University of Porto - Faculty of Engineering

MASTER IN INFORMATICS AND COMPUTING ENGINEERING

Project Report

LU Factorization

IMPACT OF PARALLELISM AND CACHE-AWARE PROGRAMMING

Miguel Rodrigues
up201906042@edu.fe.up.pt
Sérgio Estêvão
up201905680@edu.fe.up.pt

May 2023

Contents

List of Figures

1 Introduction

LU Factorization is a linear algebra procedure that is used to factorize a given square matrix into lower and upper triangular matrices. This technique has a wide range of engineering, physics, and computer science applications.

We want to illustrate the implementation of LU Factorization in a simple and block-oriented way in this study, and then investigate its parallel implementation utilizing the OpenMP and CUDA frameworks. Our primary goal is to improve the algorithm's efficiency while taking into account significant factors such as memory usage, data proximity, and parallelism.

The basic implementation of the LU Factorization method and its time complexity will be shown in the first step. Following that, we will show how a block-oriented approach can improve algorithm performance by breaking up the algorithm into tasks that can be executed simultaneously.

Then, we'll look at how the LU Factorization algorithm can be implemented in parallel using three different frameworks: OpenMP and CUDA. We will examine each framework's scalability and compare the obtained speedup and efficiency.

Finally, in the conclusion section, we will assess the outcomes acquired in each stage of our experiments and give insights into the practical uses of LU Factorization in real-world circumstances.

2 LU Factorization

As mentioned previously, the LU factorization is one of the most popular techniques employed in the solving of linear systems by using forward and back substitutions. It is a modified form of Gaussian elimination, based on the following equations.

$$\begin{gathered}\mathbf{Ax} = \mathbf{b}\\ \mathbf{LUx} = \mathbf{b}\\ \mathbf{Ly} = \mathbf{b}\\ \mathbf{Ux} = \mathbf{y}\end{gathered}$$

Given A a square matrix, LU factorization factors A into a lower triangular matrix L and upper triangular matrix L, thus $A = LU$. The factorization is not unique meaning that exist numerous valid L and U for a given matrix A. Another requirement is that A must be non-singular, i.e. A^{-1} is admissible.

Given this, it is possible to use a single matrix in memory to store the content of both triangular matrices. Moreover, in this work, we assume that the lower triangular has a unitary diagonal as described right below.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

Despite its efficiency regarding memory usage, LU factorization suffers from numerical instability. The algorithm does yield the correct result if main diagonal elements are much smaller than the remaining ones. In the particular case of 0, the algorithm is not able to terminate due to a division by 0 error.

To overcome this the pivoting technique is used. This technique swaps rows (or columns) and stores such information in P - the permutation matrix. With this in mind, most linear of linear algebra libraries implement LU with pivoting, thus $PA = LU$. Stability can be further increased when using full pivoting, i.e. by swapping rows and columns giving $PAQ = LU$.

Regarding LU's complexity, it requires $\Theta(\frac{2}{3}n^3)$ floating-point operations for a matrix of size $n \times n$. Partial pivoting adds a quadratic term while full pivoting increases the complexity to $\Theta(n^3)$.

In this work, the algorithms implemented do not deal with pivoting at all. We assume that the input is always a diagonal dominant matrix, which always admits an LU factorization without requiring either partial or full pivoting.

3 Libraries and Tools

In this section, we will provide a short briefing about each of the external tools and libraries that we have used to develop the programs shown in the following sections. Another relevant note is that all the tests shown in this report were executed on a machine with a 5.15.0-60-generic Kernel, running an Intel® Core™ i7-9750H CPU (Coffee Lake) with 6 cores and 12 threads with Hyper-Threading @ 2.60 GHz.

3.1 perf

`perf`[1] [1] is a command line tool built on the Linux kernel. It provides several insights and an easy-to-use command-line interface to benchmark user programs.

It provides counters that measure the cache miss ratio - for both L1 and L2 - as well as the total number of clock cycles and instructions performed during the execution of a program.

3.2 OpenMP

OpenMP[2] [2], is an API specification for parallel programming.

The API provides the convenience of not having to deal with thread management, which can get complicated very quickly. Therefore, such responsibility is passed to the compiler which implements the specification.

The usage is pretty straightforward. In source code, `#pragma` statements allow a certain region to be executed in parallel. Also, when compiling the flag `-fopenmp` must be provided to enable this functionality.

3.3 CUDA

NVIDIA's CUDA[3] [3] is a parallel computing platform and programming paradigm.

It allows developers to build code that runs on NVIDIA GPUs, resulting in considerable speedups in a wide range of applications. CUDA is a collection of C++ language extensions that allow the

[1]`perf` versions are tied to the kernel versions.
[2]Version 4.5 of OpenMP, bundled with g++-12, was used.
[3]Version 12.1 of CUDA was used.

building of parallel kernels that run on the GPU. It also includes a runtime framework for controlling the execution of these kernels on the GPU.

To utilize CUDA, the code must be built with the NVIDIA CUDA compiler and must be executed on a machine with an NVIDIA GPU that is compatible.

3.4 DPC++ and SYCL

DPC++[4] [4], Data Parallel C++, is an open-source ISO C++ compiler developed by Intel built on top of the LLVM project. It implements the C++ standard library and the SYCL[5] [5] specification.

SYCL, pronounced "*sickle*", is an abstract layer that eases the development of high-performance applications in heterogeneous platforms in C++17 or higher. Therefore, a single source is able to execute accelerated code on different chips depending on the needs of each application. SYCL also provides an extensive range of backends into which kernel functions are compiled into. Furthermore, SYCL uses the latest C++ generic programming high-level features, such as lambda functions and templates, to enhance code readability while retaining the advantages of low-level optimized code.

There are currently 5 major distinct implementations of this standard, but we have opted for the Intel implementation. The rationale behind this choice is the support for the CUDA backend, which uses, as mentioned in the previous section, the computational power of NVIDIA GPUs.

4 Sequential Algorithms

In this section, we will discuss the sequential versions of the LU Decomposition algorithm. We will start with the basic version of the algorithm, followed by the blocked version.

The sequential versions serve as a baseline for comparison with the parallel versions, which are covered in the next section. Furthermore, C++20 allows for finer-grained refinement of the program's behavior, allowing us to investigate the algorithm's performance in depth.

4.1 Basic

In this basic version, we first define the matrix type and size, which are used throughout the algorithm. The `lu` function takes as inputs the matrix A and its size N. It performs LU decomposition on the matrix in place with 2 nested loops execute the core algorithm.

The outer loop iterates over the matrix's diagonal elements. The loop invariant holds while the diagonal value is not 0 and the end of the diagonal has not been reached. If the invariant is broken by the latter condition this means that the algorithm terminated correctly, otherwise, it yields an incomplete, and therefore, incorrect result.

The inner loops zeros off the elements below the pivot element, for each row i below the pivot it divides the element $A[i,k]$ by the pivot element $A[k,k]$, essentially assigning it to the value of the lower triangular matrix L. Then we subtract this value times the corresponding elements in the pivot element's row from the current row, effectively setting them to the upper triangular matrix U.

Finally, the lower and upper triangular matrices are combined into the original matrix, and the initial matrix A is now decomposed into L and U.

[4]Version 2023.1.0 was used.
[5]2020 version was used.

```
template<typename T>
void lu(matrix_t<T> A, const matrix_size_t N)
{
    for (matrix_size_t k = 0; A[k * N + k] != 0.0 && k < N; ++k) {
        for (matrix_size_t i = k + 1; i < N; ++i) {
            A[i * N + k] /= A[k * N + k];
            for (matrix_size_t j = k + 1; j < N; ++j)
                A[i * N + j] -= A[i * N + k] * A[k * N + j];
        }
    }
}
```

4.2 Block-Oriented

The block-oriented version is, naturally, more complex than the serial version. Given this, in order to simplify the analysis, our implementation assumes that the size of the input matrix is divisible by the block's size.

This version requires a series of tricks, that will be detailed in this section, to enable parallelism. Furthermore, splitting the matrix into blocks improves cache hit rates and the overall data locality.

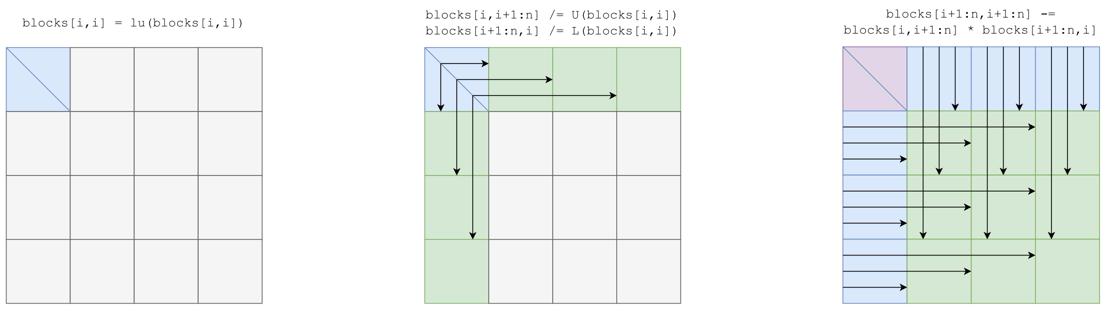
The implementation describes in this section uses a set of routines from BLAS. BLAS, short for *Basic Linear Algebra Subprograms*, is a specification of a set of routines that perform common linear algebra operations.

The blocked algorithm follows a similar approach to the one used by the sequential version, but instead of processing a single element at a time, it processes a sub-matrix. The first of the algorithm is to iterate over the diagonal blocks and perform the same procedure presented in the previous section, i.e. the simple LU. This results in the i^{th} diagonal block being factorized in LU, those matrices are used to compute the next step.

The next step involves propagating the results to the blocks in the i^{th} row and column of the block matrix. This propagation is nothing more than triangular matrix solve, i.e. `dtrsm` a procedure from BLAS level 2. Recall that for the sequential version, this step is iterative.

The final step is to update the remaining blocks. This step is much more costly concerning computational resources, it uses the `dgemm` subroutine from BLAS level 3 and performs a matrix multiplication.

The following figure provides a visual demonstration of the algorithm explained above.


blocks[i,i] = lu(blocks[i,i])
blocks[i,i+1:n] /= U(blocks[i,i])
blocks[i+1:n,i] /= L(blocks[i,i])
blocks[i+1:n,i+1:n] -=
blocks[i,i+1:n] * blocks[i+1:n,i]

(a) LU on the diagonal block **(b)** Triangular matrix solve **(c)** General matrix multiplication

Figure 1: Block based LU factorization

The C++ code which drives this approach can be expressed in the following form.

```cpp
template <typename T>
void block_lu(matrix_t<T> A, const matrix_size_t N, const block_size_t B)
{
    const matrix_size_t blocks = N / B;
    for (matrix_size_t i = 0; i < blocks; ++i) {
        lu(A, N, B, i);  // LU decomposition on the diagonal block
        for (matrix_size_t j = i + 1; j < blocks; ++j)
            utrsm(A, N, B, i, j);    // upper triangular matrix solve
        for (matrix_size_t j = i + 1; j < blocks; ++j) {
            ltrsm(A, N, B, i, j);    // lower triangular matrix solve
            for (matrix_size_t k = i + 1; k < blocks; ++k)
                gemm(A, N, B i, j, k);  // general matrix multiplication
        }
    }
}
```

5 Parallel Algorithms

In this section, we will detail the implementation that can take advantage of parallelism for LU decomposition.

5.1 OpenMP

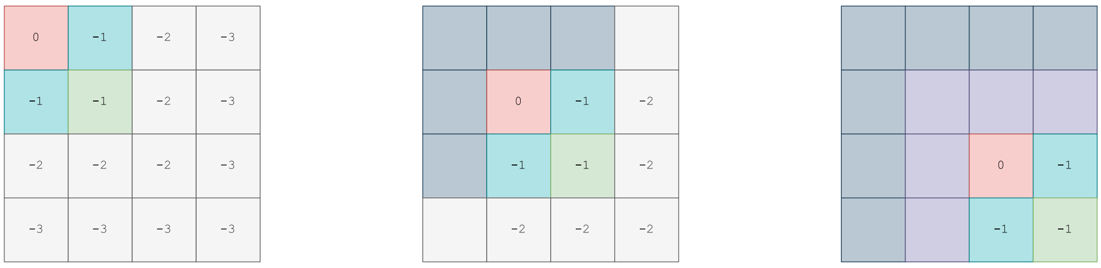
According to what was stated in the section 4.2, the block approach was crucial to enable parallelism. Furthermore, the arrows Figure 1 represent the dependencies between blocks in the input block matrix, thus blocks that do not depend on each other can be used in simultaneous computation as there is no risk of false sharing of data.

Our implementation makes heavy usage of `task` constructs. The motivation for such approach is sustained by several factors, but it happens primarily due to their flexibility which will be describing next.

According to the OpenMP specification it is possible to create a dependency DAG containing the tasks to be executed. For each task it is possible to specify a `depend` clause. This clause receives a list of values that function as references for other tasks in the same `parallel` region. The values in the `depend` clause are used to generate the appropriate tasks before executing the target task. Again, Figure 1 comes in handy as it shows the dependencies between blocks that in our implementation are determined by the memory address of the first element in each block.

Another reason for applying tasks constructs in these version was the possibility of providing tasks with a priority. This is an optimization that takes into account the time expenditure of each tasks and prioritizes those which are part of the critical path. This is achieved by assigning higher priorities to the tasks whose operations are computationally more expensive to be done as soon as possible.

The exact value is given by `omp_get_max_task_priority() - offset` where the `offset` values are according to the figure below.



(a) First iteration **(b)** Second iteration **(c)** Third iteration

Figure 2: Priority values evolution during execution

Given this, tasks are computed only when their parent tasks are completed and then based on the priority.

Another advantage of tasks is that the scheduling is done internally by the compiler. This ensures that the maximum number of available threads are used to the point where it is able to saturate the machine. On top of that, there does not exist the need to specify any type of barrier which introduce overhead.

In this context, the parallel version of the algorithm using OpenMP can be expressed as follows.

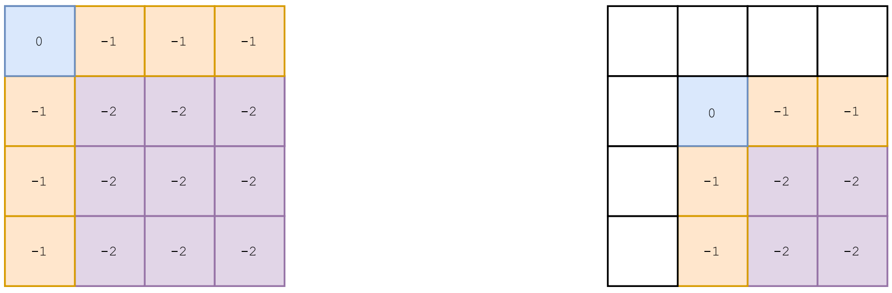
```cpp
template <typename T>
void block_lu(matrix_t<T> A, const matrix_size_t N, const block_size_t B)
{
    const int max_task_priority = omp_get_max_task_priority();
    const int blocks = static_cast<int>(N / B);

    #pragma omp parallel
    #pragma omp single nowait
    for (int i = 0; i < blocks; ++i) {
        #pragma omp task \
            default(none) shared(A, N, B) firstprivate(i) \
            depend(inout: A[(i * N + i) * B]) \
            priority(max_task_priority)
        lu(A, N, B, i);  // LU decomposition on the diagonal block

        for (int j = i + 1; j < blocks; ++j) {
            #pragma omp task \
                default(none) shared(A, N, B) firstprivate(i, j) \
                depend(in: A[(i * N + i) * B]) depend(inout: A[(i * N + j) * B]) \
                priority(std::max(0, max_task_priority - j + i))
            utrsm(A, N, B, i, j);    // upper triangular matrix solve
        }

        for (int j = i + 1; j < blocks; ++j) {
            #pragma omp task \
                default(none) shared(A, N, B) firstprivate(i, j) \
                depend(in: A[(i * N + i) * B]) depend(inout: A[(j * N + i) * B]) \
                priority(std::max(0, max_task_priority - j + i))
            ltrsm(A, N, B, i, j);    // lower triangular matrix solve

            for (int k = i + 1; k < blocks; ++k) {
                #pragma omp task \
                    default(none) shared(A, N, B) firstprivate(i, j, k) \
                    depend(in: A[(i * N + k) * B], A[(j * N + i) * B]) \
                    depend(inout: A[(j * N + k) * B]) \
                    priority(std::max({0, max_task_priority - j + i, max_task_priority - k + i}))
                gemm(A, N, B, i, j, k);  // general matrix multiplication
            }
        }
    }
}
```

5.2 CUDA

We showed in the last section how the block technique allowed us to parallelize the LU decomposition using OpenMP tasks. In this part, we look at an alternate method for parallelizing the LU decomposition with CUDA.

The concept behind this approach is to partition the input matrix into square blocks of size B and decompose each diagonal block using the LU decomposition. This necessitates a total of block iterations, where blocks are the number of blocks in the input matrix. The implementation begins with the GPU allocating memory and copying the input matrix to the device. The kernel functions `baselu`, `row_col_solver`, and `gemm` are then called to conduct LU decomposition on the diagonal blocks, solve the rows and columns of the diagonal blocks, and execute general matrix multiplication, in that order. Each kernel function represents a distinct level of the LU decomposition and is run in its own kernel launch.

The computational order for each kernel launch can be seen in the figure below.



(a) First iteration **(b)** Second iteration

Figure 3: Kernel priority evolution during execution

Managing the dependencies between blocks is one of the most challenging aspects of this system. The calculation of a specific block, in particular, is dependent on the results of preceding blocks. To address this, we synchronize the device after each kernel launch to verify that the previous kernel has completed its execution before launching the next kernel.

Furthermore, one aspect that we took into consideration in this approach was the GPU memory hierarchy. Shared memory is a software-managed cache on the GPU, and a fundamental CUDA feature, that allows for high-bandwidth, low-latency communication between threads inside a block (a group of threads running in parallel on a GPU). When compared to global memory, which is placed off-chip and has a larger access latency, it is physically positioned on the chip, closer to the processor units. By utilizing this shared memory to store the blocks that the program is operating in at a specific time, it is able to perform read and write operations much faster and achieve much better performance. In figure 4 we can visualize the memory hierarchy in a GPU in the context of the CUDA language.

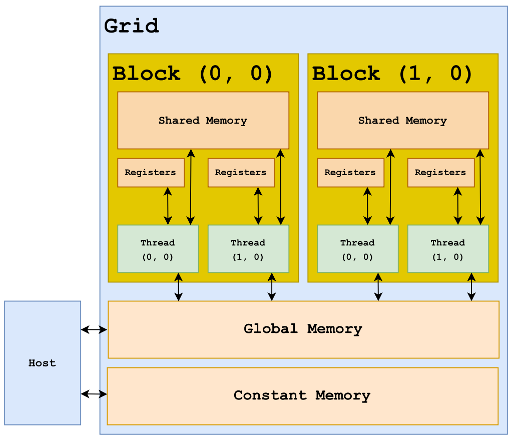


Figure 4: CUDA memory hierarchy in a GPU

The parallel version of the algorithm using CUDA can be expressed as follows.

```
template <typename T>
void lu(matrix_t<T> A, const matrix_size_t N, const block_size_t B)
{
    const block_size_t blocks = static_cast<int>(N / B);

    T* gpu_A;
    cudaMalloc((void **) &gpu_A, N * N * sizeof(T));
    cudaMemcpy(gpu_A, A, N * N * sizeof(T), cudaMemcpyHostToDevice);

    size_t shared_mem_size = B * B * sizeof(T);

    for (block_size_t i = 0; i < blocks; ++i) {
        // LU decomposition on the diagonal block
        baselu<<<1, B, shared_mem_size>>>(gpu_A, N, i);
        //kernel launches are async, so synchronize to make sure the kernel is done before continuing
        cudaDeviceSynchronize();

        int row_col_blocks = (blocks - i - 1) * 2;

        // solve the rows and columns of the diagonal block
        row_col_solver<<<row_col_blocks, B, shared_mem_size>>>(gpu_A, N, i);
        cudaDeviceSynchronize();

        int gemm_blocks = (blocks - i - 1) * (blocks - i - 1);
        // general matrix multiplication
        gemm<<<gemm_blocks, B, shared_mem_size>>>(gpu_A, N, blocks, i);
        cudaDeviceSynchronize();

    }

    cudaMemcpy(A, gpu_A, N * N * sizeof(T), cudaMemcpyDeviceToHost);
    cudaFree(gpu_A);
}
```

5.3 SYCL

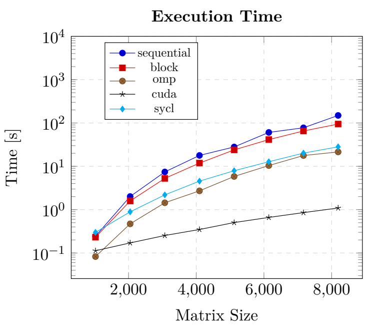
The approach to parallelize the code using SYCL is very similar to what we have done for both OpenMP and CUDA. The tasks remain the same, but each task is executed in a kernel as there are not any data dependencies.

The following snippet of code demonstrates how LU decomposition is obtained on a sub-matrix that is part of the main diagonal of the larger input matrix. Note here, that tasks are submitted to a queue that will execute these instructions on the desired chip set. Despite targeting only the CUDA backend, the following snippet can be executed from the CPU or even a dedicated FPGA.

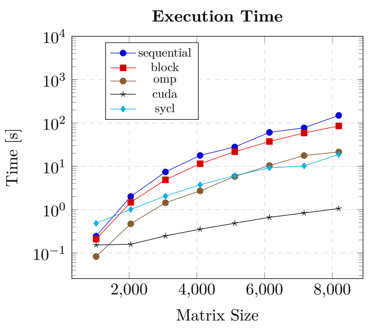
```
template <typename T>
void lu(sycl::queue& q, matrix_t<T> A,
        const matrix_size_t N, const block_size_t B, const matrix_size_t i)
{
    sycl::buffer diagonal_block(A + (i * N + i) * B, sycl::range(B, B));

    for (matrix_size_t ii = 0; ii < B; ++ii) {
        q.submit([&](sycl::handler& h) {
            auto accessor = diagonal_block.get_access(h);
            h.parallel_for(sycl::range(B - ii - 1), [=](sycl::id<1> idx) {
                const matrix_size_t jj = ii + 1 + idx;
                accessor[jj][ii] /= accessor[ii][ii];
            });
        });
        q.submit([&](sycl::handler& h) {
            auto accessor = diagonal_block.get_access(h);
            h.parallel_for(sycl::range(B - ii - 1), [=](sycl::id<1> idx) {
                const matrix_size_t jj = ii + 1 + idx;
                for (matrix_size_t kk = ii + 1; kk < B; ++kk)
                    accessor[jj][kk] -= accessor[jj][ii] * accessor[ii][kk];
            });
        });
    }
}
```

6 Results

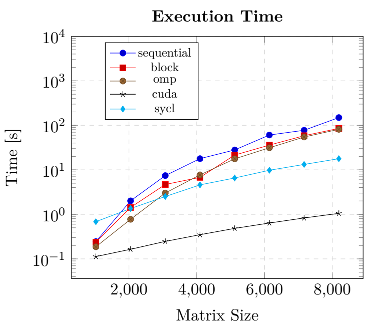
These results were obtained by calculating the average between of 3 distinct executions for each algorithm and each matrix size, in a machine with a 5.15.0-60-generic Kernel, running an Intel® Core™ i7-9750H CPU (Coffee Lake) with 6 cores and 12 threads with Hyper-Threading @ 2.60 GHz, and a GeForce GTX 1650 Mobile with Max-Q design (rev a1) with 1024 CUDA cores, a base clock speed of 1020 MHz and a boost clock speed of 1245 MHz, along with 4GB of GDDR5 memory.



(a) Block Size of 128

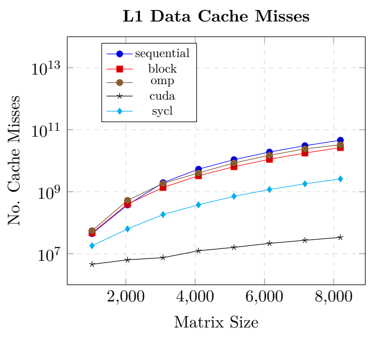


(b) Block Size of 256

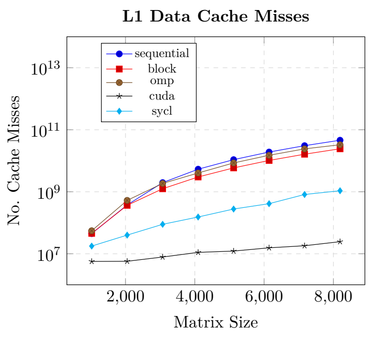


(c) Block Size of 512

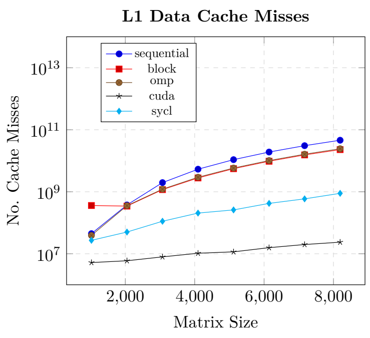
Figure 5: Execution time for the LU decomposition



(a) Block Size of 128

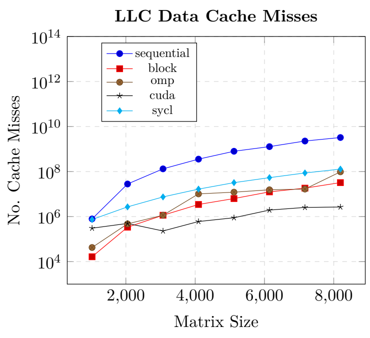


(b) Block Size of 256

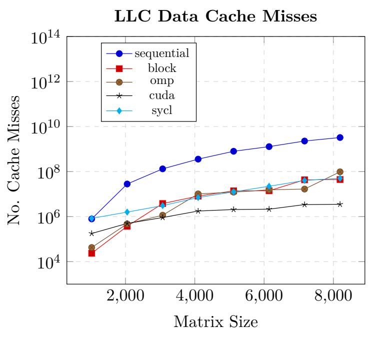


(c) Block Size of 512

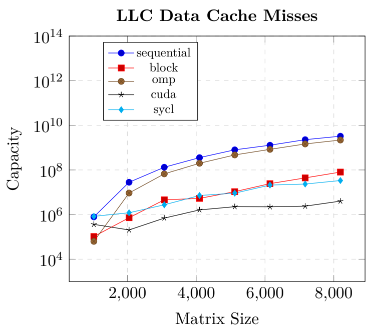
Figure 6: L1 cache misses during LU Decomposition



(a) Block Size of 128

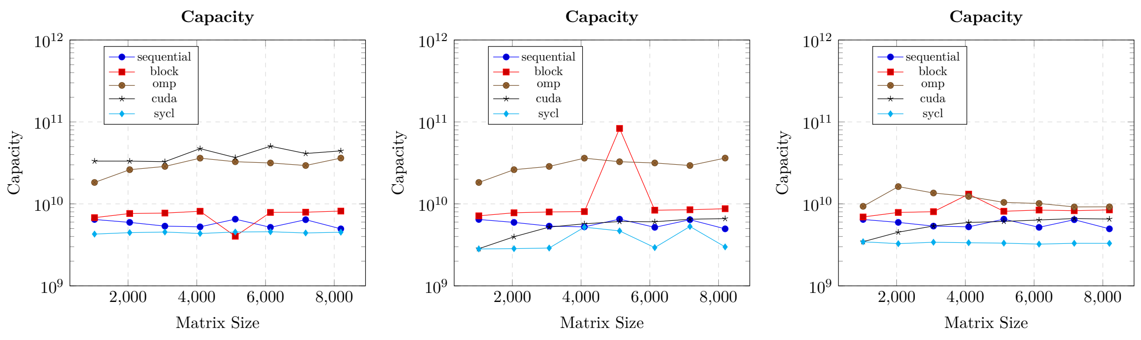


(b) Block Size of 256



(c) Block Size of 512

Figure 7: LLC (Last Level Cache) cache misses during LU Decomposition



(a) Block Size of 128 **(b)** Block Size of 256 **(c)** Block Size of 512

Figure 8: Capacity of the algorithms executing the LU Decomposition

6.1 Analysis

After collecting the results it is clear that the CUDA implementation was the fastest. As expected, it is then followed by SYCL and OpenMP in performance terms.

In this context, the single-core versions are much slower. Even though, there is a portion of code that must be executed sequentially, these 2 versions suffer the impact of not being able to parallelize the computation. This effect is especially relevant in the basic version, where for larger matrices, the total number of data cache misses increases by almost 2 orders of magnitude, considering the 128 and 256 block size versions.

Still regarding this matter, when testing the performance of the different block-based versions it was possible to notice that the L3 cache plays a significant role, especially when it comes to execution times. Here, the size of the block is an important factor since it determines whether a block or a sub-matrix, will fit in the cache or not. If the latter is not the case that means that it must fetch data from the main memory much more frequently. This introduces a significant overhead because main memory data, which usually are DRAM chips, besides being further away from the CPU, may not be instantly available due to memory refresh.

Considering this, the GPU chips offer a clear advantage when processing matrix information. This conclusion can be derived from several factors. The first is the much greater amount of threads available in GPUs when compared to CPUs, whereas in CPUs there are at most tens of threads, on the other hand, GPUs have thousands of threads taking parallel algorithms to another dimension, particularly, those involving matrices such as LU decomposition. Another aspect is that the GPU offers Video random-access memory (VRAM), memory dedicated to the GPU cores, which has, as mentioned in the section 5.2, a higher bandwidth compared to RAM and makes the read and write operation significantly faster.

Another important aspect is the differences between the results of SYCL and CUDA versions. Even though both versions execute on the same chip, CUDA is faster by at least 1 order of magnitude. Theoretically, there is no reason for this to be the case, but in reality, there are differences between the environments in which each version executes.

The first relevant aspect we found was the fact that CUDA permits a much greater tunning than SYCL. In the latter, we were not able to make use of the dedicated GPU VRAM, known as shared memory as it can be visible in Figure 4.

Another difference is related to the distinct CUDA versions used by each version. In our setup, the

CUDA runs on the more recent 12.1 version, while SYCL uses the 11.8 version - meaning that they execute on distinct major releases. Not only that, but the fact that each version uses a different compiler can have an impact. Although both programs are intended to execute the same task, compilers apply different optimization strategies and ABIs are incompatible, which naturally leads to very different binaries being produced.

The last topic to talk on this section is scalability, visible in figure 8. It is important to note that the capacity is only considering CPU instructions, so including the GPU instruction would, in theory, give a higher capacity score to the algorithms executed in the GPU.

The winner again is CUDA which had the smaller slop in the execution with a block size of 128. This is due to the fact that CUDA language gives direct access to GPU functionalities, providing the best possible environment for parallel execution.

We also noticed that the OpenMP version came in a close second and showed a constant capacity across all block sizes, however, it is limited by the physical design of the CPU.

7 Conclusions

With this work, we have tested the impact that both software and hardware can have on the execution of the LU factorization algorithm. As we have stated before, this is a relevant algorithm with many applications in real-world scenarios.

The results obtained show that the GPU offers overall better performance when it comes to parallel execution. This boils down to the GPU being designed with matrix information in mind. One of the consequences of parallelism is the need for a larger bandwidth to cope with a large amount of data being processed simultaneously.

However, it comes with the challenge of how to provide data in the right place at the right time. The main objective is to guarantee that all threads are doing the most amount of parallel work, avoiding idleness, and assuring that the data layout does not provoke access conflicts.

In spite of that, GPUs are crucial hardware components that accelerate computing when dealing with large amounts of data. This is particularly useful to solve linear systems that are used, among other currently popular domains, in machine learning and data analysis.

A Development Environment Installation and Setup

This appendix has the intent to demonstrate what is the development environment of this project. We will detail the architectures supported by this work and give a brief overview of how each component interacts with the remaining ones.

A.1 Architectures Supported

Each of the different versions shown is intended to be executed on different hardware.

The OpenMP version is meant to be executed on any CPU as long as the compiler provides support to the corresponding architecture. On the other hand, both CUDA and SYCL versions are expected to be executed on NVIDIA GPUs that are compatible with CUDA.

A.2 Software Dependencies

Before installing the dependencies it is important to ensure that the NVIDIA drivers are installed correctly. The installation of both drivers and proper project dependencies varies substantially across platforms. Given this, and for brevity's sake, it is up to the user to find out how to install those on its machine.

The dependencies required to run this project are the following:

1. GNU Make, this is the build system of the project
2. C++ compiler compatible with C++20 and OpenMP 4.5 or later
    - The only tested compiler was `g++`, however other compilers should work.
3. `nvcc`, this is NVIDIA's CUDA compiler.
4. `icpx`, this is Intel's Data Parallel C++ compiler which implements the SYCL specification.
    - Our implementation uses the SYCL 2020 specification.
    - SYCL support for the CUDA backend requires the installation of an extension [6].

A.3 Usage

The project is based on GNU Make. Thus, the building process is as simple as executing the following commands:

```
make [all | lu | lublk | luomp | lucuda | lusycl]   # Builds a target
./bin/lu.out       # Serial LU decomposition
./bin/lublk.out    # Block-based serial LU decomposition
./bin/luomp.out    # Block-based parallel LU decomposition using OpenMP
./bin/lucuda.out   # Block-based parallel LU decomposition using CUDA
./bin/lusycl.out   # Block-based parallel LU decomposition using SYCL
make clean         # Removes all the produced executables
```

[6]https://developer.codeplay.com/products/oneapi/nvidia/2023.1.0/guides/get-started-guide-nvidia

However, if the user wishes to build the `lusycl` target, it must first setup the environment[7], that is achieved by executing the command below:

```
source /opt/intel/oneapi/setvars.sh --include-intel-llvm
```

This will execute a script bundled with `icpx` that activates all the necessary components required to build SYCL applications.

[7]https://developer.codeplay.com/products/oneapi/nvidia/2023.1.0/guides/get-started-guide-nvidia#set-up-your-environment

References

[1] A. C. De Melo, “The new linux’perf’tools,” 2010.

[2] OpenMP Architecture Review Board, “OpenMP application program interface version 4.5,” 2015.

[3] N. Corporation, *CUDA Documentation*. NVIDIA Corporation, 2021.

[4] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. 01 2021.

[5] T. K. Group, *SYCL Documentation*. The Khronos Group, 2021.