; k < col; k++)
        {
            l[col * size + (index + col + 1)] = l[col * size + (index + col + 1)] - ((l[col * size + k] * l[k * size + (index + col + 1)]) / l[col * size + col]);
        }
    }
}
```

Figure 8.18: Each Matrix Index is Mapped with Thread’s Index in Executing U Value



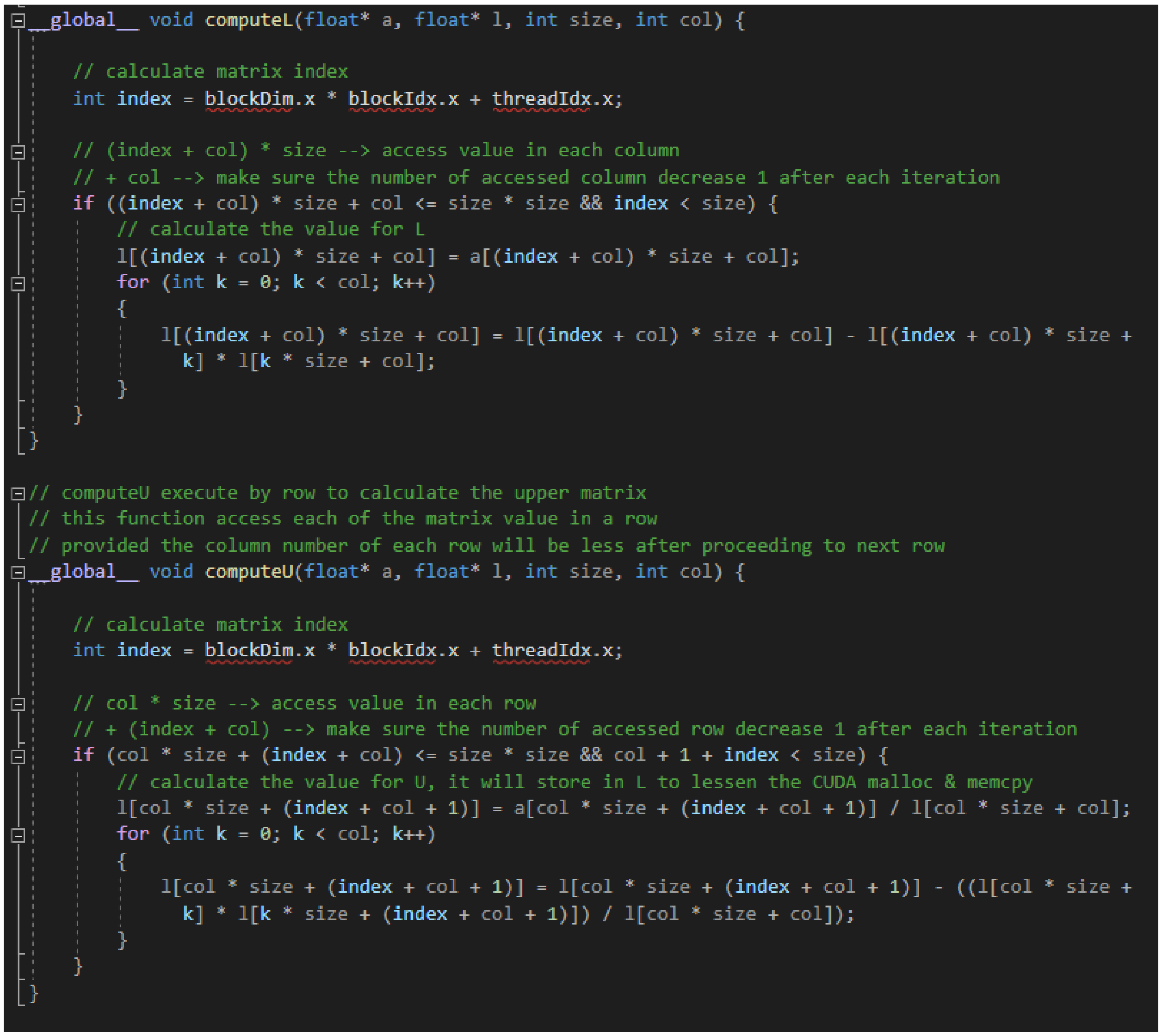
```
===============================================================================================
                    Compute Unified Device Architecture (CUDA) - Global Memory
===============================================================================================
Please enter the width of your matrix: 128
Your input matrix size is 128 x 128
Please enter the number of threads per block : 32
Your input threads size is 32

===================================================
Execution Time [ms]: 53.57932663
Execution Time per N [ms]: 0.4185884893
===================================================
```

Figure 8.19: Execution Time for Matrix Size 128 with 32 Threads using Global Memory



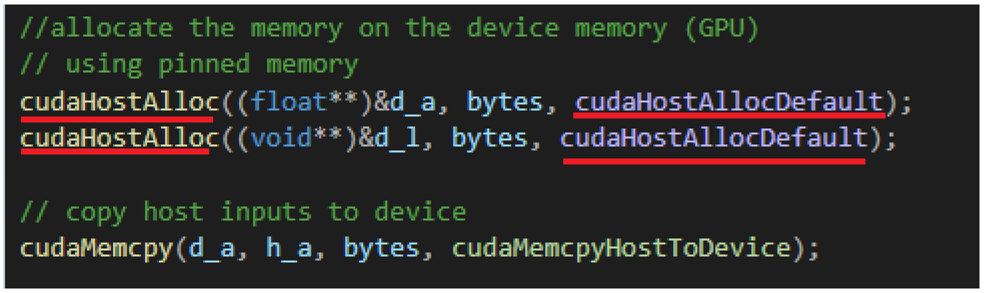
```
==========================================================================================
                    Compute Unified Device Architecture (CUDA) - Global Memory
==========================================================================================
Please enter the width of your matrix: 128
Your input matrix size is 128 x 128
Please enter the number of threads per block : 33
Your input threads size is 33

========================================================
Execution Time [ms]: 54.30377579
Execution Time per N [ms]: 0.4242482483
========================================================
```

Figure 8.20: Execution Time for Matrix Size 128 with 33 Threads using Global Memory



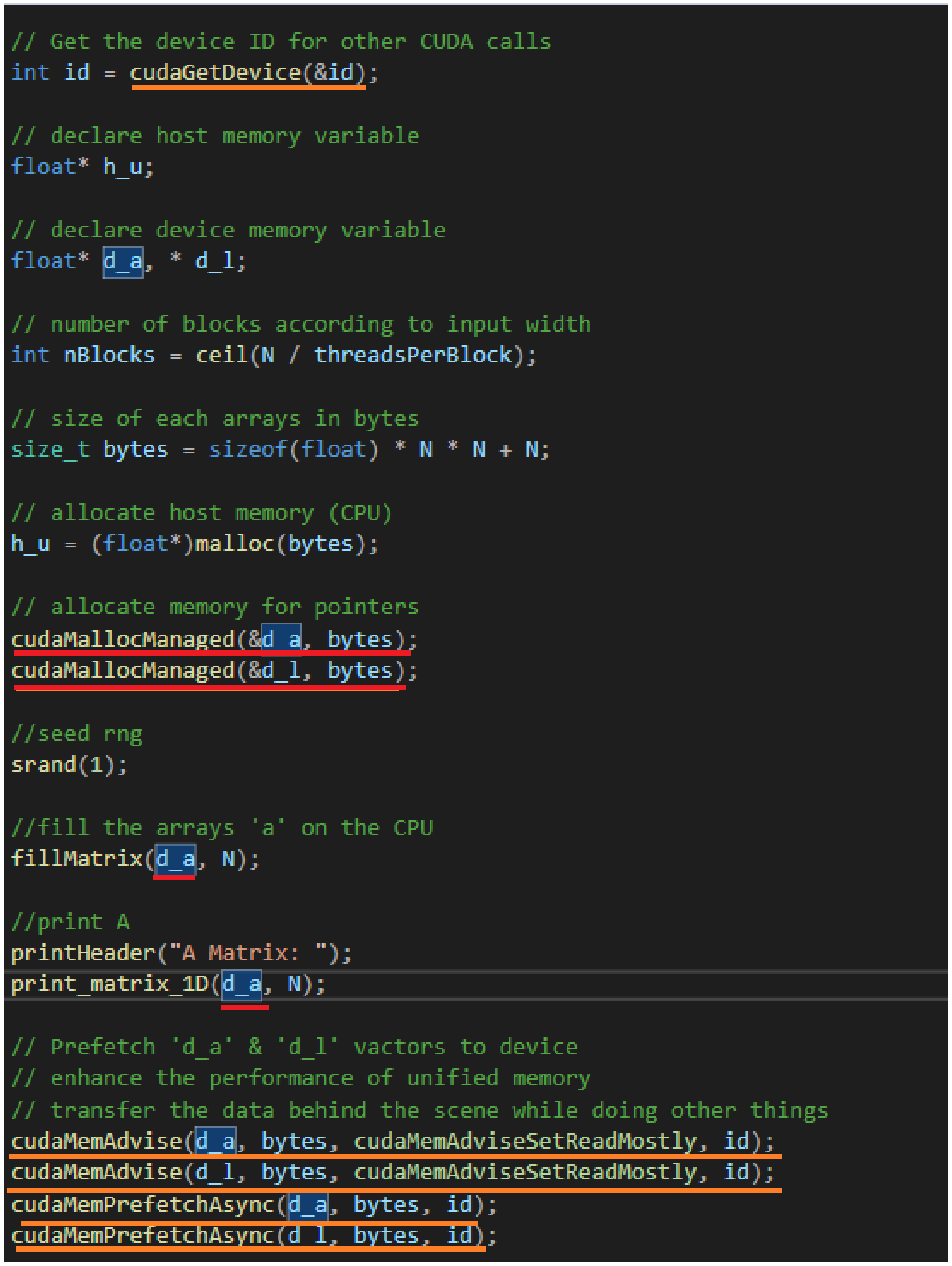
```
__global__ void computeL(float* a, float* l, int size, int col) {

    // calculate matrix index
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    // (index + col) * size --> access value in each column
    // + col --> make sure the number of accessed column decrease 1 after each iteration
    if ((index + col) * size + col <= size * size && index < size) {
        // calculate the value for L
        l[(index + col) * size + col] = a[(index + col) * size + col];
        for (int k = 0; k < col; k++)
        {
            l[(index + col) * size + col] = l[(index + col) * size + col] - l[(index + col) * size +
              k] * l[k * size + col];
        }
    }
}

// computeU execute by row to calculate the upper matrix
// this function access each of the matrix value in a row
// provided the column number of each row will be less after proceeding to next row
__global__ void computeU(float* a, float* l, int size, int col) {

    // calculate matrix index
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    // col * size --> access value in each row
    // + (index + col) --> make sure the number of accessed row decrease 1 after each iteration
    if (col * size + (index + col) <= size * size && col + 1 + index < size) {
        // calculate the value for U, it will store in L to lessen the CUDA malloc & memcpy
        l[col * size + (index + col + 1)] = a[col * size + (index + col + 1)] / l[col * size + col];
        for (int k = 0; k < col; k++)
        {
            l[col * size + (index + col + 1)] = l[col * size + (index + col + 1)] - ((l[col * size +
              k] * l[k * size + (index + col + 1)]) / l[col * size + col]);
        }
    }
}
```

Figure 8.21: Use of global memory in LU decomposition

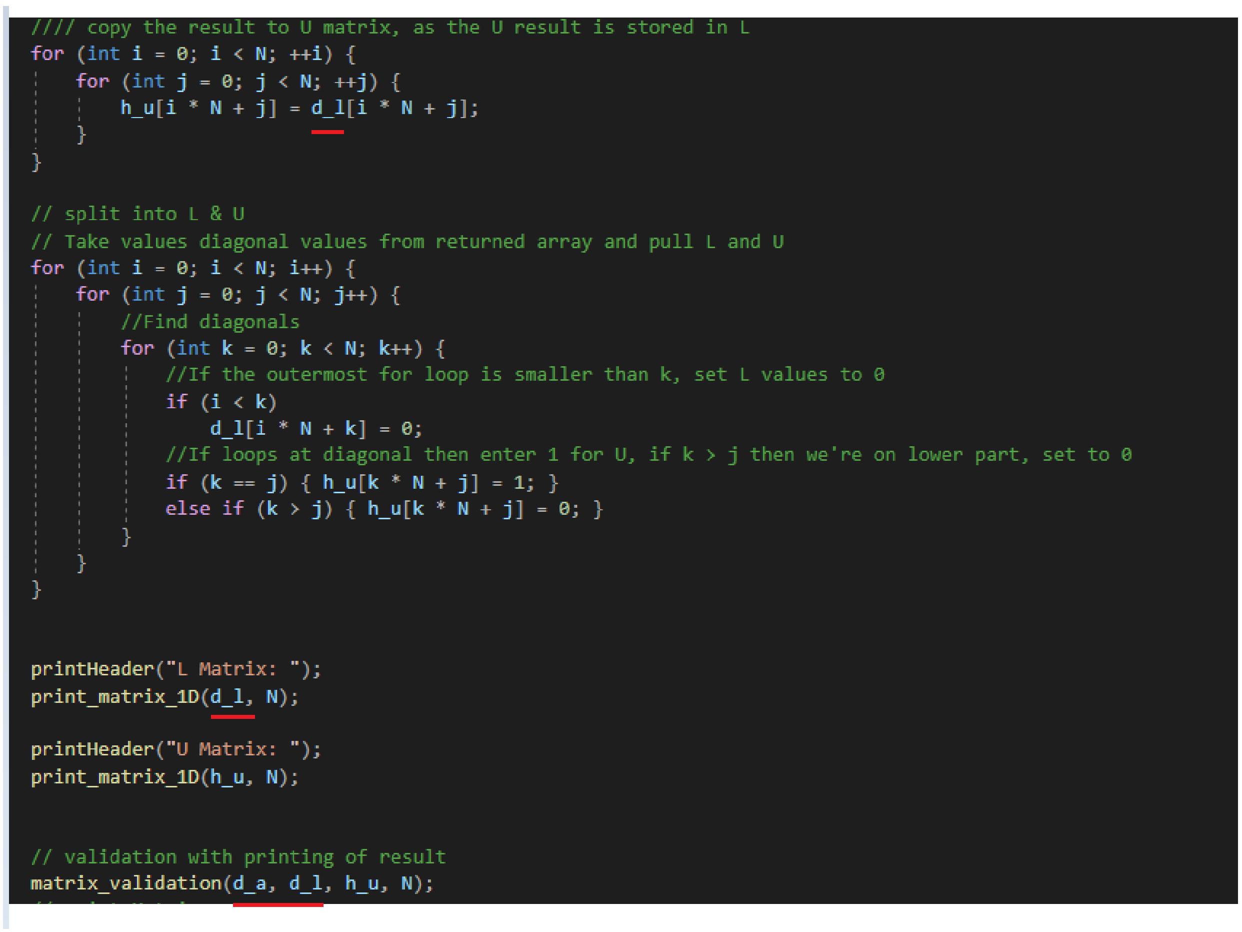


```
//allocate the memory on the device memory (GPU)
// using pinned memory
cudaHostAlloc((float**)&d_a, bytes, cudaHostAllocDefault);
cudaHostAlloc((void**)&d_l, bytes, cudaHostAllocDefault);

// copy host inputs to device
cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
```

Figure 8.22: Use of pinned memory in LU decomposition

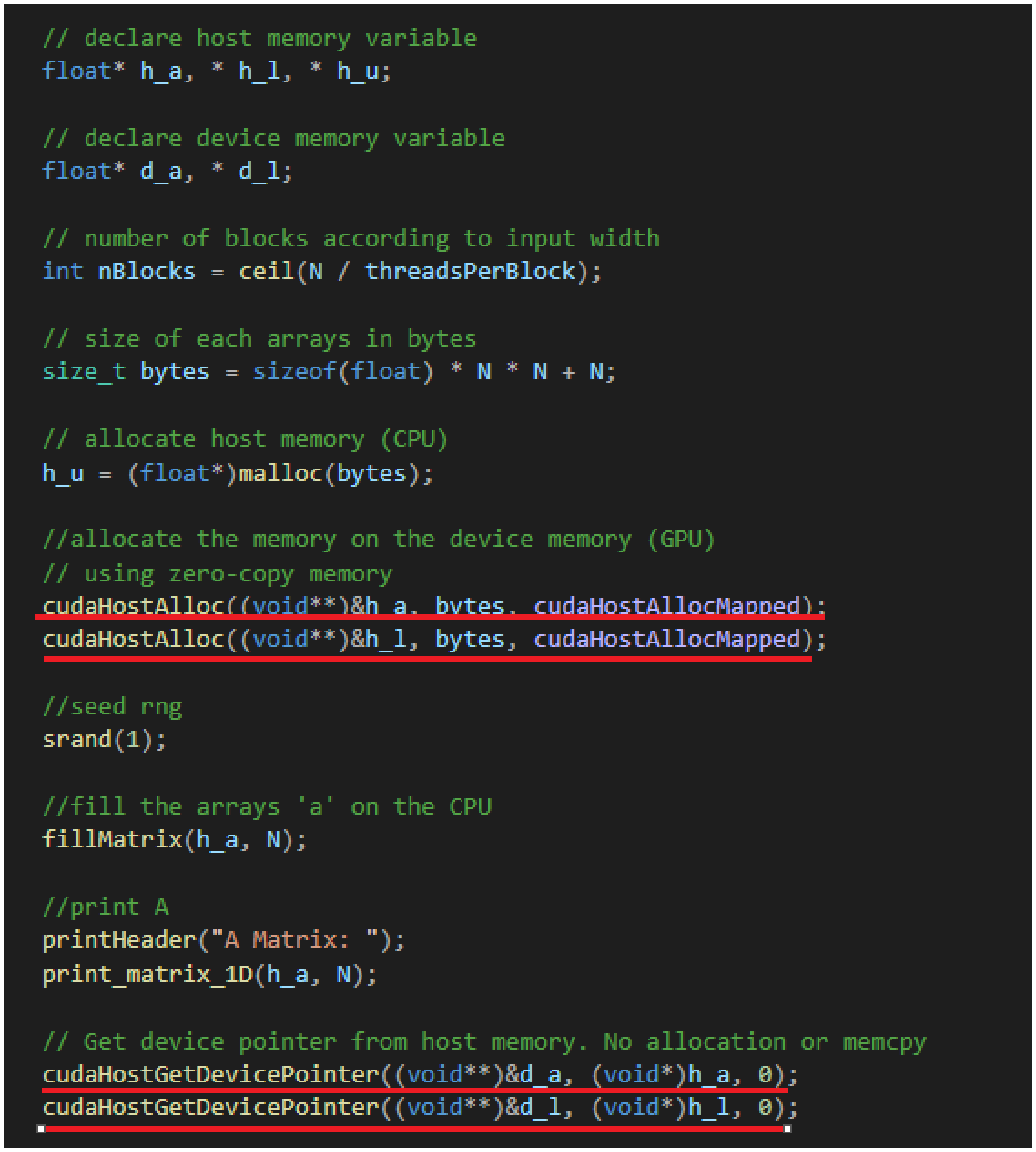


```
// Get the device ID for other CUDA calls
int id = cudaGetDevice(&id);

// declare host memory variable
float* h_u;

// declare device memory variable
float* d_a, * d_l;

// number of blocks according to input width
int nBlocks = ceil(N / threadsPerBlock);

// size of each arrays in bytes
size_t bytes = sizeof(float) * N * N + N;

// allocate host memory (CPU)
h_u = (float*)malloc(bytes);

// allocate memory for pointers
cudaMallocManaged(&d_a, bytes);
cudaMallocManaged(&d_l, bytes);

//seed rng
srand(1);

//fill the arrays 'a' on the CPU
fillMatrix(d_a, N);

//print A
printHeader("A Matrix: ");
print_matrix_1D(d_a, N);

// Prefetch 'd_a' & 'd_l' vactors to device
// enhance the performance of unified memory
// transfer the data behind the scene while doing other things
cudaMemAdvise(d_a, bytes, cudaMemAdviseSetReadMostly, id);
cudaMemAdvise(d_l, bytes, cudaMemAdviseSetReadMostly, id);
cudaMemPrefetchAsync(d_a, bytes, id);
cudaMemPrefetchAsync(d_l, bytes, id);
```



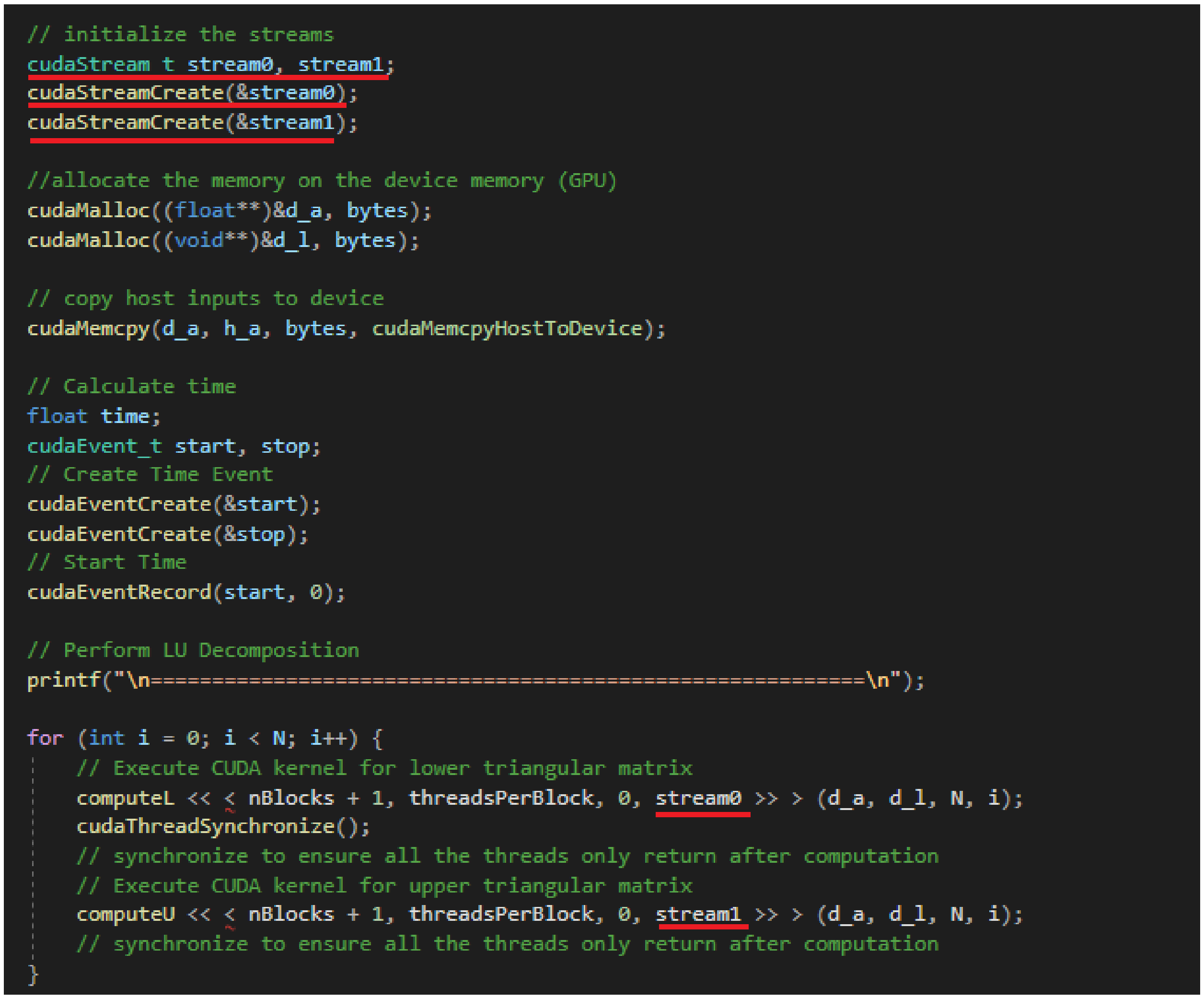
```
//// copy the result to U matrix, as the U result is stored in L
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        h_u[i * N + j] = d_l[i * N + j];
    }
}

// split into L & U
// Take values diagonal values from returned array and pull L and U
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        //Find diagonals
        for (int k = 0; k < N; k++) {
            //If the outermost for loop is smaller than k, set L values to 0
            if (i < k)
                d_l[i * N + k] = 0;
            //If loops at diagonal then enter 1 for U, if k > j then we're on lower part, set to 0
            if (k == j) { h_u[k * N + j] = 1; }
            else if (k > j) { h_u[k * N + j] = 0; }
        }
    }
}


printHeader("L Matrix: ");
print_matrix_1D(d_l, N);

printHeader("U Matrix: ");
print_matrix_1D(h_u, N);


// validation with printing of result
matrix_validation(d_a, d_l, h_u, N);
```

Figure 8.23: Use of unified memory in LU decomposition, red color indicate utilization of unified memory while orange color indicate performance tuning



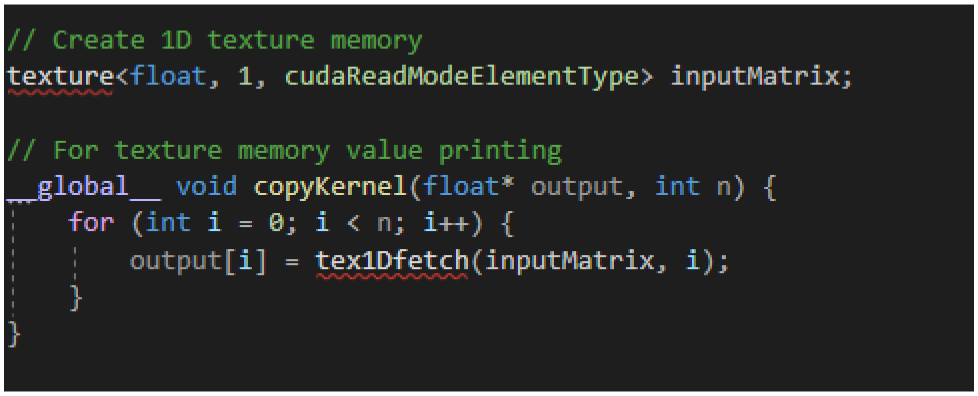
```
// declare host memory variable
float* h_a, * h_l, * h_u;

// declare device memory variable
float* d_a, * d_l;

// number of blocks according to input width
int nBlocks = ceil(N / threadsPerBlock);

// size of each arrays in bytes
size_t bytes = sizeof(float) * N * N + N;

// allocate host memory (CPU)
h_u = (float*)malloc(bytes);

//allocate the memory on the device memory (GPU)
// using zero-copy memory
cudaHostAlloc((void**)&h_a, bytes, cudaHostAllocMapped);
cudaHostAlloc((void**)&h_l, bytes, cudaHostAllocMapped);

//seed rng
srand(1);

//fill the arrays 'a' on the CPU
fillMatrix(h_a, N);

//print A
printHeader("A Matrix: ");
print_matrix_1D(h_a, N);

// Get device pointer from host memory. No allocation or memcpy
cudaHostGetDevicePointer((void**)&d_a, (void*)h_a, 0);
cudaHostGetDevicePointer((void**)&d_l, (void*)h_l, 0);
```

Figure 8.24: Use of zero-copy host code in LU decomposition

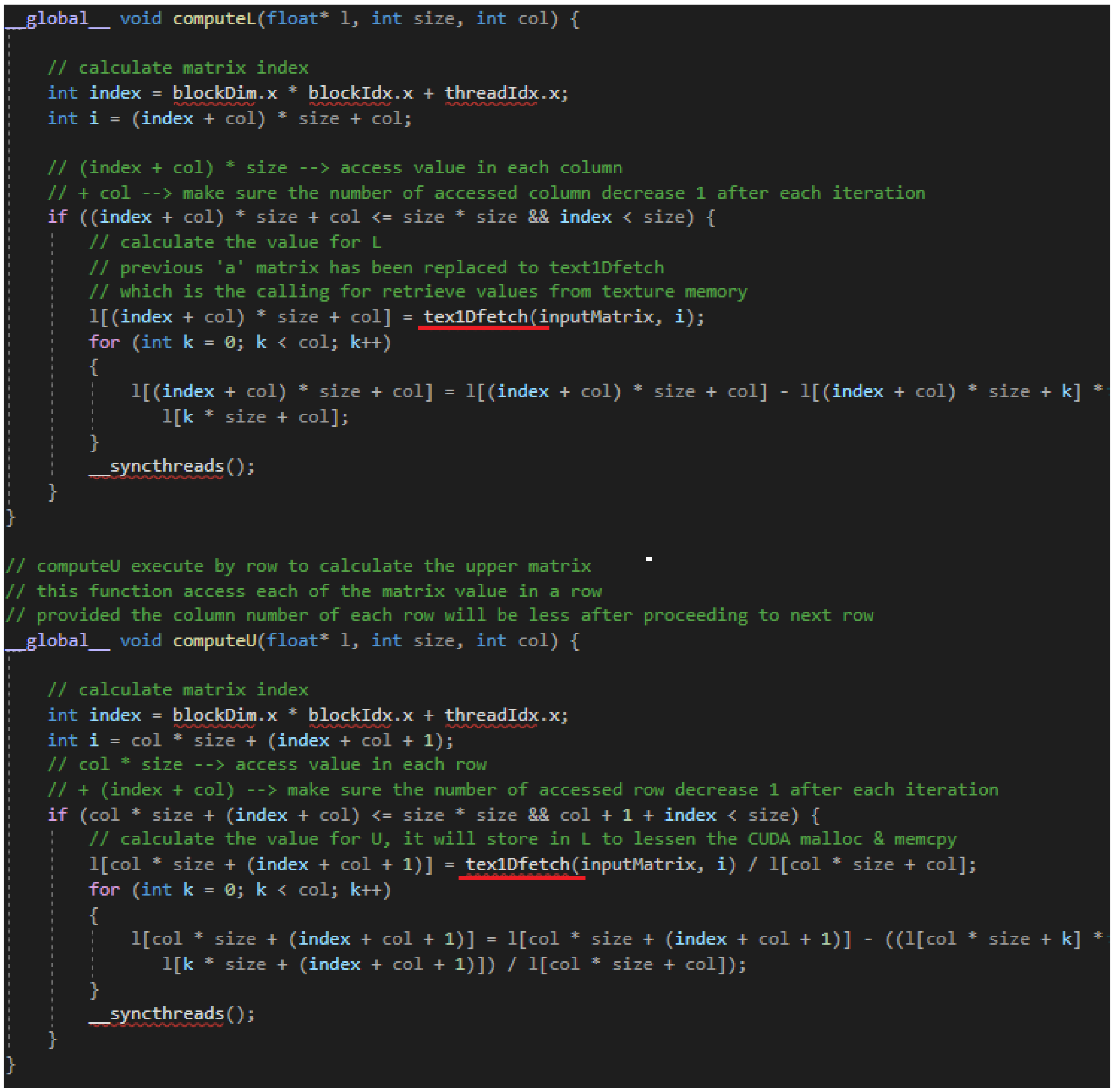


```
// initialize the streams
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

//allocate the memory on the device memory (GPU)
cudaMalloc((float**)&d_a, bytes);
cudaMalloc((void**)&d_l, bytes);

// copy host inputs to device
cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);

// Calculate time
float time;
cudaEvent_t start, stop;
// Create Time Event
cudaEventCreate(&start);
cudaEventCreate(&stop);
// Start Time
cudaEventRecord(start, 0);

// Perform LU Decomposition
printf("\n============================================================\n");

for (int i = 0; i < N; i++) {
    // Execute CUDA kernel for lower triangular matrix
    computeL << < nBlocks + 1, threadsPerBlock, 0, stream0 >> > (d_a, d_l, N, i);
    cudaThreadSynchronize();
    // synchronize to ensure all the threads only return after computation
    // Execute CUDA kernel for upper triangular matrix
    computeU << < nBlocks + 1, threadsPerBlock, 0, stream1 >> > (d_a, d_l, N, i);
    // synchronize to ensure all the threads only return after computation
}
```

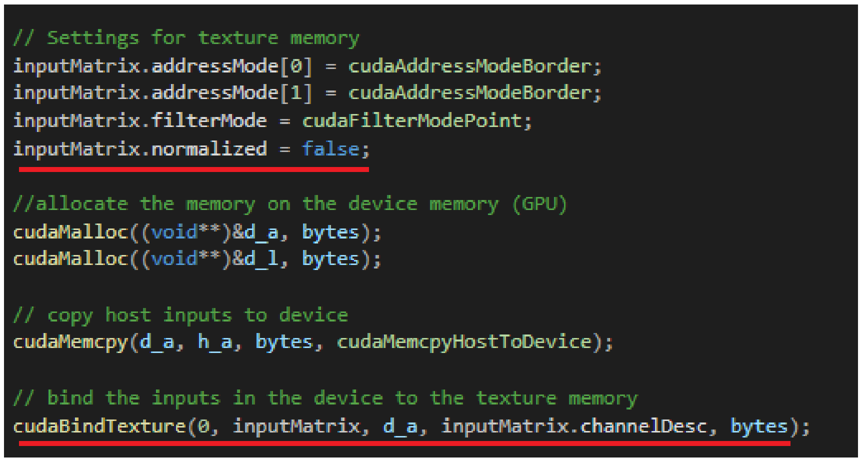
Figure 8.25: Use of concurrent kernel execution in LU decomposition



```
// Create 1D texture memory
texture<float, 1, cudaReadModeElementType> inputMatrix;

// For texture memory value printing
__global__ void copyKernel(float* output, int n) {
    for (int i = 0; i < n; i++) {
        output[i] = tex1Dfetch(inputMatrix, i);
    }
}
```



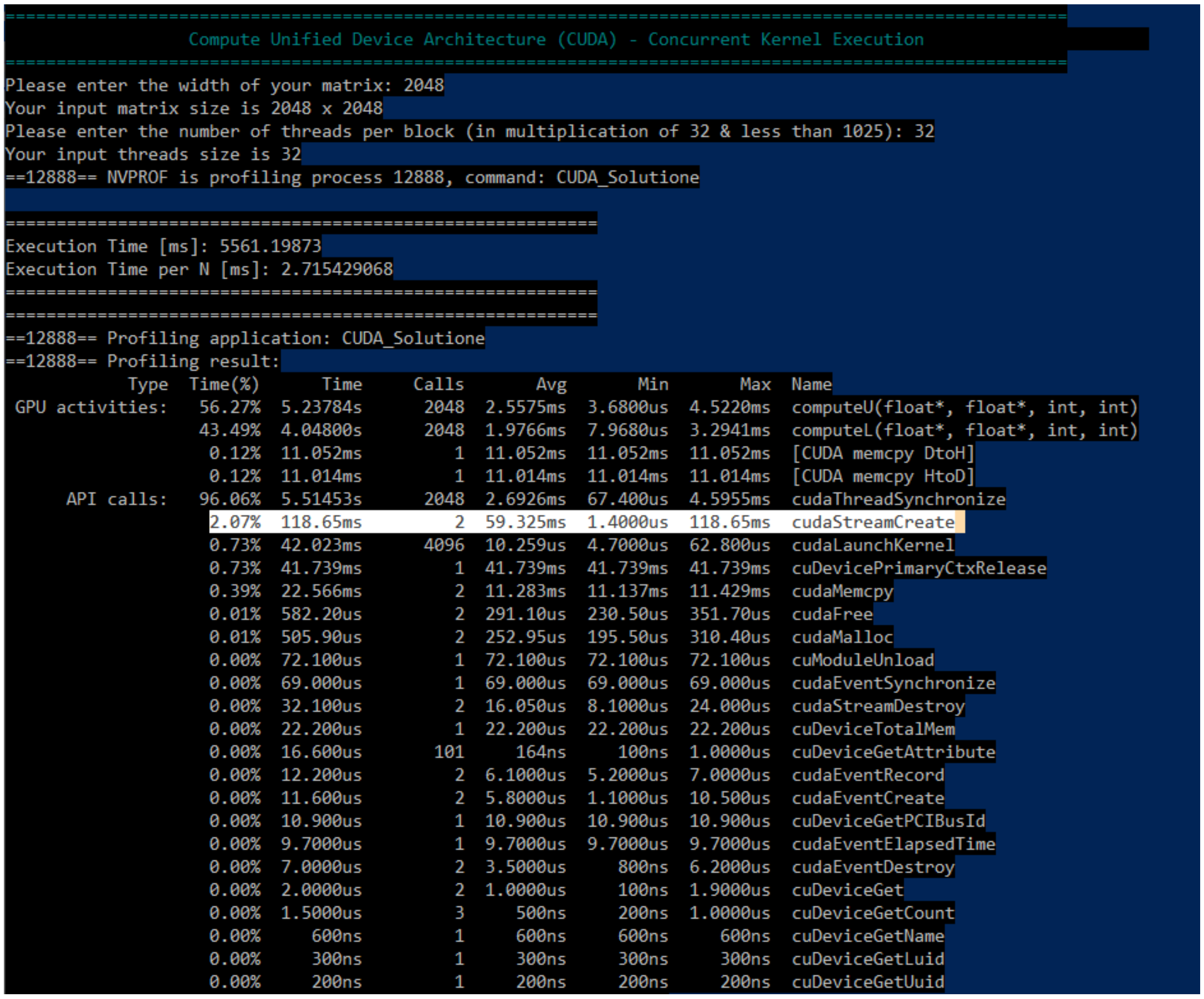
```
__global__ void computeL(float* l, int size, int col) {

    // calculate matrix index
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    int i = (index + col) * size + col;

    // (index + col) * size --> access value in each column
    // + col --> make sure the number of accessed column decrease 1 after each iteration
    if ((index + col) * size + col <= size * size && index < size) {
        // calculate the value for L
        // previous 'a' matrix has been replaced to text1Dfetch
        // which is the calling for retrieve values from texture memory
        l[(index + col) * size + col] = tex1Dfetch(inputMatrix, i);
        for (int k = 0; k < col; k++)
        {
            l[(index + col) * size + col] = l[(index + col) * size + col] - l[(index + col) * size + k] *
               l[k * size + col];
        }
        __syncthreads();
    }
}

// computeU execute by row to calculate the upper matrix
// this function access each of the matrix value in a row
// provided the column number of each row will be less after proceeding to next row
__global__ void computeU(float* l, int size, int col) {

    // calculate matrix index
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    int i = col * size + (index + col + 1);
    // col * size --> access value in each row
    // + (index + col) --> make sure the number of accessed row decrease 1 after each iteration
    if (col * size + (index + col) <= size * size && col + 1 + index < size) {
        // calculate the value for U, it will store in L to lessen the CUDA malloc & memcpy
        l[col * size + (index + col + 1)] = tex1Dfetch(inputMatrix, i) / l[col * size + col];
        for (int k = 0; k < col; k++)
        {
            l[col * size + (index + col + 1)] = l[col * size + (index + col + 1)] - ((l[col * size + k] *
               l[k * size + (index + col + 1)]) / l[col * size + col]);
        }
        __syncthreads();
    }
}
```



```
// Settings for texture memory
inputMatrix.addressMode[0] = cudaAddressModeBorder;
inputMatrix.addressMode[1] = cudaAddressModeBorder;
inputMatrix.filterMode = cudaFilterModePoint;
inputMatrix.normalized = false;

//allocate the memory on the device memory (GPU)
cudaMalloc((void**)&d_a, bytes);
cudaMalloc((void**)&d_l, bytes);

// copy host inputs to device
cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);

// bind the inputs in the device to the texture memory
cudaBindTexture(0, inputMatrix, d_a, inputMatrix.channelDesc, bytes);
```

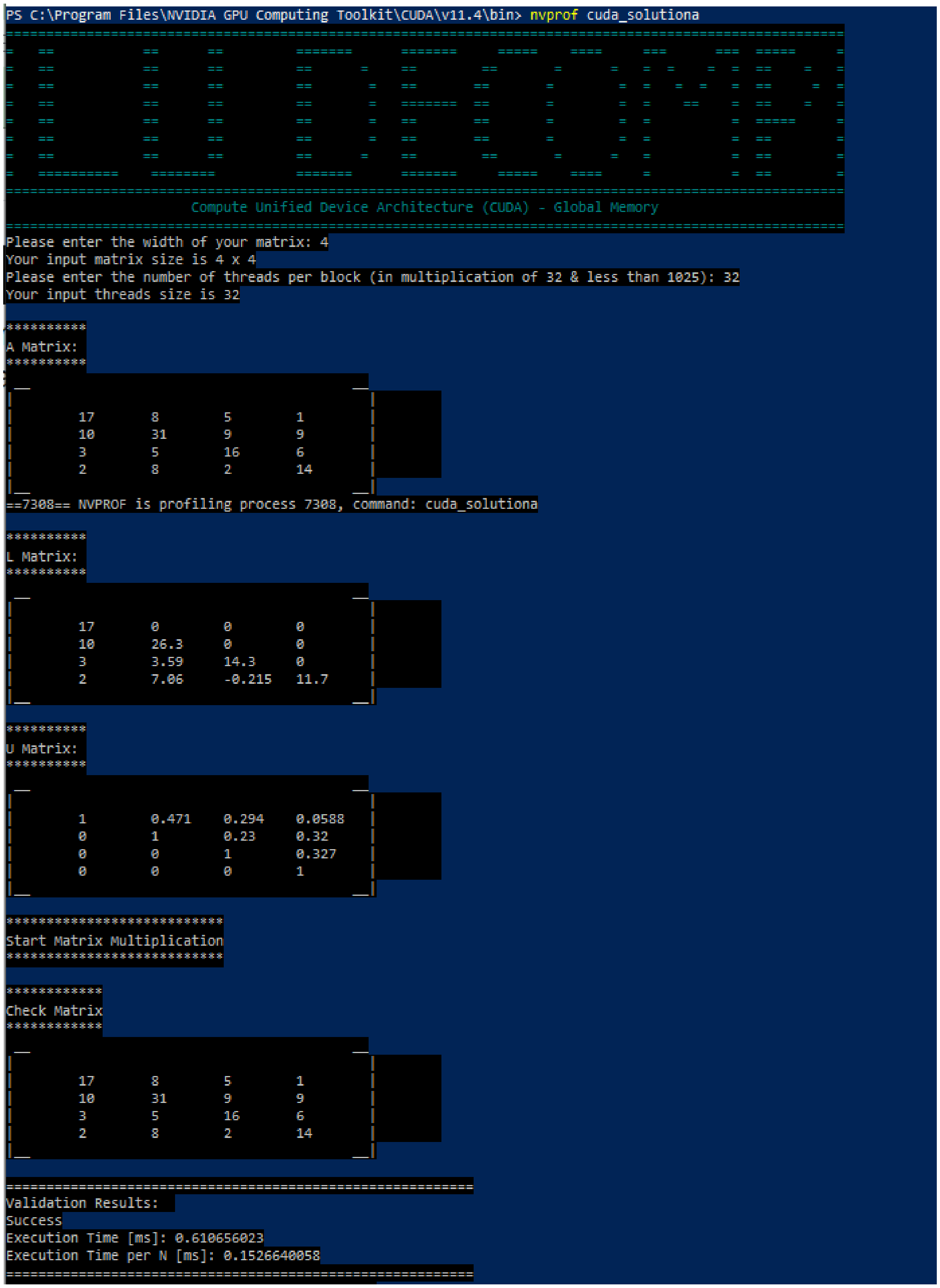
Figure 8.26: Use of texture memory in LU decomposition

### 8.3.2 Validation Results



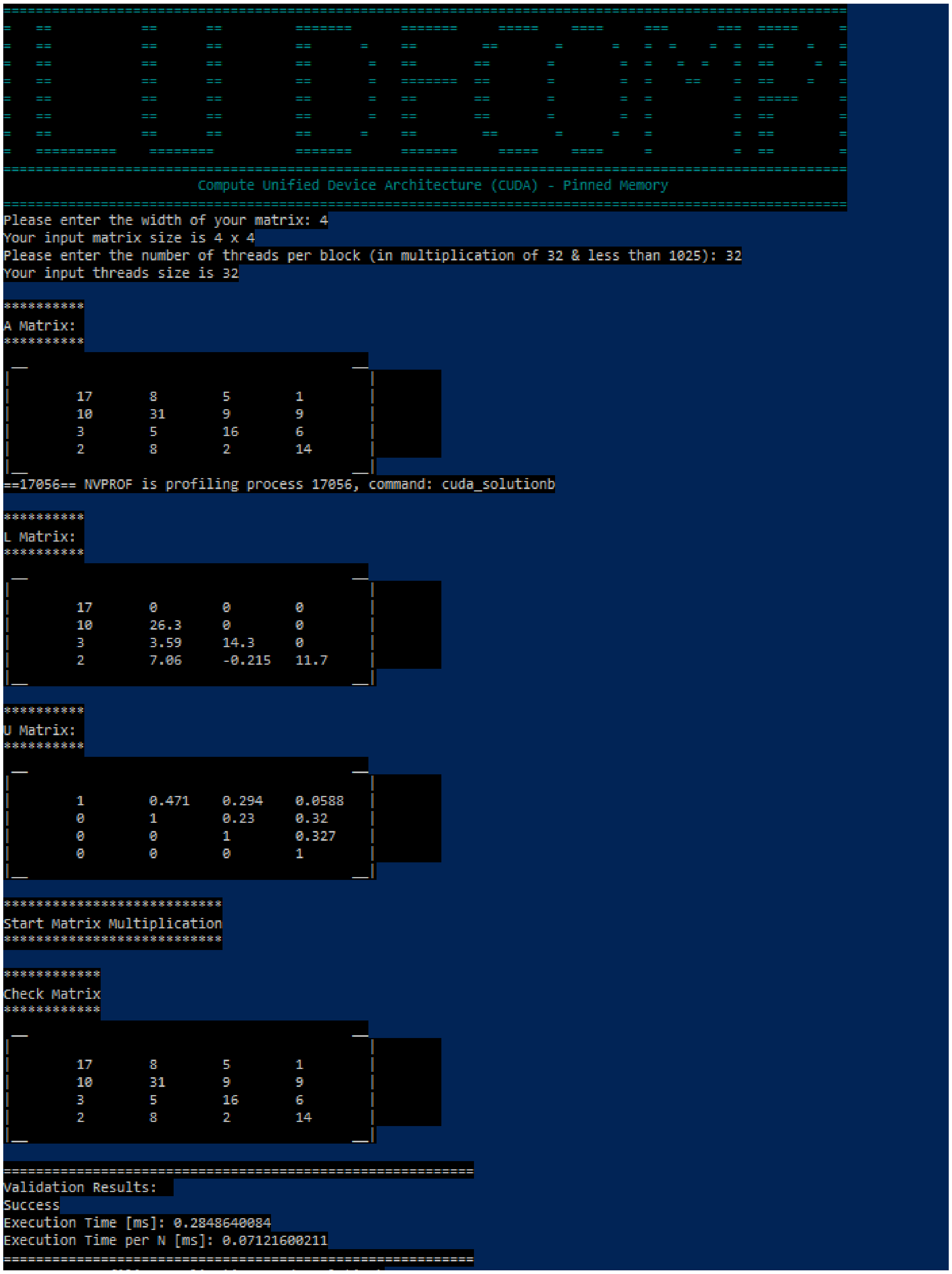
```
========================================================================================================
                  Compute Unified Device Architecture (CUDA) - Concurrent Kernel Execution
========================================================================================================
Please enter the width of your matrix: 2048
Your input matrix size is 2048 x 2048
Please enter the number of threads per block (in multiplication of 32 & less than 1025): 32
Your input threads size is 32
==12888== NVPROF is profiling process 12888, command: CUDA_Solutione

==========================================================
Execution Time [ms]: 5561.19873
Execution Time per N [ms]: 2.715429068
==========================================================
==========================================================
==12888== Profiling application: CUDA_Solutione
==12888== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.27%  5.23784s      2048  2.5575ms  3.6800us  4.5220ms  computeU(float*, float*, int, int)
                   43.49%  4.04800s      2048  1.9766ms  7.9680us  3.2941ms  computeL(float*, float*, int, int)
                    0.12%  11.052ms         1  11.052ms  11.052ms  11.052ms  [CUDA memcpy DtoH]
                    0.12%  11.014ms         1  11.014ms  11.014ms  11.014ms  [CUDA memcpy HtoD]
      API calls:   96.06%  5.51453s      2048  2.6926ms  67.400us  4.5955ms  cudaThreadSynchronize
                    2.07%  118.65ms         2  59.325ms  1.4000us  118.65ms  cudaStreamCreate
                    0.73%  42.023ms      4096  10.259us  4.7000us  62.800us  cudaLaunchKernel
                    0.73%  41.739ms         1  41.739ms  41.739ms  41.739ms  cuDevicePrimaryCtxRelease
                    0.39%  22.566ms         2  11.283ms  11.137ms  11.429ms  cudaMemcpy
                    0.01%  582.20us         2  291.10us  230.50us  351.70us  cudaFree
                    0.01%  505.90us         2  252.95us  195.50us  310.40us  cudaMalloc
                    0.00%  72.100us         1  72.100us  72.100us  72.100us  cuModuleUnload
                    0.00%  69.000us         1  69.000us  69.000us  69.000us  cudaEventSynchronize
                    0.00%  32.100us         2  16.050us  8.1000us  24.000us  cudaStreamDestroy
                    0.00%  22.200us         1  22.200us  22.200us  22.200us  cuDeviceTotalMem
                    0.00%  16.600us       101     164ns     100ns  1.0000us  cuDeviceGetAttribute
                    0.00%  12.200us         2  6.1000us  5.2000us  7.0000us  cudaEventRecord
                    0.00%  11.600us         2  5.8000us  1.1000us  10.500us  cudaEventCreate
                    0.00%  10.900us         1  10.900us  10.900us  10.900us  cuDeviceGetPCIBusId
                    0.00%  9.7000us         1  9.7000us  9.7000us  9.7000us  cudaEventElapsedTime
                    0.00%  7.0000us         2  3.5000us     800ns  6.2000us  cudaEventDestroy
                    0.00%  2.0000us         2  1.0000us     100ns  1.9000us  cuDeviceGet
                    0.00%  1.5000us         3     500ns     200ns  1.0000us  cuDeviceGetCount
                    0.00%     600ns         1     600ns     600ns     600ns  cuDeviceGetName
                    0.00%     300ns         1     300ns     300ns     300ns  cuDeviceGetLuid
                    0.00%     200ns         1     200ns     200ns     200ns  cuDeviceGetUuid
```

Figure 8.27: NVIDIA profiler showing execution performance of Concurrent Kernel Execution with matrix size 2048 and threads 32.



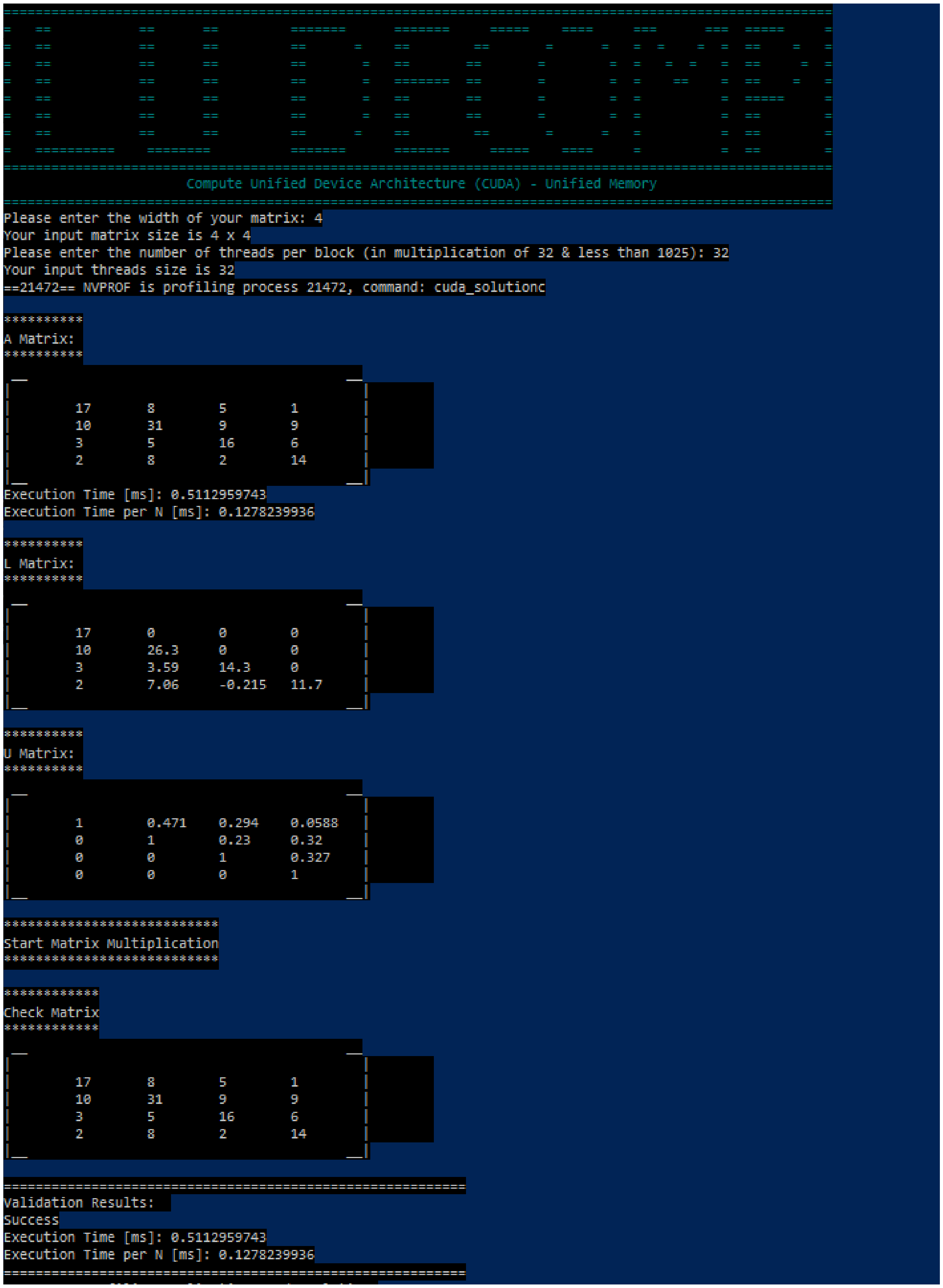
```
PS C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.4\bin> nvprof cuda_solutiona
========================================================================================================
=   ==           ==      ==         =======      =======     =====    ====     ===      ===  =====     =
=   ==           ==      ==         ==      =    ==        ==       =      =   =  =    =  =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =         = =   =  =   =  ==     =  =
=   ==           ==      ==         ==       =   ======= ==       =         = =    ==    =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =         = =          =  =====     =
=   ==           ==      ==         ==       =   ==       ==       =         = =          =  ==        =
=   ==           ==      ==         ==      =    ==        ==       =      =   =          =  ==        =
=   ==========    ========          =======      =======     =====    ====     =          =  ==        =
========================================================================================================
                       Compute Unified Device Architecture (CUDA) - Global Memory
========================================================================================================
Please enter the width of your matrix: 4
Your input matrix size is 4 x 4
Please enter the number of threads per block (in multiplication of 32 & less than 1025): 32
Your input threads size is 32

**********
A Matrix:
**********
 __                                        __
|                                            |
|        17       8        5        1        |
|        10       31       9        9        |
|        3        5        16       6        |
|        2        8        2        14       |
|__                                        __|
==7308== NVPROF is profiling process 7308, command: cuda_solutiona

**********
L Matrix:
**********
 __                                        __
|                                            |
|        17       0        0        0        |
|        10       26.3     0        0        |
|        3        3.59     14.3     0        |
|        2        7.06     -0.215   11.7     |
|__                                        __|

**********
U Matrix:
**********
 __                                        __
|                                            |
|        1        0.471    0.294    0.0588   |
|        0        1        0.23     0.32     |
|        0        0        1        0.327    |
|        0        0        0        1        |
|__                                        __|

***************************
Start Matrix Multiplication
***************************

************
Check Matrix
************
 __                                        __
|                                            |
|        17       8        5        1        |
|        10       31       9        9        |
|        3        5        16       6        |
|        2        8        2        14       |
|__                                        __|

==========================================================
Validation Results:
Success
Execution Time [ms]: 0.610656023
Execution Time per N [ms]: 0.1526640058
==========================================================
```

Figure 8.28: Validation result for CUDA global memory solution (Solution A)



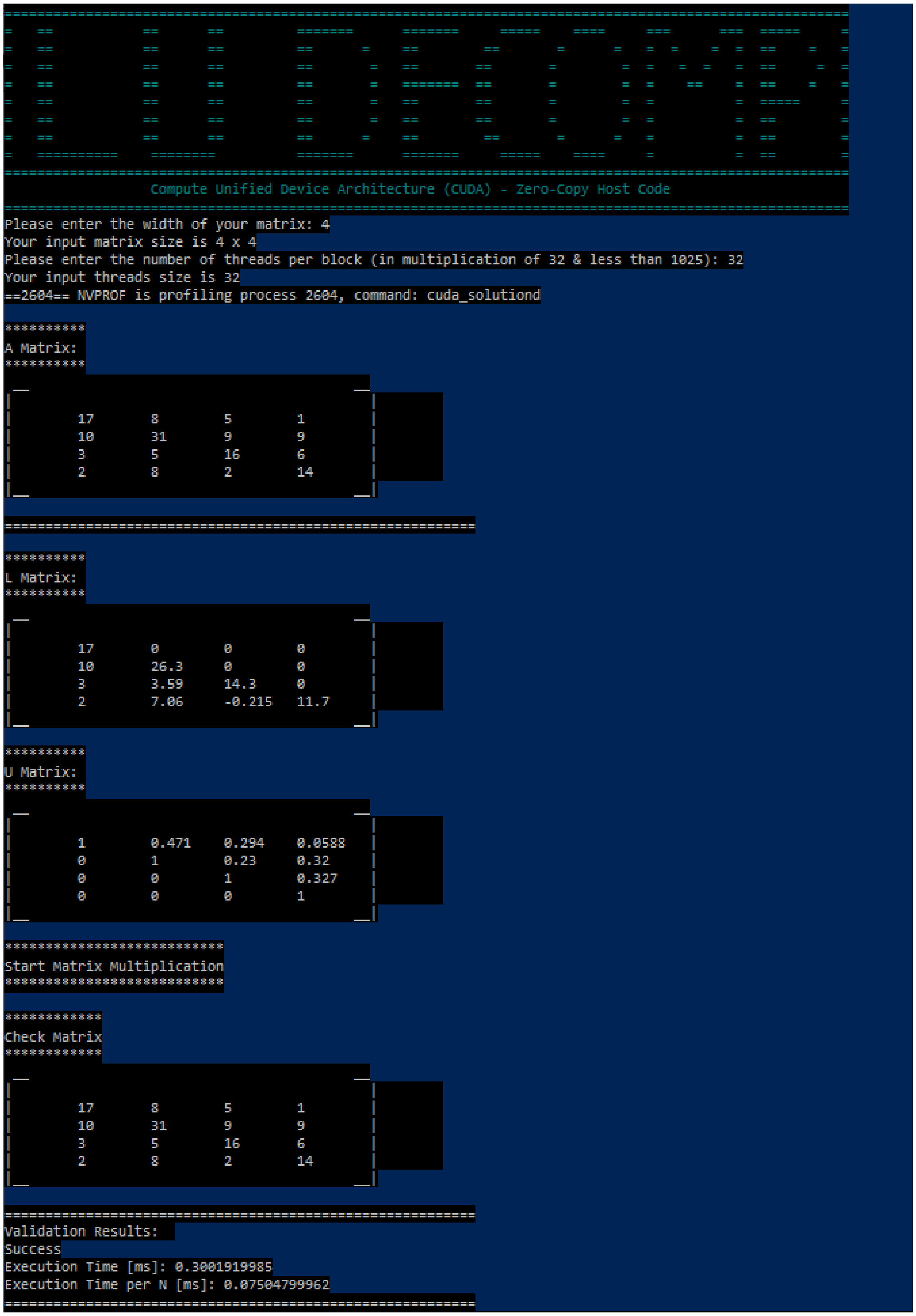
```
========================================================================================================
=   ==           ==      ==         =======      =======     =====    ====     ===      ===  =====     =
=   ==           ==      ==         ==       =   ==        ==       =       =  =  =    =   = ==     =  =
=   ==           ==      ==         ==        =  ==       ==       =         = =   =   =   = ==      = =
=   ==           ==      ==         ==        =  =======  ==       =         = =    ==     = ==     =  =
=   ==           ==      ==         ==        =  ==       ==       =         = =           = =====     =
=   ==           ==      ==         ==        =  ==       ==       =         = =           = ==        =
=   ==           ==      ==         ==       =   ==        ==       =       =  =           = ==        =
=   ==========    ========          =======      =======     =====    ====     =           = ==        =
========================================================================================================
                        Compute Unified Device Architecture (CUDA) - Pinned Memory
========================================================================================================
Please enter the width of your matrix: 4
Your input matrix size is 4 x 4
Please enter the number of threads per block (in multiplication of 32 & less than 1025): 32
Your input threads size is 32

**********
A Matrix:
**********
 __                                          __
|                                              |
|        17       8        5        1          |
|        10       31       9        9          |
|        3        5        16       6          |
|        2        8        2        14         |
|__                                          __|
==17056== NVPROF is profiling process 17056, command: cuda_solutionb

**********
L Matrix:
**********
 __                                          __
|                                              |
|        17       0        0        0          |
|        10       26.3     0        0          |
|        3        3.59     14.3     0          |
|        2        7.06     -0.215   11.7       |
|__                                          __|

**********
U Matrix:
**********
 __                                          __
|                                              |
|        1        0.471    0.294    0.0588     |
|        0        1        0.23     0.32       |
|        0        0        1        0.327      |
|        0        0        0        1          |
|__                                          __|

***************************
Start Matrix Multiplication
***************************

************
Check Matrix
************
 __                                          __
|                                              |
|        17       8        5        1          |
|        10       31       9        9          |
|        3        5        16       6          |
|        2        8        2        14         |
|__                                          __|

==========================================================
Validation Results:
Success
Execution Time [ms]: 0.2848640084
Execution Time per N [ms]: 0.07121600211
==========================================================
```

Figure 8.29: Validation result for CUDA pinned memory solution (Solution B)



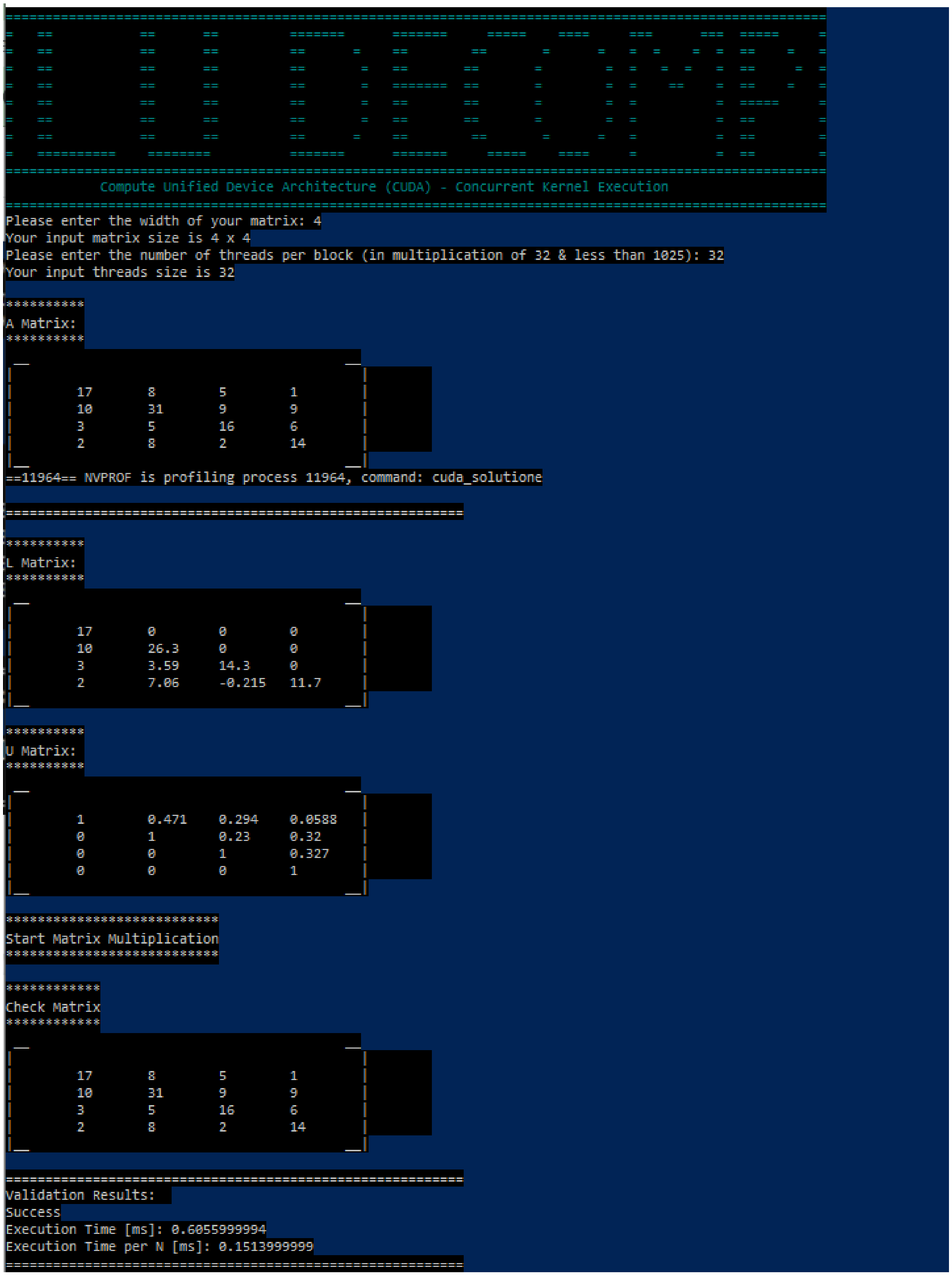
```
========================================================================================================
=   ==           ==      ==         =======      =======     =====    ====     ===      ===  =====     =
=   ==           ==      ==         ==      =    ==        ==       =      =   =  =    =  =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =   =  =   =  ==     =  =
=   ==           ==      ==         ==       =   =======  ==       =        =  =    ==    =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  =====     =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  ==        =
=   ==           ==      ==         ==      =    ==        ==       =      =   =          =  ==        =
=   ==========    ========          =======      =======     =====    ====     =          =  ==        =
========================================================================================================
                       Compute Unified Device Architecture (CUDA) - Unified Memory
========================================================================================================
Please enter the width of your matrix: 4
Your input matrix size is 4 x 4
Please enter the number of threads per block (in multiplication of 32 & less than 1025): 32
Your input threads size is 32
==21472== NVPROF is profiling process 21472, command: cuda_solutionc

**********
A Matrix:
**********

 __                                         __
|                                             |
|        17       8        5        1         |
|        10       31       9        9         |
|        3        5        16       6         |
|        2        8        2        14        |
|__                                         __|
Execution Time [ms]: 0.5112959743
Execution Time per N [ms]: 0.1278239936

**********
L Matrix:
**********

 __                                         __
|                                             |
|        17       0        0        0         |
|        10       26.3     0        0         |
|        3        3.59     14.3     0         |
|        2        7.06     -0.215   11.7      |
|__                                         __|

**********
U Matrix:
**********

 __                                         __
|                                             |
|        1        0.471    0.294    0.0588    |
|        0        1        0.23     0.32      |
|        0        0        1        0.327     |
|        0        0        0        1         |
|__                                         __|

***************************
Start Matrix Multiplication
***************************

************
Check Matrix
************

 __                                         __
|                                             |
|        17       8        5        1         |
|        10       31       9        9         |
|        3        5        16       6         |
|        2        8        2        14        |
|__                                         __|

==========================================================
Validation Results:
Success
Execution Time [ms]: 0.5112959743
Execution Time per N [ms]: 0.1278239936
==========================================================
```

Figure 8.30: Validation result for CUDA unified memory solution (Solution C)



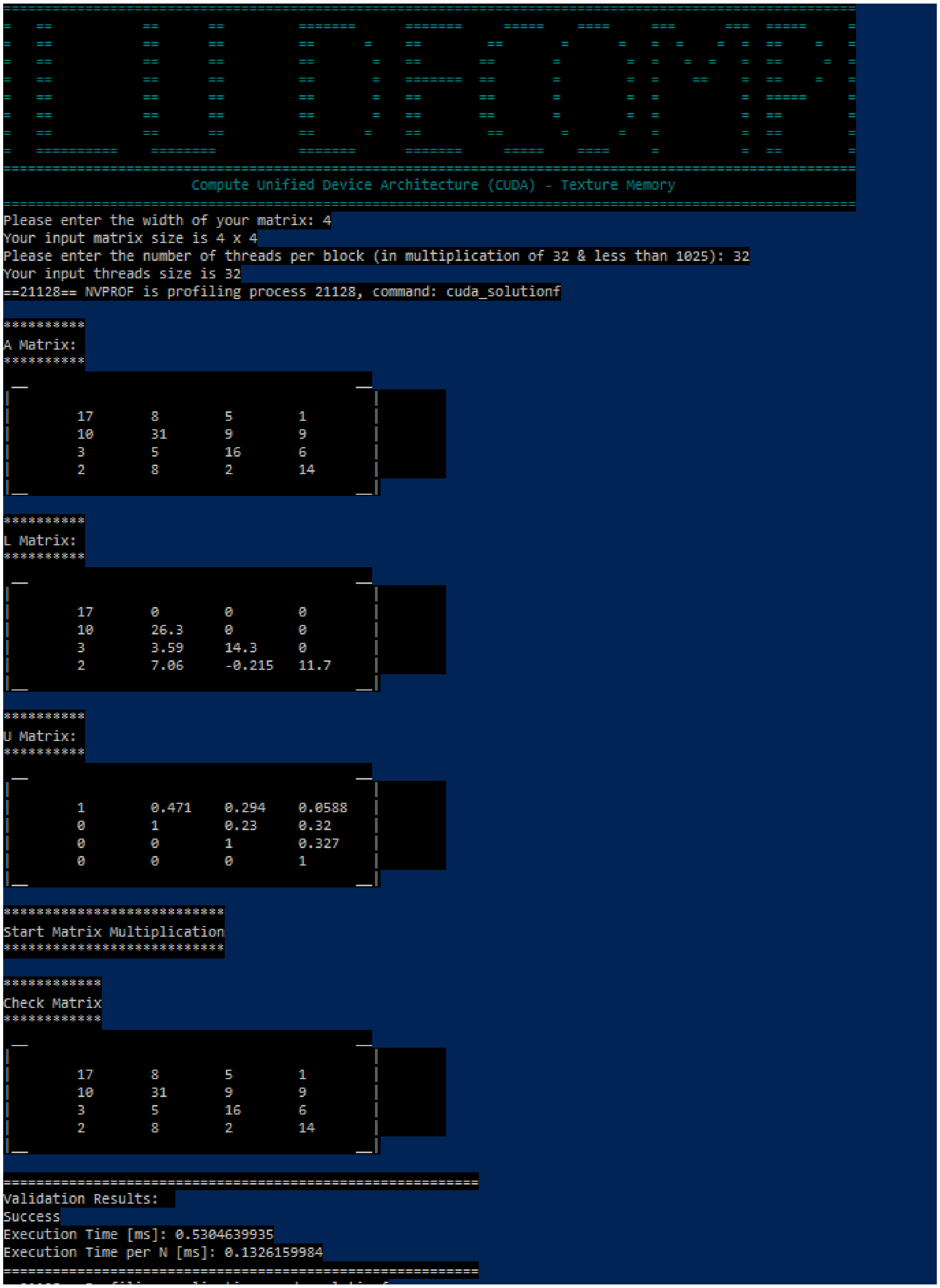
```
========================================================================================================
=   ==           ==      ==         =======      =======     =====    ====     ===      ===  =====     =
=   ==           ==      ==         ==      =    ==        ==       =      =   =  =    =  =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =   =  =   =  ==     =  =
=   ==           ==      ==         ==       =   =======  ==       =        =  =     ==   =  ==    =   =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  =====     =
=   ==           ==      ==         ==       =   ==       ==       =        =  =          =  ==        =
=   ==           ==      ==         ==      =    ==        ==       =      =   =          =  ==        =
=   ==========    ========          =======      =======     =====    ====     =          =  ==        =
========================================================================================================
                  Compute Unified Device Architecture (CUDA) - Zero-Copy Host Code
========================================================================================================
Please enter the width of your matrix: 4
Your input matrix size is 4 x 4
Please enter the number of threads per block (in multiplication of 32 & less than 1025): 32
Your input threads size is 32
==2604== NVPROF is profiling process 2604, command: cuda_solutiond

**********
A Matrix:
**********
 __                                        __
|                                            |
|        17       8        5        1        |
|        10       31       9        9        |
|        3        5        16       6        |
|        2        8        2        14       |
|__                                        __|

==========================================================

**********
L Matrix:
**********
 __                                        __
|                                            |
|        17       0        0        0        |
|        10       26.3     0        0        |
|        3        3.59     14.3     0        |
|        2        7.06     -0.215   11.7     |
|__                                        __|

**********
U Matrix:
**********
 __                                        __
|                                            |
|        1        0.471    0.294    0.0588   |
|        0        1        0.23     0.32     |
|        0        0        1        0.327    |
|        0        0        0        1        |
|__                                        __|

***************************
Start Matrix Multiplication
***************************

************
Check Matrix
************
 __                                        __
|                                            |
|        17       8        5        1        |
|        10       31       9        9        |
|        3        5        16       6        |
|        2        8        2        14       |
|__                                        __|

==========================================================
Validation Results:
Success
Execution Time [ms]: 0.3001919985
Execution Time per N [ms]: 0.07504799962
==========================================================
```

Figure 8.31: Validation result for CUDA zero-copy host code solution (Solution D)



```
=========================================================================================================
=   ==           ==      ==         =======      =======     =====    ====      ===      ===  =====     =
=   ==           ==      ==         ==       =   ==         ==       =      =   =  =    =  =  ==    =   =
=   ==           ==      ==         ==        =  ==        ==       =        =  =   =  =   =  ==     =  =
=   ==           ==      ==         ==        =  =======   ==       =        =  =    ==    =  ==    =   =
=   ==           ==      ==         ==        =  ==        ==       =        =  =          =  =====     =
=   ==           ==      ==         ==        =  ==        ==       =        =  =          =  ==        =
=   ==           ==      ==         ==       =   ==         ==       =      =   =          =  ==        =
=   ==========    ========          =======      =======     =====    ====      =          =  ==        =
=========================================================================================================
            Compute Unified Device Architecture (CUDA) - Concurrent Kernel Execution
=========================================================================================================
Please enter the width of your matrix: 4
Your input matrix size is 4 x 4
Please enter the number of threads per block (in multiplication of 32 & less than 1025): 32
Your input threads size is 32

**********
A Matrix:
**********
 __                                         __
|                                             |
|        17       8        5        1         |
|        10       31       9        9         |
|        3        5        16       6         |
|        2        8        2        14        |
|__                                         __|
==11964== NVPROF is profiling process 11964, command: cuda_solutione

==========================================================

**********
L Matrix:
**********
 __                                         __
|                                             |
|        17       0        0        0         |
|        10       26.3     0        0         |
|        3        3.59     14.3     0         |
|        2        7.06     -0.215   11.7      |
|__                                         __|

**********
U Matrix:
**********
 __                                         __
|                                             |
|        1        0.471    0.294    0.0588    |
|        0        1        0.23     0.32      |
|        0        0        1        0.327     |
|        0        0        0        1         |
|__                                         __|

***************************
Start Matrix Multiplication
***************************

************
Check Matrix
************
 __                                         __
|                                             |
|        17       8        5        1         |
|        10       31       9        9         |
|        3        5        16       6         |
|        2        8        2        14        |
|__                                         __|

==========================================================
Validation Results:
Success
Execution Time [ms]: 0.6055999994
Execution Time per N [ms]: 0.1513999999
==========================================================
```

Figure 8.32: Validation result for CUDA concurrent kernel execution solution (Solution E)



```
Compute Unified Device Architecture (CUDA) - Texture Memory

Please enter the width of your matrix: 4
Your input matrix size is 4 x 4
Please enter the number of threads per block (in multiplication of 32 & less than 1025): 32
Your input threads size is 32
==21128== NVPROF is profiling process 21128, command: cuda_solutionf

**********
A Matrix:
**********
 __                                          __
|                                              |
|        17        8         5         1       |
|        10        31        9         9       |
|        3         5         16        6       |
|        2         8         2         14      |
|__                                          __|

**********
L Matrix:
**********
 __                                          __
|                                              |
|        17        0         0         0       |
|        10        26.3      0         0       |
|        3         3.59      14.3      0       |
|        2         7.06      -0.215    11.7    |
|__                                          __|

**********
U Matrix:
**********
 __                                          __
|                                              |
|        1         0.471     0.294     0.0588  |
|        0         1         0.23      0.32    |
|        0         0         1         0.327   |
|        0         0         0         1       |
|__                                          __|

***************************
Start Matrix Multiplication
***************************

************
Check Matrix
************
 __                                          __
|                                              |
|        17        8         5         1       |
|        10        31        9         9       |
|        3         5         16        6       |
|        2         8         2         14      |
|__                                          __|

==========================================================
Validation Results:
Success
Execution Time [ms]: 0.5304639935
Execution Time per N [ms]: 0.1326159984
==========================================================
```

Figure 8.33: Validation result for CUDA texture memory solution (Solution F)

### 8.3.3 Execution Time

Table 8.15: Execution Time (in seconds) for Global Memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | 0.04984 | 0.17421 | 0.38997 | 0.79888 | **2.70531** | 9.09383 |
| 64 | 0.03661 | 0.16431 | 0.36480 | 0.83093 | 2.68963 | 9.33023 |
| 128 | **0.03506** | 0.17276 | 0.35894 | **0.78488** | 2.95359 | **9.04263** |
| 256 | 0.07706 | 0.09493 | 0.36225 | 0.81929 | 2.78124 | 9.05863 |
| 512 | 0.07178 | 0.17530 | 0.33208 | 0.79751 | 3.63940 | 9.05387 |
| 1024 | 0.06928 | **0.08537** | **0.27128** | 0.80043 | 2.97050 | 10.52962 |

Table 8.16: Execution Time (in seconds) forPinned Memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | **0.02736** | 0.10927 | 0.44917 | **2.41430** | **16.28085** | **122.32720** |
| 64 | 0.02998 | 0.11236 | 0.46603 | 2.42306 | 16.50132 | 125.31890 |
| 128 | 0.02804 | 0.10749 | 0.44658 | 2.43402 | 16.45250 | 124.80410 |
| 256 | 0.02799 | 0.10688 | 0.44865 | 2.43586 | 16.59821 | 125.30690 |
| 512 | 0.02789 | **0.10626** | **0.44364** | 2.43497 | 16.57110 | 124.97630 |
| 1024 | 0.02985 | 0.11030 | 0.45410 | 2.46505 | 16.75826 | 127.88770 |

Table 8.17: Execution Time (in seconds) for Unified Memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| 32 | 0.01169 | 0.04240 | **0.14179** | **0.54718** | 2.21280 | **8.80364** |
| 64 | 0.01650 | **0.04081** | 0.14645 | 0.56566 | 2.20501 | 8.89555 |
| 128 | 0.01541 | 0.04462 | 0.16823 | 0.56429 | 2.19999 | 8.86254 |
| 256 | 0.01633 | 0.05078 | 0.14625 | 0.54828 | **2.19696** | 8.85815 |
| 512 | **0.01134** | 0.05211 | 0.16220 | 0.55741 | 2.21114 | 8.87889 |
| 1024 | 0.01229 | 0.04665 | 0.16322 | 0.55306 | 2.29305 | 10.20860 |

Table 8.18: Execution Time (in seconds) for Zero-copy host code

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | 0.03042 | 0.11409 | **0.44836** | **2.44160** | 16.64653 | 184.68680 |
| 64 | 0.03044 | 0.11157 | 0.46024 | 2.45066 | 16.52452 | 155.92500 |
| 128 | **0.02918** | **0.10921** | 0.45841 | 2.45262 | **16.47533** | 127.40610 |
| 256 | 0.03185 | 0.11329 | 0.45866 | 2.47111 | 16.62833 | **125.10950** |
| 512 | 0.02966 | 0.11056 | 0.45597 | 2.46540 | 16.55072 | 125.34890 |
| 1024 | 0.03240 | 0.11211 | 0.45382 | 2.46707 | 16.77176 | 128.54100 |

Table 8.19: Execution Time (in seconds) for Concurrent Kernel Execution

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | **0.00988** | **0.02937** | 0.09848 | **0.34245** | 1.33443 | 5.66745 |
| 64 | 0.00996 | 0.03262 | **0.09134** | 0.33567 | **1.33904** | 5.64097 |
| 128 | 0.01280 | 0.03755 | 0.09656 | 0.34409 | 1.35146 | **5.63464** |
| 256 | 0.01021 | 0.03116 | 0.09823 | 0.35781 | 1.35480 | 5.65534 |
| 512 | 0.01021 | 0.03153 | 0.10426 | 0.34609 | 1.34376 | 5.80854 |
| 1024 | 0.01047 | 0.03413 | 0.09991 | 0.35212 | 1.34513 | 6.28754 |

Table 8.20: Execution Time (in seconds) for Texture memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | **0.01205** | 0.04550 | 0.15221 | **0.55426** | 2.62429 | 8.82198 |
| 64 | 0.01450 | 0.04404 | 0.15484 | 0.56123 | **2.16933** | **8.80271** |
| 128 | 0.01292 | 0.04359 | 0.14940 | 0.56099 | 2.19373 | 8.81063 |
| 256 | 0.01263 | **0.04228** | **0.14886** | 0.56584 | 2.18343 | 8.93800 |
| 512 | 0.01238 | 0.04270 | 0.15678 | 0.57497 | 2.18096 | 8.92234 |
| 1024 | 0.01467 | 0.04336 | 0.15300 | 0.57288 | 2.27230 | 10.25970 |

Table 8.21: Execution Time (in seconds) for global memory implementation and another five solutions with a number of threads of 32.

| **Solutions / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| Global Memory | 0.04984 | 0.17421 | 0.38997 | 0.79888 | 2.70531 | 9.09383 |
| Pinned Memory | 0.02736 | 0.10927 | 0.44917 | 2.41430 | 16.28085 | 122.32720 |
| Unified Memory | 0.01169 | 0.04240 | 0.14179 | 0.54718 | 2.21280 | 8.80364 |
| Zero-copy Host Code | 0.03042 | 0.11409 | 0.44836 | 2.44160 | 16.64653 | 184.68680 |
| Concurrent Kernel Execution | **0.00988** | **0.02937** | **0.09848** | **0.34245** | **1.33443** | **5.66745** |
| Texture Memory | 0.01205 | 0.04550 | 0.15221 | 0.55426 | 2.62429 | 8.82198 |

### 8.3.4 Speedup

Speedup = Serial Runtime / Parallel Runtime

Table 8.22:Speedup for Global Memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | 0.02006 | 0.04018 | 0.16155 | 0.65341 | 1.73363 | 5.08213 |
| 64 | 0.02731 | 0.04260 | 0.17270 | 0.62821 | **1.74373** | 4.95336 |
| 128 | **0.02852** | 0.04052 | 0.17552 | **0.66507** | 1.58790 | **5.11090** |
| 256 | 0.01298 | 0.07374 | 0.17391 | 0.63714 | 1.68630 | 5.10188 |
| 512 | 0.01393 | 0.03993 | 0.18971 | 0.65454 | 1.28867 | 5.10456 |
| 1024 | 0.01443 | **0.08200** | **0.23223** | 0.65215 | 1.57886 | 4.38914 |

Table 8.23: Speedup for Pinned Memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | **0.03655** | 0.06406 | 0.14026 | **0.21621** | **0.28807** | **0.37781** |
| 64 | 0.03336 | 0.06230 | 0.13518 | 0.21543 | 0.28422 | 0.36879 |
| 128 | 0.03566 | 0.06512 | 0.14107 | 0.21446 | 0.28506 | 0.37031 |
| 256 | 0.03573 | 0.06549 | 0.14042 | 0.21430 | 0.28256 | 0.36882 |
| 512 | 0.03586 | **0.06588** | **0.14201** | 0.21438 | 0.28302 | 0.36980 |
| 1024 | 0.03350 | 0.06346 | 0.13874 | 0.21176 | 0.27986 | 0.36138 |

Table 8.24: Speedup for Unified Memory

| **Number of threads /** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|

| **Matrix Size** | | | | | | |
|---|---|---|---|---|---|---|
| 32 | 0.08554 | 0.16509 | **0.44432** | **0.95398** | 2.11949 | **5.24965** |
| 64 | 0.06061 | **0.17153** | 0.43018 | 0.92282 | 2.12697 | 5.19541 |
| 128 | 0.06489 | 0.15688 | 0.37449 | 0.92506 | 2.13183 | 5.21476 |
| 256 | 0.06124 | 0.13785 | 0.43077 | 0.95207 | **2.13477** | 5.21734 |
| 512 | **0.08818** | 0.13433 | 0.38841 | 0.93647 | 2.12108 | 5.20516 |
| 1024 | 0.08137 | 0.15005 | 0.38598 | 0.94384 | 2.04531 | 4.52716 |

Table 8.25: Speedup for Zero-copy host code

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | 0.03287 | 0.06136 | **0.14051** | **0.21379** | 0.28174 | 0.25024 |
| 64 | 0.03285 | 0.06274 | 0.13689 | 0.21300 | 0.28382 | 0.29640 |
| 128 | **0.03427** | **0.06410** | 0.13743 | 0.21283 | **0.28467** | 0.36275 |
| 256 | 0.03140 | 0.06179 | 0.13736 | 0.21124 | 0.28205 | **0.36940** |
| 512 | 0.03372 | 0.06331 | 0.13817 | 0.21173 | 0.28337 | 0.36870 |
| 1024 | 0.03086 | 0.06244 | 0.13882 | 0.21159 | 0.27964 | 0.35954 |

Table 8.26: Speedup for Concurrent Kernel Execution

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | **0.10121** | **0.23834** | 0.63972 | 1.52431 | **3.51461** | 8.15464 |
| 64 | 0.10040 | 0.21459 | **0.68973** | **1.55510** | 3.50251 | 8.19292 |
| 128 | 0.07813 | 0.18642 | 0.65244 | 1.51704 | 3.47032 | **8.20212** |
| 256 | 0.09794 | 0.22465 | 0.64135 | 1.45887 | 3.46177 | 8.17210 |
| 512 | 0.09794 | 0.22201 | 0.60426 | 1.50828 | 3.49021 | 7.95656 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 1024 | 0.09551 | 0.20510 | 0.63057 | 1.48245 | 3.48665 | 7.35041 |

Table 8.27: Speedup for Texture memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | **0.08299** | 0.15385 | 0.41390 | **0.94180** | 1.78715 | 5.23873 |
| 64 | 0.06897 | 0.15895 | 0.40687 | 0.93010 | **2.16196** | **5.25020** |
| 128 | 0.07740 | 0.16059 | 0.42169 | 0.93050 | 2.13791 | 5.24548 |
| 256 | 0.07918 | **0.16556** | **0.42322** | 0.92252 | 2.14800 | 5.17073 |
| 512 | 0.08078 | 0.16393 | 0.40184 | 0.90787 | 2.15043 | 5.17981 |
| 1024 | 0.06817 | 0.16144 | 0.41176 | 0.91119 | 2.06399 | 4.50462 |

Table 8.28: Speedup for global memory implementation and another five solutions with a number of threads of 32.

| **Solutions / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| Global Memory | 0.02006 | 0.04018 | 0.16155 | 0.65341 | 1.73363 | 5.08213 |
| Pinned Memory | 0.03655 | 0.06406 | 0.14026 | 0.21621 | 0.28807 | 0.37781 |
| Unified Memory | 0.08554 | 0.16509 | 0.44432 | 0.95398 | 2.11949 | 5.24965 |
| Zero-copy Host Code | 0.03287 | 0.06136 | 0.14051 | 0.21379 | 0.28174 | 0.25024 |
| Concurrent Kernel Execution | **0.10121** | **0.23834** | **0.63972** | **1.52431** | **3.51461** | **8.15464** |
| Texture Memory | 0.08299 | 0.15385 | 0.41390 | 0.94180 | 1.78715 | 5.23873 |

### 8.3.5 CudaMalloc Time

Table 8.29: CudaMalloc Time (in seconds) for Global Memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | **0.16345** | **0.20160** | 0.24531 | 0.16403 | 0.22752 | 0.17173 |
| 64 | 0.17795 | 0.20896 | 0.24362 | 0.17221 | 0.17739 | 0.17184 |
| 128 | 0.16398 | 0.26123 | 0.24276 | **0.16197** | 0.17492 | 0.17777 |
| 256 | 0.17656 | 0.24479 | **0.24033** | **0.16197** | **0.16391** | 0.17856 |
| 512 | 0.17336 | 0.28929 | 0.24463 | 0.16325 | 0.16507 | **0.16170** |
| 1024 | 0.16613 | 0.29150 | 0.25821 | 0.17315 | 0.17855 | 0.16519 |

Table 8.30: CudaMalloc Time (in seconds) for Pinned Memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | 0.20331 | 0.26874 | 0.23806 | 0.24548 | 0.24231 | 0.24231 |
| 64 | 0.24526 | 0.12600 | 0.24508 | 0.13635 | **0.12553** | 0.26183 |
| 128 | 0.13930 | 0.13840 | 0.13135 | **0.12001** | 0.24118 | **0.13715** |
| 256 | **0.12168** | **0.12217** | **0.12470** | 0.13516 | 0.13449 | 0.21838 |
| 512 | 0.13820 | 0.14655 | 0.13013 | 0.13374 | 0.13519 | 0.34681 |
| 1024 | 0.26013 | 0.24562 | 0.30900 | 0.27299 | 0.33428 | 0.30501 |

Table 8.31: CudaMalloc Time (in seconds) for Unified Memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | 0.36271 | 0.27402 | **0.31198** | 0.39552 | 0.41057 | 0.35979 |
| 64 | 0.36046 | **0.20505** | 0.36988 | **0.33482** | 0.34440 | 0.41290 |
| 128 | **0.26442** | 0.22084 | 0.43072 | 0.36105 | 0.39148 | 0.34884 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 256 | 0.28507 | 0.3716 | 0.33329 | 0.34642 | 0.34359 | **0.34028** |
| 512 | 0.29153 | 0.34668 | 0.39260 | 0.37569 | **0.34308** | 0.36918 |
| 1024 | 0.31002 | 0.38122 | 0.35055 | 0.38168 | 0.36244 | 0.34129 |

Table 8.32: CudaMalloc Time (in seconds) for Zero-copy host code

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|
| 32 | 0.00198 | 0.00141 | **0.00129** | 0.00235 | 0.00435 | 0.01399 |
| 64 | 0.00139 | 0.00150 | 0.00191 | 0.00270 | **0.00271** | 0.01324 |
| 128 | 0.00148 | 0.00128 | 0.00130 | 0.00240 | 0.00349 | **0.01204** |
| 256 | **0.00134** | 0.00161 | 0.00169 | 0.00424 | 0.00447 | 0.01483 |
| 512 | 0.00161 | **0.00125** | 0.00136 | 0.00249 | 0.00410 | 0.01313 |
| 1024 | 0.00142 | 0.00125 | 0.00139 | **0.00235** | 0.00403 | 0.01814 |

Table 8.33: CudaMalloc Time (in seconds) for  Concurrent Kernel Execution

| Number of threads / Matrix Size | 64 | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| 32 | 0.00063 | 0.00033 | 0.00041 | 0.00051 | 0.00064 | **0.00085** |
| 64 | **0.00028** | 0.00036 | 0.00034 | 0.00053 | 0.00061 | 0.00091 |
| 128 | 0.00036 | 0.00041 | 0.00035 | 0.00049 | 0.00065 | 0.00102 |
| 256 | 0.00030 | 0.00031 | 0.00035 | 0.00051 | **0.00056** | 0.01106 |
| 512 | 0.00031 | **0.00030** | **0.00033** | 0.00055 | 0.00068 | 0.00099 |
| 1024 | 0.00029 | 0.00036 | 0.00042 | **0.00048** | 0.00058 | 0.01206 |

Table 8.34: CudaMalloc Time (in seconds) for Texture memory

| **Number of threads / Matrix Size** | **64** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| 32 | 0.15752 | 0.15997 | **0.12164** | 0.15038 | 0.24734 | 0.17032 |
| 64 | 0.15868 | 0.16772 | 0.14109 | **0.14222** | 0.16067 | 0.17763 |
| 128 | 0.16421 | 0.15657 | 0.16498 | 0.14562 | 0.20097 | 0.16482 |
| 256 | **0.15537** | 0.16238 | 0.16233 | 0.28759 | 0.24450 | **0.15484** |
| 512 | 0.15752 | 0.16912 | 0.16406 | 0.27869 | **0.12308** | 0.24808 |
| 1024 | 0.15962 | **0.15523** | 0.27044 | 0.29267 | 0.30266 | 0.17941 |

## 8.4 Speedup

### 8.4.1 Speedup compared to sequential

Speedup = Serial Runtime / Parallel Runtime

Table 8.35: Speedup for all model compared to sequential

| **Methods** | **Matrix Size, n (n * n)** | | | | | |
|---|---|---|---|---|---|---|
| | **64** | **128** | **256** | **512** | **1024** | **2048** |
| Sequential | 1 | 1 | 1 | 1 | 1 | 1 |
| OMP-SIMD | 2 | 1.8421 | 1.291 | 1.3861 | 1.6966 | 1.8869 |
| MPI | 0.0909 | 0.1750 | 0.4375 | 0.1289 | 0.0925 | 0.0955 |
| CUDA | 0.1838 | 0.3849 | 0.6545 | 1.4997 | 3.3817 | 7.7230 |

### 8.4.2 Predicted Amdahl's Law Speedup

Table 8.36: Predicted Amdahl's Law Speedup for all model

| **Methods** | **Matrix Size, n (n * n)** | | | | | |
|---|---|---|---|---|---|---|
| | **64** | **128** | **256** | **512** | **1024** | **2048** |
| OpenMP (OMP-SIMD) | 2.0000 | 1.8421 | 1.2910 | 1.3861 | 1.6966 | 1.8869 |
| MPI | 0.1667 | 0.2204 | 0.4534 | 0.1325 | 0.0938 | 0.0962 |
| CUDA | 0.1838 | 0.3849 | 0.6545 | 1.4997 | 3.3817 | 7.7230 |

### 8.4.3 Actual difference compared to predicted speedup

Table 8.37: Actual difference compared between actual and predicted speedup for all models

| **Methods** | **Matrix Size, n (n * n)** | | | | | |
|---|---|---|---|---|---|---|
| | **64** | **128** | **256** | **512** | **1024** | **2048** |
| OpenMP (OMP-SIMD) | 0.6667 | 0.5458 | 0.0231 | 0.0165 | 0.0183 | 0.0130 |
| MPI | -0.0758 | -0.0454 | -0.0159 | -0.0036 | -0.0013 | -0.0007 |
| CUDA | -0.0826 | -0.1466 | -0.0148 | 0.0246 | 0.1329 | 0.4316 |