Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Implementing a Cholesky decomposition using SYCL

Michal Bloch

Course of Study:	Informatik
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	Marcel Breyer, M.Sc.
Commenced:	September 5, 2024
Completed:	March 5, 2025

Abstract

PLSSVM, an LS-SVM implementation, now only uses the Conjugate Gradient algorithm for solving a set of linear equations. However, for an ill-conditioned matrix, it especially gets into trouble, as the converged solution drifts away from the actual solution due to rounding errors. Therefore, this thesis implements a different solver, e.g., the Cholesky Decomposition, which will be implemented in SYCL. We will implement multiple variations of the Cholesky Decomposition algorithm, including a blocked version, and utilize many different features of SYCL. The focus will primarily be on the fastest implementations. In the end, the fastest implementation will be integrated into PLSSVM alongside a Forward and Backward Substitution implementation for solving the set of linear equations. We will conclude with a runtime comparison between the implementations, a comparison of our best Cholesky Decomposition with the Conjugate Gradient using a dataset and a small discussion about numerical errors.

Kurzfassung

PLSSVM, eine LS-SVM-Implementierung, verwendet jetzt nur noch den Conjugate Gradient Algorithmus zur Lösung eines Satzes linearer Gleichungen. Bei einer schlecht konditionierten Matrix gerät es jedoch besonders in Schwierigkeiten, da die konvergierte Lösung aufgrund von Rundungsfehlern von der tatsächlichen Lösung abweicht. Daher implementiert diese Thesis einen anderen Solver, z.B. die Cholesky-Zerlegung, die in SYCL implementiert wird. Wir werden mehrere Variationen des Cholesky-Zerlegungsalgorithmus implementieren, einschließlich einer blockierten Version und viele verschiedene Funktionen von SYCL nutzen. Der Schwerpunkt wird hauptsächlich auf den schnellsten Implementierungen liegen. Am Ende wird die schnellste Implementierung in PLSSVM integriert, zusammen mit einer Forward- und Backward Substitution zur Lösung des Satzes linearer Gleichungen. Wir werden mit einem Laufzeitvergleich zwischen den Implementierungen, einem Vergleich unserer besten Cholesky-Zerlegung mit dem Conjugate Gradient unter Verwendung eines Datensatzes und einer kurzen Diskussion über numerische Fehler abschließen.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

A *Support Vector Machine* (SVM) is a machine learning model that creates a hyperplane which best separates data points based on their classification, making it a *maximum-margin* hyperplane. After training the model, it can then for example on a binary classification task guess the classification of a never-before-seen data point by determining on which side of the hyperplane the data point lies. SVMs are a supervised model, meaning training the model involves having a set of input-output examples. In this thesis, we will focus on the *Least Squares Support Vector Machine* (LS-SVM). The algorithm, in essence, states the training problem as a set of linear equations, while classical SVMs state it as a convex quadratic problem. This has the benefit of making the problem highly parallelizable, as algorithms solving the convex quadratic problem, such as *Sequential Minimal Optimization*, are tricky to parallelize.

For the reasons above, Alexander Van Craen and Marcel Breyer started a project to create an LS-SVM implementation dubbed *Parallel Least Squares Support Vector Machine* (PLSSVM) [VBP22]. PLSSVM first transforms the classification task into a set of linear equations and then uses a solver to find a solution. For the solver, they opted to use the *Conjugate Gradient* (CG) algorithm. It is an iterative algorithm, meaning the algorithm converges to a solution for the linear equation after enough steps. If the square matrix of the set of linear equations is ill-conditioned, then the speed of convergence becomes slower. Theoretically, the algorithm should converge to the correct solution after n steps, where n is the size of the matrix created from the linear equations. Yet, in practice, this does not always happen, as rounding errors occur based on which floating-point type is chosen for the algorithm, making CG diverge to either a result close to the solution or a result that is completely different from the solution if the matrix is very ill-conditioned.

This motivates to use a direct method for solving the set of linear equations that is more numerically stable and less sensitive to conditioning, which makes the *Cholesky Decomposition* a strong contender and so was also chosen.

In this thesis, we will develop a Cholesky Decomposition implementation using the SYCL [25c] backend, which will be used for PLSSVM, while also ensuring that the implementation is efficient in terms of computation time and memory usage. The implemented Cholesky Decomposition will then be compared to the current CG implementation in terms of speed, with the motivation being the accuracy of the solution. This thesis achieves this goal by taking a simple sequential implementation directly from the formulas of the Cholesky Decomposition algorithm to a parallel implementation, while making modifications on how the algorithm approaches the data to improve speed. This thesis will also contain introductions to optimization techniques as they are first used.

The structure of this thesis will be now presented. We begin with the Background. In this chapter the thesis will describe the Cholesky Decomposition theorem and also a blocked Cholesky Decomposition theorem as it will be implementated in a later chapter. This chapter also gives context on important basics to the GPU which will be important to understand the optimizations

used in the implementation of the blocked Cholesky Decomposition. Also here is a primer of all used SYCL functions and a description of the code structure of PLSSVM to understand how it is structured and where to add the blocked Cholesky Decomposition implementation.

The next chapter is the Related Work and there we will look at what others did in context of implementing the Cholesky Decomposition in SYCL.

In the next chapter Sequential Cholesky Decomposition using the CPU, we will take the basic algorithm for performing the Decomposition and extend it to be inplace and transform it into a different structure to prepare it for parallelization. Also we will take a short look at saving memory.

The chapter afterwards is then Parallel Cholesky Decomposition using the GPU, which is about the implementation of the Cholesky Decomposition using SYCL on the GPU. Here we will start of with a basic implementation 01 and work our way upwards to 06 with in this chapter introducted optimizations to reach better speeds. Also this chapter describes the blocked Cholesky Decomposition implementation variant which further increases the speed of the algorithm.

In the sixth chapter Implementation into PLSSVM, we will go into additional requirement for the Cholesky Decomposition to be successfully used in PLSSVM. We then describe the integration of all the part and the steps taken to modify PLSSVM to include the system with the Cholesky Decomposition.

In the next chapter Evaluation and Results, we will describe the setup used for measuring the performance of the many implementations, including look at numerical errors and how CG was compared to the Cholesky Decomposition. We then also present the measured results.

For the last chapter Conclusion and Outlook, we will summarize the results and the thesis in total and present a outlook for potential work.

2 Background

As the title of this thesis suggests, the thesis revolves around the Cholesky Decomposition. So we will start of with its basics.

2.1 Cholesky Decomposition

The Cholesky Decomposition is used in a wide variety of fields that require a solution to a set of linear equations, as it is a relatively simple algorithm and numerically stable.

Theorem 1 (Cholesky Decomposition in $\mathbb{C}$)
Let $A \in \mathbb{C}^{n\times n}$ with $n \in \mathbb{N}$, $A = A^*$ (Hermitian), and $\forall z \in \mathbb{C}^n \setminus \{0^n\} : z^*Mz > 0$ (positive-definite). Then a decomposition of the form

$$A = LL^*$$

where L is a lower triangular complex matrix with real and positive diagonal entries, is called the Cholesky Decomposition. Note: the A^* and z^* operators are the conjugate transpose of a matrix and a vector.

The theorem can be used for complex matrices, but for our purposes, we will only look at $A \in \mathbb{R}^{n\times n}$. So the theorem can be simplified.

Theorem 2 (Cholesky Decomposition in $\mathbb{R}$)
Let $A \in \mathbb{R}^{n\times n}$ with $n \in \mathbb{N}$, $A = A^T$, and $\forall z \in \mathbb{R}^n \setminus \{0^n\} : z^TMz > 0$. Then a decomposition of the form

$$A = LL^T$$

where L is a lower triangular real matrix with positive diagonal entries, is called the Cholesky Decomposition.

A useful feature of the Cholesky Decomposition is that the decomposition is unique for every matrix A satisfying its requirements. L can thus be calculated using Equation 2.1 and Equation 2.2 with $1 \le i, j \le n$.

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2} \tag{2.1}$$

$$L_{ij} = \frac{1}{L_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk} \right) \quad \text{for } i > j \tag{2.2}$$

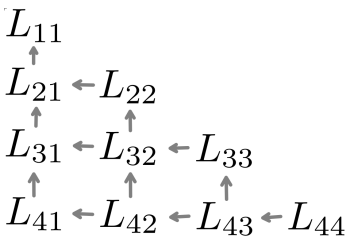


Figure 2.1: Example $L \in \mathbb{R}^{4\times4}$ dependencies in Equation 2.1 and Equation 2.2. Not every single dependency is marked with an arrow, as this would make the image unreadable. The arrows represent a dependency "direction".

The equations are split into two cases where L_{jj} handles the diagonal entries and L_{ij} handles the inner lower triangle matrix. Note that we always have $i \geq j$, as L is a lower triangular matrix. As we see with the equations, we get a time complexity of $O(n^3)$, since we have to go through each i and j with $i \geq j$, meaning we get a time complexity of $O(n^2)$, and calculating each L_{ij} itself requires going through up to j. This results in a total time complexity of $O(n^3)$.

While it would be possible to calculate each L_{ij} in parallel, it would require a lot of redundant calculations. A better approach is to utilize dynamic programming by reusing L_{ij}, but this forces us to adhere to a specific order. Applying Equation 2.1 and Equation 2.2 to an example in Figure 2.1, we see that the order is top-down and left-right. Such can be represented in algorithm 2.1. In

Algorithm 2.1 Perform the Cholesky Decomposition on A

Require: $A \in \mathbb{R}^{n\times n}$ positive definite and symmetric
Ensure: $L \in \mathbb{R}^{n\times n}$ lower triangular matrix with $A = LL^T$
  for $i \in (1, ..., n)$ **do**
    for $j \in (1, ..., i)$ **do**
      $s \leftarrow 0$
      for $k \in (1, ..., j-1)$ **do** // Summation for both cases
        $s \leftarrow s + L_{ik}L_{jk}$
      end for
      if $i = j$ **then** // Split if on the diagonal or not
        $L_{jj} \leftarrow \sqrt{A_{ii} - s}$
      else
        $L_{ij} \leftarrow \frac{1}{L_{jj}}(A_{ij} - s)$
      end if
    end for
  end for

the algorithm, we iterate through each i and j, splitting the logic based on whether we are at Equation 2.1 or Equation 2.2. However, we can reuse the sum, as the sum in Equation 2.1 becomes the sum in Equation 2.2 when $i = j$.

For actually solving a linear equation problem $Ax = b$, given a matrix A satisfying the requirements above and vector b. Due to Cholesky, we have $A = LL^T$, which gives us $LL^Tx = b \Leftrightarrow L(L^Tx) = b$. Substituting $L^Tx = t$, we get $Lt = b$. Notice L and L^T are just lower and upper triangular matrices

respectively. Thus, using the algorithm 2.2, we can compute x. We see L is independent of A, so if

Algorithm 2.2 Solve $Ax = b$ using the Cholesky Decomposition

Require: $A \in \mathbb{R}^{n \times n}$ positive definite and symmetric, $b \in \mathbb{R}^n$
Ensure: $x \in \mathbb{R}^n$
  $L \leftarrow$ Compute $A = LL^T$ using Cholesky Decomposition
  $t \leftarrow$ Compute $Lt = b$ using forward substitution
  $x \leftarrow$ Compute $L^T x = t$ using backward substitution

we precompute L, requiring $O(n^3)$, we can then solve for multiple b, making the algorithm only require $O(n^2)$ due to forward and backward substitution.

2.1.1 Forward and Backward Substitution

Let $Lx = b$ and $Ux = b$ be linear equation problems with $L \in \mathbb{R}^{n \times n}$ a lower triangular matrix, $U \in \mathbb{R}^{n \times n}$ an upper triangular matrix, and $b \in \mathbb{R}^n$ a vector. Then a solution for the former can be found using the algorithm called *forward substitution*, and for the latter, *backward substitution*. Equation 2.3 describes forward substitution and Equation 2.4 describes backward substitution using $1 \leq i \leq n$.

$$x_i = \frac{b_i - \sum_{k=1}^{i-1} L_{ik} x_k}{L_{ii}} \tag{2.3}$$

$$x_i = \frac{b_i - \sum_{k=i+1}^{n} U_{ik} x_k}{U_{ii}} \tag{2.4}$$

For forward substitution, we see x_i goes from 1 to n, and for each x_i, we have to do a sum from 1 to i, making the complexity of the algorithm have $O(n^2)$. We see where the term *forward* comes from, since to calculate the whole x vector, we have to start from the top with x_1 and then go forward and substitute into x_2, x_3, etc. Analogously for the backward substitution. Using Equation 2.3, we can construct algorithm 2.3, which operates in place. Again, we utilize dynamic programming to

Algorithm 2.3 Forward Substitution

Require: $L \in \mathbb{R}^{n \times n}$ lower triangular, $b \in \mathbb{R}^n$.
Ensure: $x \in \mathbb{R}^n$, solving $Lx = b$.
  for $i \in (1, ..., n)$ **do**
    $b_i \leftarrow \frac{b_i}{L_{ii}}$
    for $k \in (i + 1, ..., n)$ **do**
      $b_k \leftarrow b_k - L_{ki} b_i$
    end for
  end for
  $x \leftarrow b$

avoid redundancy by calculating from x_1 till x_n. Each iteration of i subtracts b_i, which is already the solution calculated before the b_k loop, from each b_k further on. This erases the need for a sum,

since $b_i - \sum_{k=1}^{i-1} L_{ik}x_k$ from Equation 2.3 is already calculated when the algorithm approaches b_i. For the backward substitution, we have a similar algorithm 2.4; just some loops go in a different direction. We will implement a similar algorithm for the Cholesky Decomposition later.

Algorithm 2.4 Backward Substitution

Require: $U \in \mathbb{R}^{n\times n}$ upper triangular, $b \in \mathbb{R}^n$.
Ensure: $x \in \mathbb{R}^n$, solving $Ux = b$.
  for $i \in (n, ..., 1)$ **do**
    $b_i \leftarrow \frac{b_i}{U_{ii}}$
    for $k \in (i + 1, ..., n)$ **do**
      $b_k \leftarrow b_k - U_{ki}b_i$
    end for
  end for
  $x \leftarrow b$

2.1.2 Blocked Cholesky Decomposition

The above Standard Cholesky Decomposition formula calculates L using only scalars. It is also possible to look at $A = LL^T$ with $A^{n\times n}$ using blocks of values, making the decomposition take the form of Equation 2.5.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \tag{2.5}$$

Here we have $A_{11} \in \mathbb{R}^{m\times m}$ with $m \leq n$, $A_{12} \in \mathbb{R}^{n-m\times m}$, $A_{21} \in \mathbb{R}^{m\times n-m}$, and $A_{22} \in \mathbb{R}^{n-m\times n-m}$. The entries L_{ij} have the same size as A_{ij}. Due to performance reasons discussed later and to avoid confusion, we change L into an upper triangular matrix (a transpose changes it back) and ignore A_{21}, since it is equal to A_{12}^T. The zeros in L are meaningless to store, so we also leave them out now. This results in Equation 2.6.

$$\begin{bmatrix} A_{11} & A_{12} \\ & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11}^T & \\ L_{12}^T & L_{22}^T \end{bmatrix} \begin{bmatrix} L_{11} & L_{12} \\ & L_{22} \end{bmatrix} \tag{2.6}$$

This form can be used to perform the Cholesky Decomposition using the algorithm 2.5 [CJS+13]. As opposed to [CJS+13], we don't explicitly compute the inverse of L^T and then matrix multiply, as it is numerically unstable. Rather, in algorithm 2.5, we perform the forward substitution. To save memory, the calculated L matrices can just replace their respective entries in A as they aren't needed for the next steps. The algorithm begins by first splitting up A into A_{11}, A_{12} and A_{22}. We then get L_{11}, L_{12} and the next A for which the whole iteration starts again. Figure 2.2 displays an continuation execution. Using the computed L, the algorithm 2.1 can then be used to find a solution for each given b vector.

Algorithm 2.5 Decompose $A = L^T L$ using the Blocked Cholesky Decomposition

Require: $A \in \mathbb{R}^{n \times n}$ positive definite and symmetric, $m \in \mathbb{N}$ with $m \leq n$.
Ensure: $L_{11} \in \mathbb{R}^{m \times m}$ upper triangular, $L_{12} \in \mathbb{R}^{n-m \times m}$, $A'_{22} \in \mathbb{R}^{n-m \times n-m}$
  while $n > m$ with n from $A^{n \times n}$ **do**
    $A_{11}, A_{12}, A_{22} \leftarrow$ Split A using m into sizes defined in Equation 2.6
    $L_{11} \leftarrow$ Decompose $A_{11} = L_{11}^T L_{11}$ using the Standard Cholesky Decomposition
    $L_{12} \leftarrow$ Compute $L_{11}^T L_{12} = A_{12}$ with forward substitution for each column of L_{12} and A_{12}
    $A \leftarrow A_{22} - L_{12}^T L_{12}$ // Next iteration A shrinks by m
  end while
  $L_0 \leftarrow$ Decompose $A = L_0^T L_0$ using the Standard Cholesky Decomposition
  $L \leftarrow$ Combine all previously calculated L_{ij} and L_0 into a final matrix L

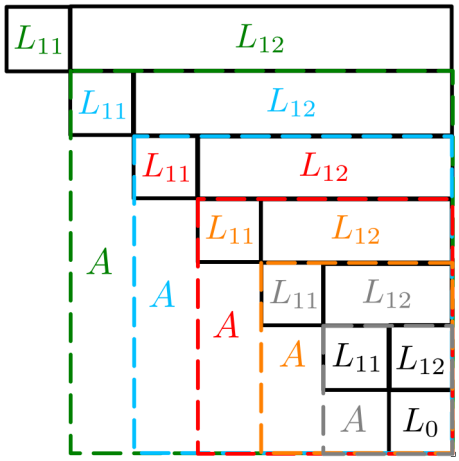


Figure 2.2: Example of produced L matrices for a big matrix A. The first iteration produces the green matrices, the second the blue, and so on. We also see the algorithm can be performed entirely in place, as it fits in A.

2.2 Graphics Processing Unit

A Graphics Processing Unit (GPU) is a specialized electronic circuit mainly used for computer graphics. The graphics pipeline includes many tasks that can be solved independently from each other, meaning to have a high throughput, the GPU is highly parallel. But at the cost of latency, meaning serial tasks are better executed on the CPU in most configurations. This makes GPUs an invaluable component for computing large datasets, which is why we are mainly focusing on them.

2.2.1 Hardware

Figure 2.3 displays an example model of a GPU layout. Computation on the GPU is done by tiny processors called *streaming multiprocessors* (SM), which each operate concurrently and create, manage, schedule, and execute cores in groups. An SM can also execute multiple groups at once. Depending on the GPU, typically there are 32 cores in a group. For memory, each core has its own

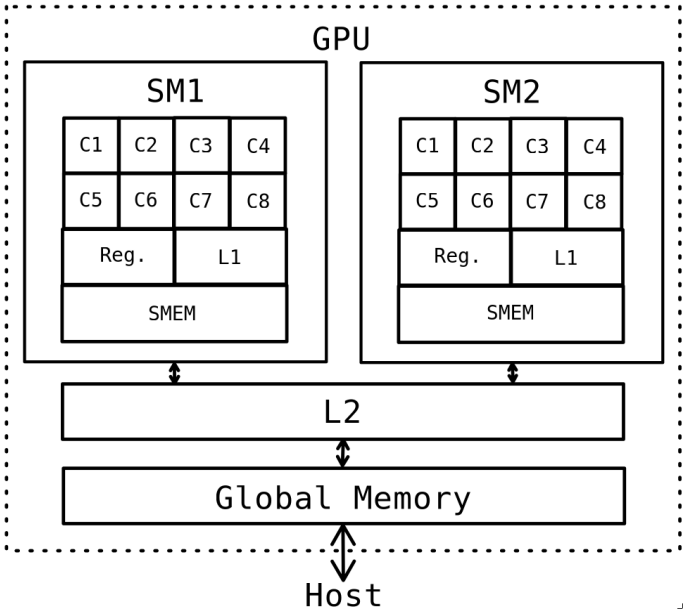


Figure 2.3: An example GPU model with two SMs, each with 8 cores. Each SM contains its own SMEM and its own set of registers, with an L1 cache that caches each load to the L2 cache. The L2 cache then caches each load from global memory. Global memory is the largest memory type and serves as the buffer for the host to read and write to.

set of registers, which, depending on the GPU, typically have at most 255 registers per core. If a given task's memory exceeds the register count, then there is a *register spill*, meaning different types of slower memory are used for further storage, thus it should be avoided if possible. For retrieving values to work on, the *Global memory* is used, which is a sort of main memory for the entire GPU. Each core has read-write access to it, but it is the slowest memory, making it best to just retrieve data values and then store the result. To alleviate some memory loading time, each SM contains an L1 cache, and for the whole GPU, there is the L2 cache, which both store values next to the loaded one, meaning global memory loading of sequential data is faster than random access. Global memory is also used as the buffer for the *host* to write its task data and, after GPU execution, to extract the results. In this context, the host means the CPU issuing the tasks to run.

Each SM also has a faster memory called *Shared Memory* (SMEM). Every core on the same SM has access to the same SMEM, meaning cores being executed by different SMs have differing SMEMs. SMEMs are split into sections called *memory banks*. The amount depends on the GPU, but typically it is 32. Only one value may be retrieved from a memory bank at once. As previously stated, cores are executed in groups, meaning we should try to spread each core out to each memory bank, since if two cores retrieve values from the same memory bank, we get a *bank conflict*, which forces sequential retrieval. So it should be avoided as much as possible. Figure 2.4 shows an example model with 5 memory banks. Accessing $0, 5, 10, 15, 20$ at once within a group of cores executed by the SM would lead to a 5-fold bank conflict. An exception is broadcasting. If all cores access the same address, then the value is only read once and broadcasted to all cores.

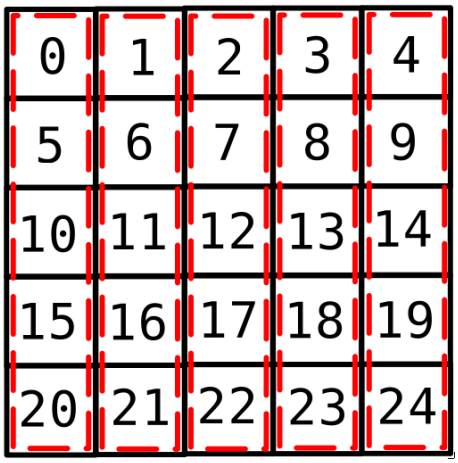


Figure 2.4: Example of a 5×5 shared memory layout with 5 memory banks. As shown in the figure, accessing 0, 1, 2, 3, and 4 at the same time is possible, but accessing 0 and 5 results in a bank conflict, making a core first load 0 and then 5.

2.3 SYCL

For computing large tasks, modern computing systems often rely on heterogeneous devices such as CPUs, GPUs, FPGAs, and so on. Such a setup allows for using a better-fitted device for computing a task, such as a GPU in this thesis. Programming for such devices tends to be difficult due to the complexity of managing multiple device backends. For example, CUDA, which is used for programming NVIDIA GPUs, can only be used for said NVIDIA GPUs, and code that runs on the CPU tends to have to be separate from the GPU code. There is already OpenCL that supports multiple backends, but its boilerplate-heavy and old C code makes it an unattractive choice. However, there are also new contenders like Kokkos.

SYCL (pronounced ”sickle”) is an abstraction layer made with the main goal of solving the above problem. It is open, uses modern C++ features, and provides a single-source programming model for heterogeneous devices, enabling developers to write host and device code in a unified codebase. As SYCL itself is just a specification, multiple implementations exist, with notable ones being *Intel oneAPI DPC++* [25f] and *AdaptiveCpp* [25a]. For this thesis, we chose AdaptiveCpp as it is community-driven and includes a *just-in-time* (JIT) compilation, making kernel performance adjust best itself at the cost of having the first kernel launch not perform as well.

2.3.1 SYCL Programming Model

In this thesis, we will control the GPU using SYCL. As SYCL abstracts from the device used, it introduces a different view of the device. Going forth, we'll use SYCL's terminology, so here we will describe how SYCL interacts with a GPU. SYCL defines a hierarchy of 4 ranges called, in order, *ND-range*, *work-group*, *sub-group*, and *work-item*, as displayed in figure 2.5[1]. Each work-item is a

[1] https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-2/thread-mapping-and-gpu-occupancy.html

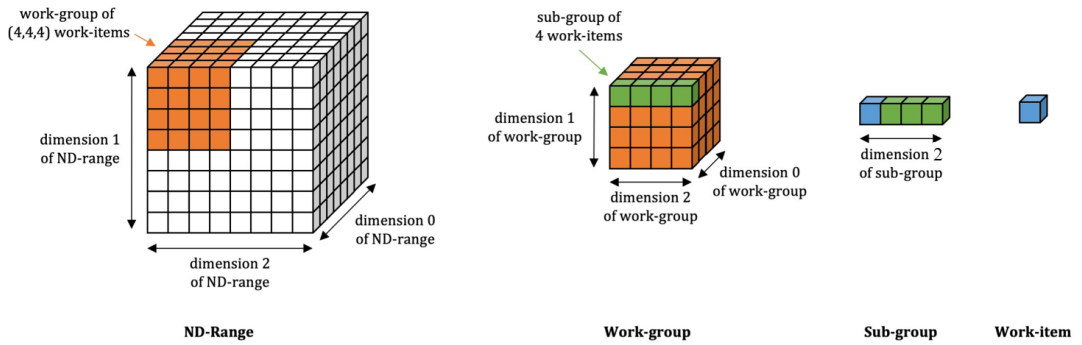


Figure 2.5: An example workload: an $8 \times 8 \times 8$ ND-range, $4 \times 4 \times 4$ work-group, 4 sub-groups, and 1 work-item.

core of an SM, and a sub-group is the group of cores that an SM executes at once. The SMEM is called *local memory*. The work-group is a group of work-items that all belong to the same SM and is allocated a section of the local memory, meaning an SM could have multiple work-groups run at the same time as long as each work-group can allocate itself enough resources on the SM. Note that a work-group is not a sub-group, as a sub-group is a group of work-items executed by the SM at the same time, while a work-group is a collection of work-items sharing a local memory section. Preferentially, the sub-group size should divide the work-group size, because if not, then we lose performance as the sub-group will only be able to execute a part of itself but not entirely, since there is nothing to do. Just like in the GPU, work-items that are part of differing work-groups cannot access each other's local memory. The ND-range doesn't have a GPU equivalent, but typically it represents the whole data range. Using the characteristics described, a typical SYCL kernel looks like this:

1. Copy a chunk of data from global memory in an ND-range to the local memory of a work-group.
2. Copy required scalars from local memory to registers of a work-item.
3. Do work and store results in registers.
4. Store large intermediate results in local memory.
5. Copy results to global memory.

2.3.2 Used SYCL Functions and Objects

We will now go through all relevant SYCL objects and functions. Code presented in the text is C-like pseudocode, used only to clarify what the parameters and return values are.

Queue As mentioned above, SYCL can be used for multiple devices. So we first need to get a reference to the device that we want to use. This is achieved using `sycl::queue(selector, listOfProperties)`, which is an object that binds to a device and can be issued commands. `selector` is a function that takes in a device and returns a score on how well it fits our needs

defined in the function. The queue will bind itself to the device with the highest score. If multiple devices have the highest score, then one of them will be selected depending on the implementation. For our purposes, we will always be using `sycl::gpu_selector_v` as we are targeting single GPU systems and we just want the GPU. `listOfProperties` can be passed using an initializer list and defines properties used to modify how the queue operates. In our case, we will first use `sycl::property::queue::in_order()` as each of our tasks passed into the queue depends on the results of the previous tasks, and secondly, we will use `sycl::property::queue::enable_profiling()` as we would like to measure the time each task requires.

Event The host and device operate asynchronously from each other, meaning the host just issues tasks and continues execution. SYCL also has the feature of dependency handling for multiple tasks dependent on each other. To coordinate the dependencies, SYCL uses events (`sycl::event`) so that the host can assign which task needs to be completed before starting the new queued-up task. For example, in listing 2.1, we have 3 tasks where the bottom two have to wait until the first one is finished. But for our use case, we don't use much of any events because of the previously mentioned `sycl::property::queue::in_order()`.

Wait Due to the asynchronous design of SYCL, before we access data from the device, we first need to wait (`q.wait()`). This can be done from the queue `q` as well as from the latest event `e`.

Data Management For putting task data on the device, first we need to allocate memory on the device. This is done using `memptr = sycl::malloc_device<T>(size, q)`. `memptr` is an address to the first memory position of the allocation, which we can treat like a C pointer **but only in the GPU task**, as the address points to an undefined location for the host. `T` is the primitive type we would like to use. In our case, we will be using `float` and `double`. `size` is how much of `T` we would like to allocate. `q` is the queue to associate with the allocated memory region. To clean up the allocation, the function `sycl::free(memptr, q)` is used.

To actually get data onto or from the device, we use `e = q.memcpy(dest, src, sizeInBytes)`. `e` is an event, since `memcpy` is asynchronous, meaning the host needs to call `wait` before trying to access the copied data in the host memory. `q` is the queue used to queue up the `memcpy`. `dest` is the first address of a memory array we want to copy to. Most of the time, on the first `memcpy`, we set it to the address of the allocated GPU memory (`memptr`), and after task execution, we call `memcpy` again, but this time `dest` is allocated memory on the host side. Listing 2.2 shows an example of this. `src` is then the first address of a memory array we want to copy from. Again, the first `memcpy` is probably an address on the host, and the second `memcpy` is going to be the address of the allocated GPU memory. `sizeInBytes` is the size of the memory array we want to copy. Here we need to be careful, as the methods use `sizeInBytes`, so using the previously defined variables, it should be `sizeInBytes = size * sizeof(T)`.

SYCL also provides a different method of data management using *buffers*, which, in short, is a container for data that can be accessed from both a device **and** the host. The SYCL runtime manages the background to make this possible. However, we will use the previously described explicit model, as it has fewer wrappers and gives more control over how and when data is managed.

Simple Tasks For actually running something on the device, SYCL defines multiple task types. We won't be using them in the final implementation, but they are used for the first few. To create a task that is only executed once, we use `e = q.single_task(event, task)`. `event` is a

dependency, meaning what task in the queue should happen before, as seen in listings 2.1. `task` is a function that performs our task on the device. It can freely read and write from the previously defined pointer `memptr`, as it is our global memory. For GPUs, `single_task` should be avoided as it's very inefficient. We will discuss more details later.

A second task type, which is more GPU appropriate, is `e = q.parallel_for(range, event, task)`, which runs multiple times. `range` is created using `sycl::range<Dim>(c, b, a)`, with `Dim` being a dimension and `a`, `b`, `c` being the sizes each dimension gets. It corresponds to ND-range in the SYCL programming model but lets the SYCL runtime handle the work-group size. This parameter makes the task run $a \times b \times c$ times and as much as possible in parallel. `task` is a function that receives as input an object of type `sycl::id<Dim>`. In the `id` object, the ID of the work-item is found, containing the index for each dimension, which can be retrieved using `[i]` with dimension index `i`. In the context of a GPU, `a` defines the size of the "first" dimension, meaning `[0]` and `[0] + 1` are subsequent cores. This is important for optimization. Listings 2.3 show an example using `parallel_for` that doubles each value in a matrix loaded in global memory. It's not really usable, since just loading the matrix to the device will probably take more time, but it's a really simple example.

Advanced Tasks While for sufficiently simple tasks the task types above are enough, for more complex tasks we would like to specify the work-group size so that we can use local memory. To achieve this, we first call `e = q.submit([&](sycl::handler &cgh) {...})`. `cgh` is a configuration object for configuring the task and then subsequently calling it. With it, we can call `cgh.depends_on(event)` for specifying a dependency and `sycl::local_accessor<T, SubDim>(range, cgh)` for specifying that we use local memory and how much should be allocated. `SubDim` here is the number of dimensions we'll use for accessing local memory, and `range` defines the size for each dimension to take.

For the task itself, we only use `cgh.parallel_for(nd_range, task)`, which, as before, executes a task multiple times and, if possible, in parallel. `nd_range` is of type `sycl::nd_range` and specifies the ND-range for the first parameter, while the second parameter is the size for each work-group. `task` is then a function that runs the task, which also receives an object of type `sycl::nd_item<Dim>` containing the index on the ND-range from the work-group perspective (accessed using `get_group`) and also the index on the work-group from the work-item perspective (accessed using `get_local_id`).

Using local memory, there are cases where we would like to have all work-items in a work-group be done up to a certain point. For example, waiting until all work-items are done loading data from global memory. This can be done using `group_barrier`. An example of using `parallel_for` and `group_barrier` can be seen in listings 2.4, where we transpose a matrix. It works by first having each work-item load a chunk of global memory into the work-items' shared local memory, waiting until all are loaded in their relevant spots, and then finally writing it back into global memory using a transposed local memory. This may seem redundant, but due to memory coalescing, it is actually faster than just directly writing global memory to global memory. More details later.

Listing 2.1 Two independent tasks depend on the first task using SYCL events.

```
// Creation of queue q and loading data onto the GPU
sycl::event e = q.single_task(...);
q.single_task(e, ...);
q.single_task(e, ...);
```

Listing 2.2 Example of copying data onto the device, doing some work, and then copying the data, which is now the result, back from the device.

```
// Creation of queue q and data dataptr of size n of type float
float* memptr = sycl::malloc_device<float>(n, q);
sycl::event e = q.memcpy(memptr, dataptr, sizeof(float) * n);
// Do work on the data on the device using memptr
e.wait(); // Wait till memory won't change
q.memcpy(dataptr, memptr, sizeof(float) * n).wait();
sycl::free(memptr, q);
```

Listing 2.3 Example task of doubling each element of a matrix, showcasing the simple `parallel_for`.

```
// Creation of queue q and loading matrix of size nxn onto the GPU
q.parallel_for(sycl::range<2>(n, n), [=](sycl::id<2> idx) {
  memptr[n * idx[1] + idx[0]] *= 2;
}).wait();
```

Listing 2.4 Example task of transposing a single-precision matrix using local memory. This uses `block` to create and access the local memory. The more advanced `parallel_for` now also accepts the work-group size, which we have as the local memory block. So, every work-item handles one entry. Addressing is done by the linear function $n \cdot i + j$; however, due to the inclusion of the work-group, we again have to use some addressing, which is luckily just the same function, as the work-group is also square. This results in the two times recursive linear function seen in the listing. Transposition is then done by just flipping `locali` and `localj`, and also `groupi` and `groupj`. Note: This isn't an in-place transposition, as we use `memptr1` as input and `memptr2` as output, which shouldn't overlap.

```
// Creation of queue q and loading matrix of size nxn onto the GPU. n is divisible by 32
q.submit([&](sycl::handler &cgh) {
  sycl::local_accessor<float, 2> block(sycl::range(32, 32), cgh);
  cgh.parallel_for(
    sycl::nd_range{ sycl::range(n, n), sycl::range(32, 32) }, [=](sycl::nd_item<2> idx) {
    const auto groupi = idx.get_group(0), groupj = idx.get_group(1);
    const auto locali = idx.get_local_id(0), localj = idx.get_local_id(1);
    block[locali][localj] = memptr1[n * (groupi * 32 + locali) + (groupj * 32 + localj)];
    sycl::group_barrier(idx.get_group());
    memptr2[n * (groupj * 32 + locali) + (groupi * 32 + localj)] = block[localj][locali];
  });
}).wait();
```

2.4 PLSSVM Codebase

As stated in chapter 1, PLSSVM is a LS-SVM implementation supporting multiple device frameworks such as SYCL, which this thesis focuses on. Currently, the workflow is:

1. Load in a train and test dataset.
2. Define parameters to be used for the SVM.
3. Fit the SVM using the parameters on the train dataset.
4. Estimate the accuracy of the model on the training data.

The project contains multiple bindings to perform these actions, such as with Python, C++, and on the command line interface. In the codebase, this is implemented by splitting the interfaces into multiple implementations, but all are using the same core implementation written in C++. The core contains implementations of all device backend types. Overall, all the core works using the `plssvm::csvm` class. Its task is to be a base class abstracting from a specific backend implementation. It's responsible for most operations, most notably fitting, predicting, and scoring.

The `fit` operation creates the set of linear equations and passes it further to a dedicated method for solving the problem, `solve_lssvm_system_of_linear_equations`. This method, in turn, checks if the matrix A for $Ax = b$ fits into the device and calls `assemble_kernel_matrix` to place the matrix into the device. The operation is also dependent on a solver type, for which PLSSVM defines two. These are `cg_explicit`, which fully packs the matrix A into the device, and `cg_implicit`, which allows for larger-than-device memory matrices A. But that, as a trade-off, makes the later called CG slower. `assemble_kernel_matrix` is handled by a child class deriving from `plssvm::csvm`, for which it defines how the backend type should place the matrix into the device. Then finally, the also contained CG implementation in `plssvm::csvm` is called to find solutions for multiple b vectors that are on the host, which is where this thesis will mostly focus. CG gets called multiple times depending on the SVM parameters given, and then the resulting model is placed into `plssvm::model`, which stores the model parameters and can be saved on the file system. Here is a simplified summary:

1. Call `fit`
   - 1.1 Load in the matrix A
   - 1.2 Call `solve_lssvm_system_of_linear_equations`
     - 1.2.1 Check if A fits in the device (depending on `cg_implicit` or `cg_explicit`)
     - 1.2.2 Call `assemble_kernel_matrix`, which runs device-specific implementation
     - 1.2.3 Call `conjugate_gradients`
   - 1.3 Save as a `plssvm::model`

Currently, PLSSVM does not handle multiple solver types for solving systems of linear equations differently. To insert another solver type into PLSSVM, we need to address all the switch cases that introduce a new unhandled case. When implementing the switch case, if we explicitly assemble the kernel matrix, we can take the option to do so. This allows PLSSVM to load the matrix A into GPU memory and then provide it to us. To implement a new parameter in PLSSVM, we can follow the approach used for existing parameters. This includes defining the parameter in the command line parser, creating an IGOR named parameter and passing it into the fit function. With the new solver type, we can then simply call our Cholesky Decomposition implementation alongside the existing CG call site.

3 Related Work

The Cholesky Decomposition is a widely used algorithm, leading to numerous implementations across various platforms. Due to the popularity of NVIDIA GPUs, the go-to implementation for a dense Cholesky Decomposition on the GPU is found in the cuSolver [25b] library, which is written in CUDA. It contains the function `cusolverDnpotrf`, which, after specific setup for the cuSolver library, can be executed to perform the Cholesky Decomposition on a dense matrix. The library also provides an implementation for sparse matrices. However, a significant limitation of this library is its reliance on CUDA, making it vendor-specific and thus restricted to NVIDIA GPUs.

Several papers have explored variations of the Cholesky Decomposition. For instance, [BJN+23] implements *symPACK* for sparse matrices, specializing in solving the Cholesky Decomposition on a multinode cluster with a mix of GPUs and CPUs. This implementation utilizes the UPC++ library for communication, but for GPU operations, it relies on cuBLAS <cite> and cuSolver, which are also written in CUDA and thus limited to NVIDIA GPUs. Many other implementations follow a similar pattern.

To avoid CUDA, there is an implementation in OpenCL [BAB11]. Thanks to OpenCL, this implementation can run on non-NVIDIA GPUs and, similar to SYCL, even on devices that are not GPUs at all. However, the multiple implementations made by the authors seem to only cover the standard Cholesky Decomposition. Reformulating the algorithm to use blocks rather than single scalars can significantly enhance performance, a topic that this thesis will explore. The authors do however employ memory coalescing and use local memory for optimizing their implementations.

Another implementation written in OpenCL can be found in [18]. In contrast to the previous implementation, this one utilizes the blocked Cholesky Decomposition algorithm, which aligns with our approach. They also implement the Cholesky Decomposition function for the block triangular matrix in a similar manner. However, for splitting the triangular matrix, they use a variable split size. This can lead to issues with the compiler being uncertain about loop iteration counts, potentially resulting in no loop unrolling and a consequent loss of performance. Additionally, they adhere more strictly to the blocked Cholesky Decomposition algorithm than we do, first calculating the inverse of the top left block of the split after decomposing it, followed by matrix multiplication. We believe this step can be avoided using forward substitution, which should be faster than computing an inverse and performing a matrix multiplication, while also being more numerically stable.

A project called *MAGMA* (Matrix Algebra on GPU and Multicore Architectures) [TNLD10] is a collection of linear algebra libraries similar to cuSolver and cuBLAS. MAGMA includes implementations of various linear algebra routines, including the Cholesky Decomposition. The library provides full implementations of its functions in both CUDA and HIP, thus supporting NVIDIA and AMD GPUs. However, as of this writing, it does not fully support SYCL, which limits its applicability in heterogeneous computing environments.

Additionally, there is a project that provides a Cholesky Decomposition interface in SYCL called the `oneAPI Math Library` (oneMath) [25e] from Intel. This library offers the `potrf` function for both real and complex matrices, satisfying the requirements for the Cholesky Decomposition. Yet, under closer inspection, it doesn't actually implement the Cholesky Decomposition fully in SYCL but rather provides a SYCL binding to various Cholesky Decomposition implementations on other backends, such as CUDA. However they do provide some BLAS operations in SYCL <cite>, but it doesn't seem to have a Cholesky Decomposition implementation and it is unsupported by oneMath.

In summary, while there are several implementations of the Cholesky Decomposition across different libraries and platforms, an implementation of the Cholesky Decomposition solely in SYCL to the best of our knowledge has not been done recently yet.

4 Sequential Cholesky Decomposition using the CPU

To springboard ourselves to an implementation of a Cholesky Decomposition using SYCL, we first create a straightforward sequential algorithm, with which we can base our parallel implementation on. Algorithm 2.1 already fills that role. We also notice it is already, in a sense, in place. This is important, as we would like the algorithm to not store our entire input matrix twice. This can be seen by looking at the two cases where an "old" value of A_{ij} is needed, which is in the `if` $i = j$ `then ... else ...` statement. If we can confirm the algorithm hasn't previously set L_{ij} to some value and we are retrieving A_{ij}, then the algorithm is indeed in place. In the case of $i = j$, we retrieve A_{ii} and set L_{jj}. Since $i = j$, it's equivalent to retrieving A_{ij} and setting L_{ij}. In the other case of $i \neq j$, we directly retrieve A_{ij} and set L_{ij}. So, in other words, the `ij` loop just visits the whole A matrix once and updates each value accordingly, dependent on the value in the current A_{ij} and on the values previously calculated in the L matrix. This means A_{ij} values should still be in the matrix when needed and where not previously overwritten, leading to the algorithm being in place. So, in an implementation, we can substitute each use of A with L, and we just need to allocate L and L holds the input at the beginning.

4.1 Submatrix Cholesky

However, to parallelize an algorithm, we would like to have a range of values that need to be changed and, most importantly, don't depend on each other. This means there should be no order; otherwise, it's basically a sequential algorithm. In our in-order variant of the Algorithm 2.1, looking at the `i` loop with $i = 1$, we calculate L_{11} and use it for $i = 2$ with $j = 1$ in the $L_{ij} \leftarrow \frac{1}{L_{jj}}(A_{ij} - s)$ statement, leading to a dependency. This can be generalized by seeing that the reading of the diagonal element L_{jj} depends on the `i` loop state $i = j$, as that is where the value of L_{jj} was set. In other words, let i be a value between 1 and n. Then, each j loop iteration with $j < i$ depends on a previous `i` loop iteration with $i = j$, so here we are forced to be strictly sequential on the `i` loop. Looking at the `j` loop in Algorithm 2.1, we get a similar issue but in a different location. We'll again look to see if a `j` loop iteration depends on another `j` loop iteration, but this time with the same i. Let i be a value between 1 and n, and j be a value between 1 and i. Then $j \leq i$, and we choose $k = j - 1$. Then we see with $s \leftarrow s + L_{ik}L_{jk}$ that L_{ik} is $L_{i,j-1}$, meaning the value calculated in the previous `j` loop iteration. Again, here we are forced to be sequential. Both explanations are illustrated in Figure 4.1.

So, our goal now is to transform the algorithm to a different type of algorithm that **can** be parallelized as good as possible. For this, we'll use the Forward and Backward algorithms introduced in Algorithm 2.3 and Algorithm 2.4 as inspiration. Our key, just like in the Forward

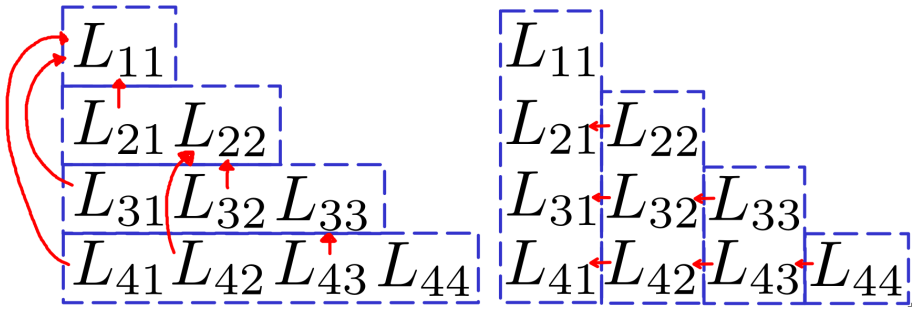


Figure 4.1: Left: If we want to parallelize the i loop, then the i iterations should depend on each other. This is displayed with the blue blocks being sort of "barriers." However, as seen by the text, there are places in the algorithm where an i iteration does require the result of another, indicated by the red arrows. Right: The same situation, but this time to parallelize the j loop, which again isn't possible due to dependencies shown by the red arrows.

and Backward Substitution algorithms, will be to, rather than having a single iteration setting an entry in the L matrix, set what is possible in a single iteration in the L matrix and then, using the value, update the rest of the matrix. We can achieve this by noticing that Algorithm 2.1 sums values $L_{ik}L_{jk}$ and then subtracts it from the respective entry. Here, we could possibly take each individual $L_{ik}L_{jk}$ and subtract it from the entire matrix as they come. This can be achieved by holding to a constant k. Then, $L_{ik}L_{jk}$ are all from the same column, meaning if we can calculate a column without $L_{ik}L_{jk}$, then we have a new algorithm. By looking at algorithm 2.1, we notice that on each i for $j = 1$, so the first column, we don't use the k loop. Thus, we don't require $L_{ik}L_{jk}$; we just do a square root if $i = j$, or else a division. In the next column, $j = 2$, we use the k loop but only with $k < j$, so we just use the previous columns. Using our findings, we construct the algorithm described in 4.1. The algorithm, in essence, inverts the sum calculation s in Algorithm 2.1 and

Algorithm 4.1 Perform the Cholesky Decomposition on A using Submatrix Cholesky

Require: $A \in \mathbb{R}^{n\times n}$ positive definite and symmetric
Ensure: $L \in \mathbb{R}^{n\times n}$ lower triangular matrix with $A = LL^T$

```
for k ∈ (1, ..., n) do                          // This loop can now be parallelized
    L_kk ← √L_kk
    for i ∈ (k + 1, ..., n) do
        L_ik = L_ik / L_kk
    end for
    for j ∈ (k + 1, ..., n) do                  // This loop can now also be parallelized
        for i ∈ (j, ..., n) do
            L_ij ← L_ij − L_ik L_jk
        end for
    end for
end for
```

rather subtracts $L_{ik}L_{jk}$, as both values are known, from the rest of the matrix. It works by now using a different k loop, which defines at what column we are currently. On a column, we first

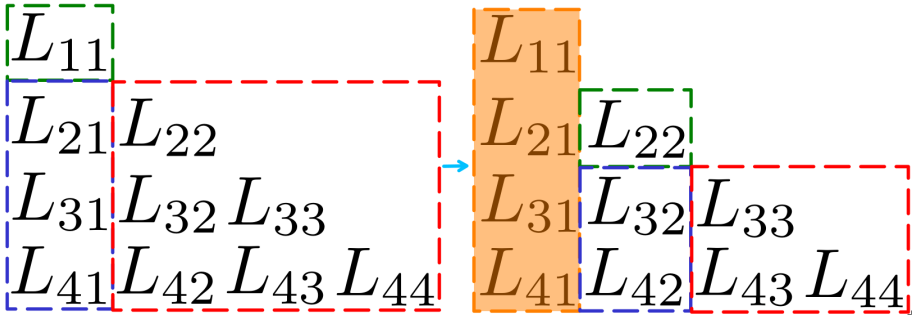


Figure 4.2: Example of running a single iteration of the 4.1 algorithm. Green indicates what is going to be square rooted, blue indicates which column is to be divided, and red is what submatrix will get updated. Green and blue together represent the *update column* routine, while red forms the *update submatrix* routine. The orange area marks the columns that are done. Left to right shows that effectively each iteration has the problem again, making the loop recursive in nature.

square root the diagonal element and then divide it with each column entry under the diagonal, which we'll call the *update column* routine. Then, we subtract $L_{ik}L_{jk}$ from each element to the right of the column, which we'll call the *update submatrix* routine. A visualization of the algorithm is illustrated in figure 4.2.

Using our new algorithm 4.1, we have now loops i and ji with iterations independent of each other. However, the k loop isn't parallelizable since otherwise we'd have *update column* and *update submatrix* run in the wrong order. It must run in the order illustrated in figure 4.2. Parallelization will come in the next chapter.

4.2 Saving Memory

Due to the fact that A is a symmetrical matrix and both of our Algorithms 2.1 and 4.1 use only the lower triangle matrix, we can save memory by only storing the relevant part of the matrix. However, this will make indexing the matrix for the algorithm different, which needs to be addressed. As the square structure is now broken, we'll now use a linear address space where each row is stored one after the other. We'll need a function to calculate the allocation size based on a given matrix size N and a function to calculate the index to the linear address space based on a given i and j. The allocation size can be simply determined by recognizing that the amount of elements of a triangle is a triangular number and can be calculated with the formula 4.1.

$$\text{alloc}(N) = \frac{N(N+1)}{2} \tag{4.1}$$

For indexing the address space, we can see, orienting on figure 4.3, that the number of elements over the indexed element is again a triangle number and can be calculated with $\frac{i(i+1)}{2}$. Then we also additionally offset in the address space by simply adding j on top. This can be achieved with the formula 4.2.

$$\text{index}(i,j) = \frac{i(i+1)}{2} + j \tag{4.2}$$

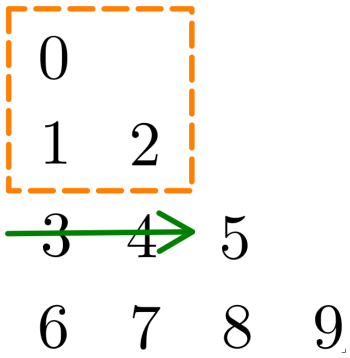


Figure 4.3: Example of calculating the index using our row-based method. Orange is the triangular block before, which is of size i, and the green arrow is the offset from the left, which is of size j. Note: i marks the row and j marks the column.

Later, we'll also use a number of other indexing strategies based on different memory layouts. These will be introduced as needed in the next chapter.

5 Parallel Cholesky Decomposition using the GPU

In this chapter, we'll sometimes introduce a so-called microbenchmark, which we'll use to validate if an optimization does, in fact, do something. These benchmarks, however, are only performed on one dataset and not for long. They serve only to point out if an optimization technique could potentially work or something is bad for runtime. They are performed on a Quadro RTX 4000 NVIDIA GPU and Intel(R) Core(TM) i9-10980XE CPU. For each microbenchmark, the SYCL queue is `q`, which will be in order, and the data will always be in `gpuA`, which has allocated enough memory depending on the task at hand. The microbenchmark will be split by a comment indicating which two code snippets we are comparing. We'll be focusing mostly on optimizing for the GPU, as our top goal is parallelization, for which a GPU is very good. However, later in the evaluation chapter, we'll also test out our best-performing GPU implementation on the CPU.

Sections in this chapter contain the algorithm name that will be designed with an index in brackets. The index will then be used in the evaluation benchmarks to reference the algorithm used.

At the end of each implementation, we will include a process table for the reader to see how an optimization fares against the previously introduced implementations.

5.1 Naive Cholesky (01)

After doing the heavy work in the previous chapter, we can now start on implementing a parallel algorithm. Simply directly using Algorithm 4.1, we get the listing 5.1. The implementation works in the following way. Just like in algorithm 4.1, we have the k loop. As stated before, this is a necessary dependency. For our *update column* routine, we implement it as a `single_task` one-to-one as in the algorithm. We choose `single_task` to avoid calculating the square root multiple times. For our *update submatrix* routine, we this time use a `parallel_for` but opt to use an internal for loop since the submatrix we update is triangular, so this way we avoid uselessly iterating over unused indexes. In SYCL, `parallel_for` has the limit of forcing the indexable range to be from zero to a certain number, which is a problem for us as our algorithm 4.1 specifies the loop range to be from $k + 1$ to N. We alleviate the problem by specifying in the range of the `parallel_for` how much we want to iterate, which is just the difference $N - (k + 1)$. To have the correct j again, we can simply add $k + 1$ onto the index variable given by the `parallel_for`. Also, since a SYCL `parallel_for` doesn't accept a zero range, we also have to safeguard the *update submatrix* routine with $k < N - 1$, since else we get $k \geq N - 1 \Leftrightarrow 0 \geq N - 1 - k = N - (k + 1)$, which is exactly what we want to avoid.

Listing 5.1 Naive Cholesky (01) using rowbased indexing

```
// Initialize a SYCL queue q and allocate enough triangle memory into gpuA and load in matrix
A in a rowbased data format
sycl::event e;
for (size_t k = 0; k < N; ++k) {
  e = q.single_task(e, [=] {
    gpuA[index(k, k)] = sycl::sqrt(gpuA[index(k, k)]);
    for (size_t i = k + 1; i < N; ++i) {
      gpuA[index(i, k)] /= gpuA[index(k, k)];
    }
  });
  if (k < N - 1) {
    e = q.parallel_for(sycl::range<1>(N - (k + 1)), e, [=](sycl::id<1> idx) {
      const auto j = idx[0] + k + 1;
      for (size_t i = j; i < N; ++i) {
        gpuA[index(i, j)] -= gpuA[index(i, k)] * gpuA[index(j, k)];
      }
    });
  }
}
```

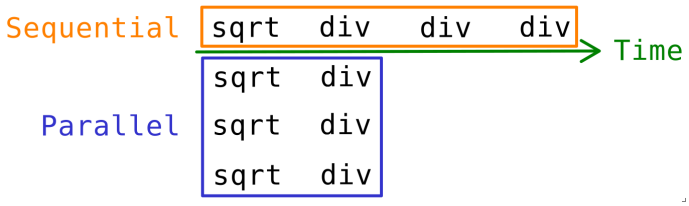


Figure 5.1: A showcase for seeing why calculating a square root multiple isn't times always bad. As we see here, while the sequential version does, in total, perform fewer calculations, it still takes longer, as the parallel version can make the calculations in parallel.

This implementation works, but it has multiple problems. The biggest one is that it will be very slow due to the sequential for loops. A GPU, as stated in section 2.2, specializes in high throughput using its parallel capabilities. By forcing the GPU to be sequential, we massively hinder its throughput as it doesn't have good latency. So, avoiding calculating the square root for *update column* has actually hindered our performance, since it all happens in parallel. First calculating the square root, then updating each element will be slower than in parallel calculating a square root and then updating a single value. This concept is illustrated in figure 5.1. Similarly, this concept can also be applied to our *update submatrix* routine.

01
$5600s \pm 0.2$

Table 5.1: Progress overview up to 01 with $N = 60000$ and single-precision.

Listing 5.2 Simple Cholesky (02) using rowbased indexing

```
// Initialize a SYCL queue q and allocate enough triangle memory into gpuA and load in matrix
A in a rowbased data format
sycl::event e;
for (size_t _k = 1; _k < N; ++_k) {
  const auto k = _k - 1;
  e = q.parallel_for(sycl::range<1>(N - _k), e, [=](sycl::id<1> idx) {
    const auto i = idx[0] + _k;
    gpuA[index(i, k)] /= sycl::sqrt(gpuA[index(k, k)]);
  });
  e = q.parallel_for(sycl::range<2>(N - _k, N - _k), e, [=](sycl::id<2> idx) {
    const auto i = idx[0] + _k;
    const auto j = idx[1] + _k;
    if (i >= j) {
      gpuA[index(i, j)] -= gpuA[index(i, k)] * gpuA[index(j, k)];
    }
  });
}
e = q.parallel_for(sycl::range<1>(N), e, [=](sycl::id<1> idx) {
  const auto k = idx[0];
  gpuA[index(k, k)] = sycl::sqrt(gpuA[index(k, k)]);
});
```

5.2 Simple Cholesky (02)

Using our findings from before, we can implement a implementation in listing 5.2. The implementation addresses the previous issues by now making everything parallel. We split our *update column* routine into two parts. In the loop, the first `parallel_for` is the first part, and it does most of *update column* but **not** writing the square results to the diagonal. This must be done as we don't know which `parallel_for` iteration will be the last; thus, we need to pause the diagonal update for later, which is after the loop in the last `parallel_for`. Here, we simply do the square root again but now to update the diagonal, which can be done in one parallel call. Our *update submatrix* now doesn't have any loops but is now a two-dimensional `parallel_for` with a safeguard inside to only update values that are actually in the lower triangle submatrix, since it must have $i \geq j$.

This implementation should use the GPU in a better way. Which leads us to a great introduction to *memory coalescing*. As mentioned in the background, a GPU works with groups of work-items called sub-groups that execute code at once. Due to global memory access being expensive, the GPU tries to minimize the amount of memory accesses by optimizing for sequential memory access within a sub-group. This means that with a sub-group consisting of 32 work-items and each work-item accessing memory from within a 32 range of all other data accesses of the sub-group, the GPU only has to call a single $32 \cdot 4 = 128$ Bytes load from global memory if we are using 4 Byte values. A further example is found in figure 5.2. We already have this optimization for the *update submatrix* routine, since as we are row-based, then index(i, j) and index$(i, j + 1)$ are subsequent data elements, and subsequent work-items, which are then in a sub-group, are separated by j. This means we consecutively access global memory and thus use memory coalescing. However,

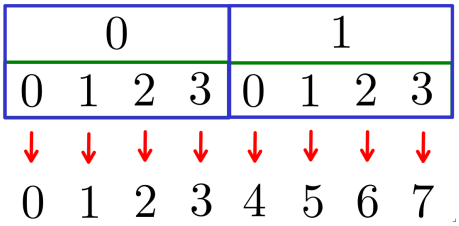


Figure 5.2: An example of memory coalescing with a sub-group size of 8, with a two-dimensional work-items range, with green being the lower and blue being the higher. Each blue group has 4 greens. This shows how the work-items must be ordered in multiple dimensions to still achieve memory coalescing.

update column does not, since here we go down the column where the data elements are not stored consecutively. It isn't a big deal, as *update column* only has the time complexity of $O(n)$ compared to $O(n^2)$ for the *update submatrix* routine, but it's still a possibility to explore further.

Until now, we've only been using SYCL's simple task types like `single_task` and `parallel_for`, while omitting the more advanced task types that give us access to local memory. Local memory can further reduce global memory access costs by intermediately storing values for multiple work-items to use. Since our tasks until now are arithmetically simple, we most likely need to improve memory access.

01	02
$5600s \pm 0.2$	$643s \pm 0.5$

Table 5.2: Progress overview up to 02 with $N = 60000$ and single-precision.

5.3 No Square Root Cholesky (03)

Before continuing on with implementing the usage of local memory, we will explore here in short a slightly different approach to both *update column* and *update submatrix* while avoiding the square root. Looking at our previous implementation, we see that we first divide by the square root of the diagonal, which is *update column*, and then multiply two values of the column together in *update submatrix*. This is represented with $\frac{a}{\sqrt{d}} \cdot \frac{b}{\sqrt{d}}$, which can be simplified to $\frac{a \cdot b}{d}$. This approach eliminates a square root and even removes a task from the loop, as *update column* doesn't need to be called anymore before *update submatrix*, since it doesn't require the column values to be divided beforehand. This change results in the listing 5.3. In the implementation, we first invoke, in contrast to 5.2, the *update submatrix* routine, but now also including the division with the diagonal. After the loop, we perform the *update column* routine by first updating the diagonal as before, but now also in a separate task updating **all** the columns at once, making the *update column* routine now have the time complexity of $O(n^2)$.

This could potentially lead to a faster runtime, as the implementation 5.3 only calls half of the tasks of listing 5.2. The potential speedup can be observed in the microbenchmark of listing 5.4. This results in one task running in $0.00701228s$ and many sequential tasks in $0.0856572s$. The result is

Listing 5.3 Simple Cholesky (02) using rowbased indexing

```
// Initialize a SYCL queue q and allocate enough triangle memory into gpuA and load in matrix
A in a rowbased data format
sycl::event e;
for (size_t _k = 1; _k < N; ++_k) {
  e = q.parallel_for(sycl::range<2>(N - _k, N - _k), e, [=](sycl::id<2> idx) {
    const auto i = idx[0] + _k;
    const auto j = idx[1] + _k;
    if (i >= j) {
      gpuA[index(i, j)] -= gpuA[index(i, k)] * gpuA[index(j, k)] / gpuA[index(k, k)];
    }
  });
}
// Square root the diagonal
q.parallel_for(sycl::range<2>(N, N), [=](sycl::id<2> idx) {
  const auto i = idx[0];
  const auto j = idx[1];
  if (i > j) {
    gpuA[row_idx(i, j)] /= gpuA[row_idx(j, j)];
  }
});
```

Listing 5.4 Microbench: Many non sequential tasks vs. one task

```
q.parallel_for(sycl::range<2>(N, N), [=](sycl::id<2> idx) {
  const auto i = idx[0];
  const auto j = idx[1];
  if (i > j) {
    gpuA[index(i, j)] /= 2;
  }
});
// --- Here we'll use not an inorder queue to not have a sequential for loop
for (size_t i = 0; i < N; ++i) {
  q.parallel_for(sycl::range<1>(i + 1), [=](sycl::id<1> idx) {
    const auto j = idx[0];
    gpuA[index(i, j)] /= 2;
  });
}
```

obvious, but it is of interest to see how much of an improvement it makes. However, short tests of the implementation indicate that the optimization may fall short, which is why we'll benchmark the method separately in the evaluation and discuss further what has happened in the evaluation. There are also numerical concerns, as while we do eliminate potential rounding errors due to the square root, we also have to consider the multiplication, which could result in high values possibly leading to an infinity value. These scenarios are data-dependent, and one solution isn't much worse than the other. So, we've decided to use the solution that benefits us the most for runtime.

01	02	03
$5600s \pm 0.2$	$643s \pm 0.5$	$660s \pm 0.1$

Table 5.3: Progress overview up to 03 with $N = 60000$ and single-precision.

5.4 Local Memory Cholesky (04)

Now we'll finally use local memory. But first, we'll need to discuss how to effectively use it here. For *update column*, we can simply instruct one work-item to store the diagonal element in local memory, and all the work-items in a sub-group can then use it to update their individual column entries. This is what we'll do. For *update submatrix*, we know that we use $L_{ik}L_{jk}$ to update the individual entries of the submatrix. Here, we can see that we can either store L_{ik} for multiple j in local memory, or the other way around, or both for multiple i and j. For implementing 04, it was chosen to store L_{ik} values, as it was the simplest at the time and due to the fact that we'd like to accommodate the *update column* routine with better memory coalescing for better performance by choosing a column-based indexing method this time. Since we don't want to just store one L_{ik} into local memory but rather multiples of 32, which is the sub-group size on the GPU, to achieve memory coalescing, we will also need to consider padding in case we are out-of-bounce from our lower triangular matrix.

The purpose of padding is to avoid if statements that would branch our code, meaning it allocates memory locations that serve only to prevent memory access violations. This is preferable to branching, as due to the GPU working in sub-groups, we'd have situations where some work-items go into one branch while others go into another. The GPU handles this situation by running the sub-group partially, meaning only the work-items together in one branch are executed while the other work-items still in the same sub-group have to wait until the GPU starts partially executing them. So, if we are guarding global memory so that only some work-items enter the `if` and load, then the other work-items also have to wait, but they do nothing in the meantime. If there is another branch for the waiting work-items, for example, loading in a default value, then they can only execute it after the global memory load for the other work-items has completed. So, in sum, it's as if each work-item had to load from global memory and **also** load the default value. The is called *branch divergence*. Additionally, there is the problem of branches, which in general hinder the execution pipeline and optimization. From this, we evaluate that allocating a little more memory for padding is preferable for better runtime.

For our new column-based indexing, we'll have again a linear address space, but now rather than storing the triangle matrix row by row, it is stored column by column. Additionally, padding needs to be inserted. We define the size of our local memory as `WIDTH`, meaning each time we'll store

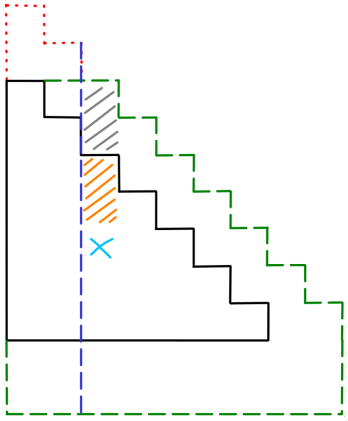


Figure 5.3: Illustration of our column-based indexing method on a matrix $N = 7$ with padding $\texttt{WIDTH} = 3$. The top red area is the additionally allocated memory to complete the triangular form for easier calculation. Green indicates the padding that is used for the column-based indexing method. The blue line offsets our search for the index (which is the light blue cross) from the left, which is why we subtract the two triangle numbers. The gray entries are then the offset from the top down along the padding, which is $\texttt{WIDTH} - 1$, and the orange entries are the offset down from the diagonal, which is $i - j$.

a WIDTH amount of L_{ik} into local memory. So, for our lower triangular matrix in memory, we'll need to pad $\texttt{WIDTH} - 1$ after each diagonal. We'll also use the constant as the amount of work-items per work-group, meaning how many work-items will use the loaded L_{ik} and then their own L_{jk} to calculate the entry. Again, this requires padding, as we don't want to write into an illegal location. Using our requirements for padding, which is also illustrated in figure 5.3, we can now construct the allocation size function and index function.

We start with the easier allocation size function. To have an easier understanding of what is going on, we'd recommend following along with the example area in figure 5.3. We can simplify a lot of calculations, especially for the index function, by also again adding a little padding to the top, completing the allocated area with padding into a big lower triangle matrix and a rectangular matrix. The size of the triangle is then calculated with $\frac{(N + \texttt{WIDTH} - 1)(N + \texttt{WIDTH})}{2}$ and the rectangle with $(N + \texttt{WIDTH} - 1)(\texttt{WIDTH} - 1)$, which combined results in our function 5.2. For the index function, we first need to get the size of the amount of entries to the left of what we are indexing. This can be achieved by subtracting two triangle numbers, so $\frac{N(N+1)}{2} - \frac{(N-j)(N-j+1)}{2}$, which simplifies to

$$\begin{aligned}
\frac{N(N+1)}{2} - \frac{(N-j)(N-j+1)}{2} &= \frac{N(N+1) - (N-j)(N-j+1)}{2} \\
&= \frac{N^2 + N - (N-j)^2 - (N-j)}{2} \\
&= \frac{N^2 - (N-j)^2 + j}{2} \\
&= \frac{N^2 - N^2 + 2Nj - j^2 + j}{2} = \frac{j(2N - j + 1)}{2}
\end{aligned}$$

Listing 5.5 Local Memory Cholesky (04) *update column* routine using column-based indexing

```
// Initialize a SYCL queue q and allocate enough triangle memory into gpuA and load in matrix
A in a columnbased data format with padding
// Iterate from 1 to N using _k
e = q.submit([&](sycl::handler &cgh) {
  auto localCoefk = sycl::local_accessor<real_type, 1>(sycl::range(1), cgh);
  cgh.parallel_for(
    sycl::nd_range{ sycl::range(ceil_div(N - _k, WIDTH) * WIDTH), sycl::range(WIDTH) },
    [=](sycl::nd_item<1> idx) {
    const auto groupi = idx.get_group(0);
    const auto locali = idx.get_local_id(0);
    const auto offseti = _k + groupi * WIDTH;
    if (locali == 0) localCoefk[0] = gpuA[index(k, k)];
    sycl::group_barrier(idx.get_group());
    gpuA[index(offseti + locali, k)] /= sycl::sqrt(localCoefk[0]);
  });
});
// Call update submatrix
// Update diagonal
```

So we get the identity

$$\frac{N(N+1)}{2} - \frac{(N-j)(N-j+1)}{2} = \frac{j(2N-j+1)}{2} \tag{5.1}$$

But for us, the side of the triangle of our allocated area is a little larger than N, which is $2(\texttt{WIDTH} - 1) + N$, so we'll substitute that in to accommodate. Now, with the left part done, we need the offset of how much down we must go. This can be calculated by combining the padding elements' size above the diagonal, which is $\texttt{WIDTH} - 1$, and the i value offset back based on the column we are on, so $i - j$. Everything combined with $s = \texttt{WIDTH} - 1$ and $t = 2s + N$ results in the formula at 5.3.

$$\text{alloc}(N) = \frac{(N + \texttt{WIDTH} - 1)(N + \texttt{WIDTH})}{2} + (N + \texttt{WIDTH} - 1)(\texttt{WIDTH} - 1) \tag{5.2}$$

$$\text{index}(i, j) = \frac{j(2t - j + 1)}{2} + i - j + s \tag{5.3}$$

With our special column-based indexing with padding, we can now construct implementation 04, starting with the *update column* routine, which is implemented as shown in listing 5.5. As we've specified before, local memory has only one entry. Keeping in sync with our *update submatrix* routine, our *update column* routine also uses `WIDTH` as the amount of work-items per work-group. This must be done; otherwise, we could overstep our allocated limits. For the global range, since we want lines aligned with the work-group size, which is `WIDTH`, we first need to get how many work-groups can fit in our range, which can be calculated using $\lceil (N - \_k) \div \texttt{WIDTH} \rceil$. In code, dividing and then ceiling the output is done through `ceil_div(a, b)` with a possible implementation of `a / b + (a % b != 0)`. Then, multiplying the answer with our work-group size, we get the global range.

Listing 5.6 Local Memory Cholesky (04) *update submatrix* routine using column-based indexing

```
// Initialize a SYCL queue q and allocate enough triangle memory into gpuA and load in matrix
A in a columnbased data format with padding
// Iterate from 1 to N using _k
// Call update column
e = q.submit([&](sycl::handler &cgh) {
  auto localCoefj = sycl::local_accessor<real_type, 1>(sycl::range(WIDTH), cgh);
  cgh.parallel_for(
    sycl::nd_range{
      sycl::range(ceil_div(N - _k, WIDTH) * WIDTH, ceil_div(N - _k, WIDTH)),
      sycl::range(WIDTH, 1)
    }, [=](sycl::nd_item<2> idx) {
      const auto groupi = idx.get_group(0);
      const auto groupj = idx.get_group(1);
      const auto locali = idx.get_local_id(0);
      const auto offseti = _k + groupi * WIDTH;
      const auto offsetj = _k + groupj * WIDTH;
      if (groupi < groupj) return;
      const auto coefi = gpuA[index(offseti + locali, k)];
      localCoefj[locali] = gpuA[index(offsetj + locali, k)];
      sycl::group_barrier(idx.get_group());
      for (size_t u = 0; u < WIDTH; ++u) {
        gpuA[index(offseti + locali, offsetj + u)] -= coefi * localCoefj[u];
        sycl::group_barrier(idx.get_group());
      }
    });
});
// Update diagonal
```

Inside the task itself, it is now important to correctly calculate the index or address of the values we are targeting. Here, this is done through `offseti`, which specifies the start of the data belonging to the work-group. This is calculated using `groupi` $\cdot$ `WIDTH`, which simply offsets us on the global range, and by adding `_k`, we account for the fact that the `parallel_for` starts at 0. This was described in more detail in section 5.1. We'll use this method of address in our next implementations as well. Indeed, to avoid potential undefined behavior due to data races between work-groups, the best practice is to cut up the whole global range, which is one-to-one with the addressed data, into blocks and assign each block to a work-group. In this case, we cut up the range into $\lceil (N - \_k) \div \texttt{WIDTH} \rceil$ pieces, as specified by the global range, with blocks of size `WIDTH`. But note that the blocks in this case are one-dimensional. Returning to the implementation, we instruct our first work-item to load in the diagonal into local memory, wait for all work-items until it's done, and then do the *update column* routine as we've seen before.

The *update submatrix* routine is a little more complicated due to being two-dimensional. It's displayed in listing 5.6. In the implementation, just as with before in listing 5.5, we specify $\lceil (N - \_k) \div \texttt{WIDTH} \rceil$ in the global range, but this time as its height, which gets multiplied as before with `WIDTH`. Due to the fact we are now indexing column-based, we need to place the work-items on the height axis. For the width (not to be confused with `WIDTH`), we now also use $\lceil (N - \_k) \div \texttt{WIDTH} \rceil$ **but** without multiplying by `WIDTH`. We do this since each of our work-groups stores multiple L_{ik}

values of differing i, and the work-items will then load their respective L_{jk} into registers, use the L_{ik} value in local memory, do their calculation, and then **continue** to the next column, now going to use the next L_{ik} value in local memory without leaving the task. This way, a work-item does multiple entries on the width axis; thus, we'd have to divide the global range width by the amount of columns the work items will go along. However, that is just the size of local memory, which is `WIDTH`, so this simply results in not multiplying by `WIDTH`.

In the task itself, we start, as we've said earlier, with listing 5.5, where we calculate the work-group offset in global memory with `offseti` and `offsetj`, now pointing to the data for the work-group. Then, we place a safeguard in the form of an `if` to disallow work-groups that are not in the lower triangle region. This is enough, as we know the submatrix that is being iterated over is itself a lower triangle matrix. So, the first work-group will have its top left entry as the first diagonal element and the next work-groups will either have nothing (so they are out of the triangle), be inside the triangle (where they will be completely full), or again be on the diagonal (where the top left entry will have the first diagonal element of the work-group). The latter two cases only happen if `groupi` $\geq$ `groupj`. This alignment only works if the work-groups' data is square-shaped, as if it were rectangular, then it could be the case that the top left element is empty. This would break the `if` guard, since by comparing the group values, we only check if the top left element is inside our matrix; yet here, there are cases where there are values just not on the top left.

Next, each work-item loads its respective L_{jk} into its own register, since no other work-item also requires it, and they load L_{ik} into local memory, each writing to its respective spot in local memory. After the work-group has waited for all work-items to finish reading and placing their values, then the calculation starts. Each work-item along the column takes its register-stored L_{jk} and the together-stored L_{ik} in local memory, multiplies them, and then subtracts from its respective entry in global memory, which in fact requires another load. However, since we go through the matrix once, we don't really need to do anything special here but keep ourselves to memory coalescence. We also have an interesting wait instruction inside the loop. We don't actually need the wait, as the work-items are not dependent on other work-items writing to local memory; however, short tests revealed a surprising speedup in runtime by including the wait. We are not entirely sure why this happens. Normally, it is preferable to avoid using unnecessary barriers, but it seems that this is not the case here. Our best guess is that it occurs due to the need for something to align itself, although we are uncertain about what that might be. Nevertheless, since this approach results in better runtime performance, we have decided to keep it here. Figure 5.4 visualizes the algorithm again.

The implementation, combined with listings 5.5 and 5.6, should ease the usage of global memory due to the reuse of values using local memory. However, in retrospect, we could have separated the constant `WIDTH` into two parts: the first being the amount of work-items in a work-group, which would dictate how far down we go on a column in the *update submatrix* routine, but also how many entries a work-group handles in the *update column* routine. The second constant would then be the actual size of the local memory, which would dictate how far the work-group would be able to go right in the *update submatrix* routine. We omitted to do that since it would increase complexity, especially with the index calculation and `if` guard, but it would grant us more fine control over how the algorithm behaves. However, a problem is our big use of padding. While on sufficiently large matrices it should be outgrown, as it only grows linearly compared to quadratically with the matrix, it still results in wasted memory. Preferentially, we would like an implementation that does use

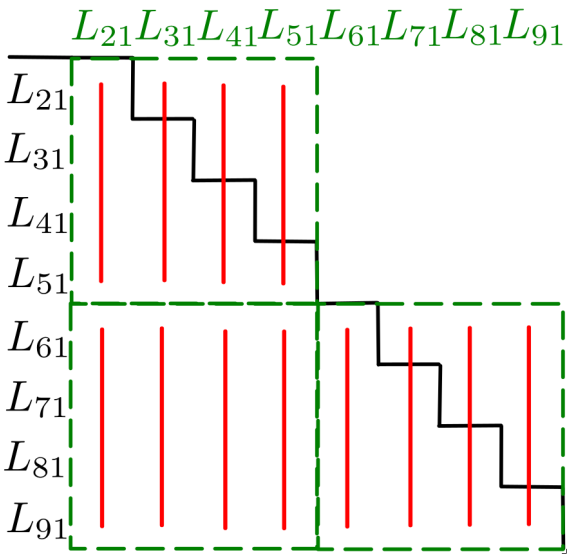


Figure 5.4: Example execution of listing 5.6 implementation. The left values are what are loaded by the registers of each work-item, while the green values above are the values loaded into local memory. The green blocks are then the regions executed by work-groups. So, not the entire green value line is loaded into local memory at once, but only what is directly above the green block. The red lines indicate the work-item placement of what is executed in a single for loop iteration in the 5.6 implementation.

padding but more moderately, to allow even just slightly larger matrices to fit in the global memory of the GPU. The most glaring issue, however, is the big k loop on the outside, as it goes through the matrix multiple times and spawns a lot of tasks.

01	02	03	04
$5600s \pm 0.2$	$643s \pm 0.5$	$660s \pm 0.1$	$454s \pm 0.01$

Table 5.4: Progress overview up to 04 with $N = 60000$ and single-precision.

5.5 Simple Blocked Cholesky (05)

Looking at how we've progressed so far, it seems the best way forward is to try and do a lot at once. Take our previous implementation step from 02 to 04. Short tests on a moderately sized matrix conclude that it's indeed faster (more in the evaluation). Until now, on our Algorithm 4.1, we've managed to use local memory to sort of create "blocks" that get executed by work-groups. While there are possibly ideas to further optimize *update column* and *update submatrix*, they wouldn't be as effective as optimizing the big loop, which is the k loop, since if we could manage to somehow use a blocked method for it, we could in fact save a lot of iterations, making it an optimization definitely worthwhile. Which is why we'll now leave our scalar Algorithm 4.1 behind and focus on the blocked Cholesky Decomposition Algorithm 2.5.

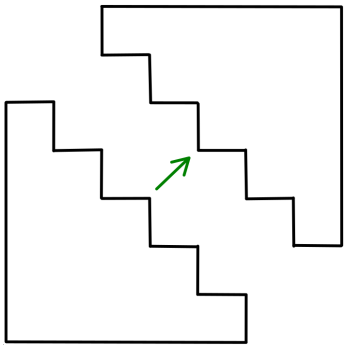


Figure 5.5: We will now transpose our view on the matrix. So the first column becomes the top row and the top diagonal element becomes the left most diagonal element and so on.

Comparing both algorithms 4.1 and 2.5, we notice a lot of similarities, since both have the *update column* and *update submatrix* routine. But while algorithm 4.1 only does a square root and divisions for *update column*, the blocked algorithm 2.5 calculates a scalar Cholesky Decomposition and a matrix multiplication with an inverse (for which we do a forward substitution). For *update submatrix*, our old algorithm does a sort of vector-matrix multiplication with a transpose, as can be seen in figure 5.4, since how the values L_{ik} and L_{jk} are ordered is very similar to how to order a written matrix multiplication. And for our blocked Cholesky Decomposition, we also do a matrix multiplication with a transpose, but now with matrices, not vectors. So to implement are:

1. A standard Cholesky Decomposition on a block.
2. A forward substitution given a block as our matrix and another block with vectors as columns.
3. A matrix multiplication of two blocks, with one of them being transposed.

For the block size, we'll define `WIDTH`. Also, now implementing the blocked Cholesky Decomposition, we'll transpose our view of the L matrix in memory, so that we remain consistent with [CJS+13] and with the PLSSVM view of the matrix. Every time we reference the lower triangular matrix, we now mean the upper triangular matrix. Figure 5.5 shows our change.

5.5.1 Indexing

Since we are now using a completely new algorithm, we have to yet again define new indexing and allocation size functions. But this time it will be our final one, as it will also be compatible with PLSSVM. Due to our blocked approach now, we'll need to set up the data to be in blocks. This way, if we call *update column* or *update submatrix*, we'd be sure that whatever we do with the block in a work-group, everything is an accessible value and shouldn't be an unallocated location. This is important since, for our blocked algorithms, we are going to assign `WIDTH` $\times$ `WIDTH` amount of work-items per work-group, intending for each work-item to have its own spot when accessing a block. That way, if we want to, for example, copy a block, a single copy statement is all that is needed.

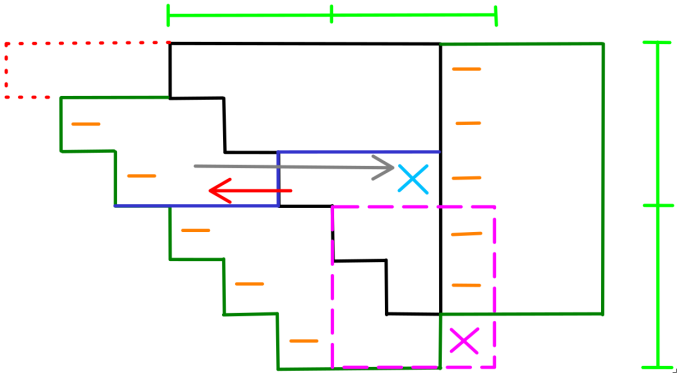


Figure 5.6: We will break down the image into what each of the colored components does. This is a matrix with $N = 5$ and `WIDTH` $= 3$. Black is the matrix itself, storing our values. Green represents two views to look at the padding from. We use both of them since we will have the issue of needing to reuse memory locations. Orange marks the memory locations on the right that will be accessed by the block in that location, which is outlined using light green. On the left green padding, the orange marks are the memory locations from the right that are accessed by the blocks. Ultimately, these do not influence the index calculation but are there to visualize the idea of looping the padding around to create the block form seen in Figure 5.7. Light blue is the index we want to calculate. Blue is the offset from the incomplete triangle made out of black, left green, and dotted red above light blue. The red arrow is the offset to complete left (using i), so that gray, the offset of j, lands on the right spot. Magenta marks the problem block at the end that doesn't have enough padding elements.

Again, as with both of our indexing methods before, we'll use a linear address space, so our allocation size function just needs to return the total size of what we are allocating. For parity with PLSSVM, this will be Equation 5.4.

$$\text{alloc}(N) = \frac{N(N+1)}{2} + N \cdot \text{WIDTH} \tag{5.4}$$

It creates a triangle but also pads each row with the padding `WIDTH`. For indexing, we use the incomplete triangle above of what we are indexing to shift on i. We can use Equation 5.1 for simplification. We also get the offset from the left j minus how far down we are i to correctly align ourselves back due to the shift of the incomplete triangle we did before, which takes us straight to the diagonal element. These result in together the indexing function at Equation 5.5.

$$\text{index}(i, j) = \frac{i(2(\texttt{WIDTH} + N) - i + 1)}{2} - i + j \tag{5.5}$$

Figure 5.6 shows a detailed example of the indexing calculation and how the data is viewed. However, we come to the problem that our last block could potentially be missing padding elements. But this can be remedied by closing off the end with an `if` statement. The `if` checks at the start if $i \geq N$, and if true, then sets `i = N` and `j = N - 1`, and the calculation then goes along normally. Using these coordinates, we always have a valid padding element. After applying the indexing, we now get a blocked view like in Figure 5.7. Just what we have been looking for.

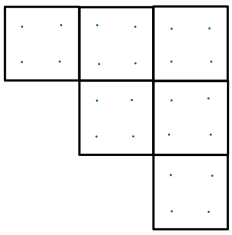


Figure 5.7: The block form we want to have at the end.

Listing 5.7 Cholesky Decomposition on a single block

```
template <typename real_type, size_t WIDTH>
void cholesky_decomposition(sycl::group<2> g,
                            sycl::local_accessor<real_type, 2> cholblk) {
  const auto locali = g.get_local_id(0);
  const auto localj = g.get_local_id(1);
  for (size_t k = 0; k < WIDTH - 1; ++k) {
    if (locali > k) {
      cholblk[locali][localj] -= cholblk[k][locali] * cholblk[k][localj] / cholblk[k][k];
    }
    sycl::group_barrier(g);
  }
  const auto diag = sycl::sqrt(cholblk[locali][locali]);
  sycl::group_barrier(g);
  cholblk[locali][localj] = (locali == localj) * diag
                          + (locali != localj) * cholblk[locali][localj] / diag;
}
```

5.5.2 Cholesky Decomposition

We'll again need to implement the Cholesky Decomposition; however, this time we have some special specifications. Most notably, we now have a limited size matrix, since it has to fit into a single block, and our matrix is now transposed. So, for example, the first *update column* refers to updating the first diagonal and then updating the row to the right of it. As the Cholesky Decomposition will be used multiple times, it's best to create a function out of it. As input, we will want the diagonal block to be loaded into local memory already. The function will then only operate within the given local memory without affecting other local memory segments or the global memory and will be handled by a work-group. Our implementation is listed in 5.7. The implementation works by implementing the k loop that runs *update column* and *update submatrix* like before, but here we implement the no square root variant of the Cholesky Decomposition, since it results in fewer barriers to synchronize all the work-items. This happens because, otherwise, with an *update column* in the loop, we would need a barrier between *update column* and *update submatrix*, since we first have to update the column **then** we can update the submatrix, and then another barrier, as the matrix needs to be done updating the submatrix before calling *update column*. Without the *update column*, so implementing the no square root Cholesky, we just need to make sure the submatrix updates don't overlap, hence the single barrier in the k loop.

Again, to clarify, we have a transposed matrix, so our first ”column” is, in this case, the top row. This works to our favor, as here `localj` work-items are adjacent to each other, and if we set the work-group size also to two dimensions with a length of `WIDTH`, then each local memory location has a work-item that manages it. So the k loop in the implementation just needs to filter if the work-item is within the submatrix. If it is within, then we do the same `update submatrix` as with implementation 03; however, this time the matrix elements are obtained from local memory. The k loop then iterates until all submatrices are done.

After the loop, similarly to 03, we would want to update all the diagonals with the square root. However, this is hard here, since we have a `WIDTH` $\times$ `WIDTH` amount of work-items, but we require just a `WIDTH` amount of work-items to accomplish the update. But also, similarly to 03, we would want to perform the *update column* routine. We will combine both tasks and do them at once, since we have enough work-items. Since the diagonal update and *update column* need the diagonal entries, we first collect those and then wait, because afterwards we will modify everything. The last statement of the implementation does both of the tasks. If the work-item is on the diagonal, then it sets the diagonal to its square root. Else, we take the square-rooted diagonal and use it to divide the ”column” values.

Note: In the implementation, we do use branching; however, in this case, it isn't as bad since, as stated before, `localj` are adjacent to each other. As the dimension size is `WIDTH`, which is the block size, and if we set `WIDTH` to a multiple of the sub-group size, then each work-item in the same sub-group has the same `locali`. As an example, if `WIDTH` = 32, which is our sub-group size here, then `localj` has a range of zero to 31, and each has the same `locali`. This is important, as a sub-group is run simultaneously, and for branching, we would like to avoid divergence. With each work-item in a sub-group having the same `locali`, a branch with a `locali` in its condition will not cause divergence, since all work-items must take the same path. So no work-items need to wait needlessly. However, it still causes stalling in the execution pipeline and hinders optimization, so it does still cause some problems.

Also, generally in the implementation, care wasn't given to limiting ourselves on the j axis, meaning, for example, in the k loop, we have also overwritten values outside of the submatrix. Since with the upper triangular form, the submatrices should be at $i \leq j$, but we only check if we are below the ”column”. This behavior is wanted, as it removes branches, improving, for example, the ease of optimization for the compiler. We also do this in the last statement for updating ”columns” by just updating the whole line. Notice also the `(locali == localj) * ... + (locali != localj) * ...` structure. This is meant to filter if we are on the diagonal or not without needing a branch. It works by `locali == localj` or `locali != localj` simply becoming a zero if wrong and one if correct. So it then ”zeros” the wrong case and lets through the correct case, which are then merged with an addition, resulting in the value as if a branch were there. However, it is not exactly like a branch in this case, as it could be the case that one of the values is an atypical floating-point number, like NaN or Inf, and then the value assigned will also be atypical. But we don't care here since that would affect only the values in the padding. For normal values inside the L matrix, this won't happen, since they are from the beginning typical. There is the possibility of the square root getting a negative number or returning zero somehow, but that means that the input matrix doesn't fulfill the requirement of being positive-definite.

With a Cholesky Decomposition implementation within a block, we can now move forward to the Forward Substitution implementation.

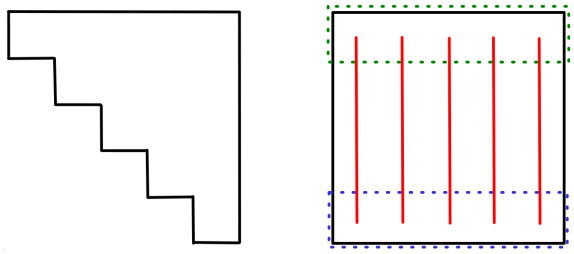


Figure 5.8: Left is the Cholesky block. Right is the vector matrix block with 5 vectors, as illustrated with the red lines. Also, the dotted green area is the ”top” of the vector matrix, and the dotted blue area is the ”bottom” of the vector matrix.

5.5.3 Forward Substitution

To keep things consistent, we'll now also implement the forward substitution as a function, taking in the Cholesky decomposed matrix L over local memory and a matrix block of the whole matrix with the vectors we need to calculate the forward substitution on. In other words, the forward substitution we are about to do executes the algorithm on multiple vectors at once. Figure 5.8 illustrates the inputs we require.

The local memory will again be a square block of size `WIDTH`, and our matrix block with the vectors will also be a square matrix of size `WIDTH`. We do this so we again have that each work-item points to exactly one spot in the local memory and one entry in the matrix block. The matrix block will not be in local memory but will be given through as a register, meaning each work-item gets its relevant value in the matrix through a register. This is sufficient since our implementation does not require the work-items to share their values. The implementation can be seen in listing 5.8.

The inputs are `triagblk` for the Cholesky matrix L and `x` for the singular matrix entry. To immediately dispel confusion, the implementation does, in fact, have a third parameter, which is `prevrslt`, that is simply additional local memory that will be needed for working on the implementation. The implementation works by implementing Algorithm 2.3, but now using many work-items. Remember, each work-item has a specific spot in the matrix of vectors. We will use this as our mental model for the implementation. Work-item index i specifies for what row of the vector it is used, and index j specifies what vector the work-item is on. We start off by having each work-item along the column of the vectors store the diagonal element of the Cholesky block it is associated with. We don't really need to do this, but due to each work-item later accessing it and it not changing, we can save some performance by storing it in a register, since registers have a faster access time, and we also free up potential bank conflicts that may arise. All in all, it is good practice to store values we can, even from local memory, outside of the loop. The Cholesky local memory block will not be changed by the algorithm, as we don't need to. Each work-item will calculate the output value of the row and vector it is on. So, for example, work-item $i = 0$ and $j = 0$ is responsible for the first vector's first row, so it receives the value there at the start through `x` and then outputs the solution to the same spot through `x`.

Next, we enter the loop. The loop has two relevant sections. The first is to calculate the current row's solution to the forward substitution, and the second is to take the solution and update all the rows below by subtracting. This is exactly an iteration of the i loop in Algorithm 2.3, with the first section doing the division, while the second section handles the subtraction. In the loop, the

Listing 5.8 Forward Substitution function

```
template <typename real_type, size_t WIDTH>
void solve_forwardsub(sycl::group<2> g,
                      sycl::local_accessor<real_type, 2> triagblk,
                      sycl::local_accessor<real_type, 1> prevrslt,
                      real_type &x) {
  const auto locali = g.get_local_id(0);
  const auto localj = g.get_local_id(1);
  const auto diag = triagblk[locali][locali];
  for (size_t k = 0; k < WIDTH - 1; ++k) {
    if (locali == k) {
      prevrslt[localj] = x /= diag;
    }
    sycl::group_barrier(g);
    if (locali > k) {
      x -= triagblk[k][locali] * prevrslt[localj];
    }
    sycl::group_barrier(g);
  }
  if (locali == WIDTH - 1) {
    x /= diag;
  }
}
```

division is guarded by an `if` to ensure that only the row we are on gets divided, since the other rows before are already done, and the rows after still need to be subtracted. As the `if` only depends on the `locali`, it shouldn't cause any branch divergence. Inside the `if`, we divide the current row by the previously saved diagonal and then store it in our other local memory section we included previously. This will be the task of the other local memory: for storing the solution of the current row so that the work-items below it can use it for subtraction. The barrier then guarantees that all vectors are done with the division.

The next part, as stated, does the subtraction. Orienting on Algorithm 2.3, b_i is here the `prevrslt[localj]`, which is the previously calculated solution on a row above, and L_{ki} is `triagblk[k][locali]`, which represents the inner upper triangle entries of the given Cholesky block. All of this is guarded by an `if` statement to ensure that we are indeed below the current row of the k loop, which is where the rows need to be subtracted. We then have a barrier again to prevent the values in `prevrslt` from being overwritten by faster sub-groups while still being used by slower sub-groups.

At the end, we still have to divide the last row by the diagonal, which is done with the last `if` statement. The `if` guards that the last row actually performs the division, since all the above rows are already finished. We have to do that, as the loop before is terminated one iteration earlier. This is done to avoid writing needlessly to `prevrslt` and to avoid the barrier. Notice that the k loop here and the k in listing 5.7 both use a constant for the termination condition, meaning a value that can be predicted at compile time. This is done explicitly, as loops in general consist of a loop body and a termination jump, which is a branch. To make optimization easier for the compiler and to prevent stalls in the execution pipeline, we would prefer not to have it. This is why, with a constant, the

Listing 5.9 The *update column* for the blocked cholesky algorithm

```
// Initialize a SYCL queue q and allocate enough triangle memory into gpuA and load in matrix
A in a blockbased data format with padding
for (size_t k = 0; k < Ndiv - 1; ++k) {
  e = q.submit([&](sycl::handler &cgh) {
    auto cholblk = sycl::local_accessor<real_type, 2>(sycl::range(WIDTH, WIDTH), cgh);
    auto prevrslt = sycl::local_accessor<real_type, 1>(sycl::range(WIDTH), cgh);
    cgh.parallel_for(
      sycl::nd_range{
        sycl::range(WIDTH, (Ndiv - 1 - k) * WIDTH,
        sycl::range(WIDTH, WIDTH)
      }, [=](sycl::nd_item<2> idx) {
        const auto g = idx.get_group();
        const auto groupj = g.get_group_id(1);
        const auto locali = g.get_local_id(0);
        const auto localj = g.get_local_id(1);
        const auto offseti = k;
        const auto offsetj = k + 1 + groupj;

        cholblk[locali][localj] = gpuA[index(locali + k * WIDTH, localj + k * WIDTH)];
        sycl::group_barrier(idx.get_group());
        cholesky_decomposition<real_type, WIDTH>(g, cholblk, jitter);
        sycl::group_barrier(idx.get_group());

        real_type x = gpuA[index(locali + offseti * WIDTH, localj + offsetj * WIDTH)];
        solve_forwardsub<real_type, WIDTH>(g, cholblk, prevrslt, x);

        gpuA[index(locali + offseti * WIDTH, localj + offsetj * WIDTH)] = x[k];
      });
  });
  // ...
}
// Calculate the last diagonal block with Cholesky
```

compiler will likely *unroll* the loop, meaning the loop goes again, and the loop body is repeatedly copied with an increasing loop iterator as much as the constant specifies. This is possible since the compiler knows exactly how often the loop is executed.

Combining both our implementations gives us our full *update column* routine, which then looks like in listing 5.9. In essence, it gets the diagonal and then calls the Cholesky Decomposition function with it, filling out the local memory with matrix L. Then it gets the matrix block, which, as stated before, consists of multiple vectors, and feeds the local memory with the matrix block into the Forward Substitution function. Both functions we have already described thoroughly. However, the k loop has changed. Since now we do not go through each k but go through each block, this was done with the help of the constant `Ndiv` $= \lceil N \div$ `WIDTH`$\rceil$. It states how many `WIDTH`s or blocks we need to encapsulate the entire N. So k here now means the current block. The `paralel_for` ND-range is then the top block row, since `Ndiv - 1 - k` means: take our row of blocks (`Ndiv`),

Listing 5.10 Matrix Multiply with a transpose with two blocks

```
template <typename real_type, size_t WIDTH>
void trans_matmul(sycl::group<2> g,
                  sycl::local_accessor<real_type, 2> leftblk,
                  sycl::local_accessor<real_type, 2> topblk,
                  real_type &sum) {
  const auto locali = g.get_local_id(0);
  const auto localj = g.get_local_id(1);
  sum = 0;
  for (size_t k = 0; k < WIDTH; ++k) {
    sum += topblk[k][localj] * leftblk[k][locali];
  }
}
```

subtract the submatrix we are currently on (k) as per Cholesky Decomposition, and finally take off 1 since the range should just have the ”column”, not also the diagonal. The work-group range is then just as expected: a square block of size `WIDTH`.

We should also not forget to decompose all the diagonal blocks again, but this time by filling them in. However, we will exclude showing the code, as it simply involves calling the Cholesky Decomposition again for each of the whole diagonal blocks.

5.5.4 Matrix Multiplication with the Transposition

For calculating the *update submatrix*, we must perform a matrix multiplication between the matrix of the top row and its transposition. We will again create a separate function, as it will come into play later. With the matrix multiplication, we must then subtract it from the submatrix, as described in Algorithm 2.5. However, with the function, we will not modify global memory to keep things tidy. The function will receive two blocks, which it will use to multiply together, but one of them will be transposed. It's visible in listing 5.10. Overall, it is just a dot product. That makes sense, since in this task we focus only on a single work-item, but we know that it runs on all of them. Meaning for just calculating one entry in the matrix, a dot product is required, which coincides with how matrix multiplication is done. We have two input parameters: `leftblk` and `topblk`. Parameter `leftblk` will be transposed; however, it is **not** expected of the function to transpose it. It may do so if it is necessary, but it does not have to. The function can also just transpose the indexation, which is what we are currently doing. The solution to the matrix multiplication is then outputted through the variable `sum`.

The function starts off by setting the summation variable to zero, then proceeds into the k loop. Note that we use a constant termination condition here again, allowing the compiler to unroll the loop if it deems it faster. Inside the loop, we perform the multiplication for the dot product. However, `leftblk` needs to be treated as a transpose, so we swap the indexing parameters. Keep in mind that this function will be called a lot due to being used in *update submatrix*, which has a time complexity of $O(n^2)$. So optimization here is very important. A thing to consider is bank conflicts. If we had taken a row-based indexing approach, then `topblk[k][localj]` should have been `topblk[localj][k]`, since the whole matrix multiplication would be transposed. Using `topblk[localj][k]` would result

Listing 5.11 The *update submatrix* for the blocked cholesky algorithm

```
// Initialize a SYCL queue q and allocate enough triangle memory into gpuA and load in matrix
A in a blockbased data format with padding
for (size_t k = 0; k < Ndiv - 1; ++k) {
  // Call update column
  e = q.submit([&](sycl::handler &cgh) {
    auto leftblk = sycl::local_accessor<real_type, 2>(sycl::range(WIDTH, WIDTH), cgh);
    auto topblk = sycl::local_accessor<real_type, 2>(sycl::range(WIDTH, WIDTH), cgh);
    cgh.parallel_for(
      sycl::nd_range{
        sycl::range((Ndiv - 1 - k) * WIDTH, (Ndiv - 1 - k) * WIDTH),
        sycl::range(WIDTH, WIDTH)
      }, [=](sycl::nd_item<2> idx) {
        const auto g = idx.get_group();
        const auto groupi = g.get_group_id(0);
        const auto groupj = g.get_group_id(1);
        const auto locali = g.get_local_id(0);
        const auto localj = g.get_local_id(1);
        if (groupi > groupj) return;
        const auto offseti = k + 1 + groupi;
        const auto offsetj = k + 1 + groupj;
        topblk[locali][localj] = gpuA[index(locali + k * WIDTH, localj + offsetj * WIDTH)];
        leftblk[locali][localj] = gpuA[index(locali + k * WIDTH, localj + offseti * WIDTH)];
        sycl::group_barrier(idx.get_group());
        real_type sum;
        trans_matmul<real_type, WIDTH>(g, leftblk, topblk);
        gpuA[index(locali + offseti * WIDTH, localj + offsetj * WIDTH)] -= sum;
      });
  });
}
// Calculate the last diagonal block with Cholesky
```

in a very bad bank conflict, due to each work-item accessing local memory column-wise, meaning all in the same bank, since `topblk` is a local memory region created with a size that is a multiple of the sub-group size. Coincidentally, this is also the amount of memory banks. A `topblk[localj][k]` would align itself perfectly onto a single bank, and the call would then be carried out sequentially for every single work-item in the sub-group. But due to our column-based approach, we avoid the situation entirely. This was also a consideration for why we ended up using column-wise indexing.

Using the function, we can now construct *update submatrix*, which will simply load in both relevant row blocks, perform the matrix multiplication with the transpose, and then subtract it from the submatrix. So we have listing 5.11 and other pieces required for 05 filled out with comments indicating what is called in that location. The implementation uses two local memory sections to load in the row blocks. These must, of course, have a square size of `WIDTH` to match the row blocks. The ND-range is then set to blocks of `Ndiv - 1 - k` to encompass the whole submatrix while still staying under k. Here, we have an `if` guard again, similar to implementation 04, since we have square blocks. It is sufficient to just look at the group index, as the top diagonal will always contain

a value if the block is indeed inside the upper triangular area. We are careful to offset all access to global memory, since the position is dependent on k and the group index affecting the blocks, while the local index affects the internal structure of a block.

We start by loading in the blocks from the top row, which is at k. We load one block offset by group index i and the other offset by group index j, since we ultimately want to use both blocks to calculate the matrix multiplication at the group index i and group index j block spot in the submatrix. This also shows parallels with the standard Cholesky Decomposition with $L_{ik}L_{jk}$. With both of the blocks, we perform our matrix multiplication with a transpose and subtract the result from the previously mentioned ij block spot in the submatrix.

This should complete the implementation of 05 and is now our first blocked Cholesky Decomposition implementation. By dividing the amount of time needed to go through the whole matrix, we achieve a sizable performance boost.

01	02	03	04	05
$5600s \pm 0.2$	$643s \pm 0.5$	$660s \pm 0.1$	$454s \pm 0.01$	$48s \pm 0.04$

Table 5.5: Progress overview up to 05 with $N = 60000$ and single-precision.

5.6 Final Cholesky (06) and (07)

Due to time constraints, we cannot, regrettably, describe implementation 06 and 07 in more depth. We will, however, briefly describe the trick used to further gain more performance. For 06, we noticed that in, for example, the *update column* implementation, 05 just gets the diagonal block and then updates the ”column” row to the right. So, each work-group loads the same diagonal. The idea is to take a single work-group and use it multiple times for updating multiple blocks on the row, by only loading the diagonal block once. This saves global memory access, and the amount of work can be increased until we either run out of registers to store the calculated results, not enough local memory for storing the multiple blocks to do at once, or we become memory bound. However, values determining how far we can go without needing another work-group are preferably known at compile time to promote loop unrolling, as the trick creates a lot of loops; also, the value must be manually tuned. In the *update submatrix*, the same concept can be used but in a two-dimensional way. This is then very similar to what we did in 04, but we handle multiple blocks rather than scalars, and we store multiple blocks in both dimensions, while 04 only stored multiple in one dimension.

The 07 implementation just changes 06 a little in how *update submatrix* goes through the row blocks. Rather than loading all the row blocks into local memory at the start, we only load one of the dimensions. The other dimension will load one block at a time. Then, in every iteration, the whole block line parallel to the loaded blocks will get updated, and then we switch the one block to the next block for the second iteration.

01	02	03	04	05	06	07
$5600s \pm 0.2$	$643s \pm 0.5$	$660s \pm 0.1$	$454s \pm 0.01$	$48s \pm 0.04$	$27s \pm 0.05$	$25s \pm 0.3$

Table 5.6: Progress overview up to 07 with $N = 60000$ and single-precision.

6 Implementation into PLSSVM

To implement the Cholesky Decomposition into PLSSVM, we still must consider that if we have a fully calculated matrix L, then we still need to perform the algorithm 2.2 to get an answer, meaning we still have to implement Forward and Backward Substitution implementations. To maintain a low runtime, we have decided to implement them also using SYCL, targeting the GPU. A big factor for the decision is the fact that we actually already have what we need, so we can reuse our optimized functions. The Forward Substitution in a blocked form can be described to work in two steps. First, we calculate the forward substitution of the top of all the given vectors with the top of L. This will be done in blocks of size `WIDTH`, since then we can just reuse our function `solve_forwardsub` in listing 5.8. With the top calculated, we enter the second step of updating the body of the vectors by using the L matrix row (remember it is still in an upper triangular form) to perform matrix multiplication with the top of the vectors matrix. Since L is upper triangular, it requires the block to be transposed before matrix multiplication. These are exactly the inputs for the previously created function `trans_matmul`. Then we repeat both steps again until the vectors are completely calculated.

These algorithms can then be implemented as shown in listing 6.1 as a mix of C++ and pseudocode. We do this to avoid introducing a new indexing method since the vector matrix isn't an upper triangular matrix. The indexing strategy is just rectangular indexing with padding on the end and bottom. The implementation is very similar to 05, only that we now include a `gpuB`, which is the pointer to the vector matrix block, and we don't perform the Cholesky decomposition anymore. Another difference is that we don't stop before the iteration at the bottom. Here, we go completely from top to bottom, but for the last iteration, we have an `if` statement that is used to omit the second task, since SYCL does throw an error for submitting a `parallel_for` with a range of size zero.

Starting with the first task, it holds a local memory block for storing the diagonal block of L and the one-dimensional local memory section `prevrslt` required for our function `solve_forwardsub`. The task goes through all the blocks currently at the top, as indicated by the outer for loop. The task first loads the diagonal block from matrix L into our setup local memory. The diagonal is at the same height as the top blocks, since it is the one that brings the top vector block to the solution. Then, we load a vector matrix block from the top and use our pre-made Forward Substitution function for a block. The result is then written to replace the read top block.

The second task is nearly equivalent to the *update submatrix* routine from 05, except that we load different data. Altogether, we have a blocked Forward Substitution implementation. However, we still require a Backward Substitution implementation. The matrix L is already transposed, so we can take it as is for the Backward Substitution. Since the Backward Substitution is just the Forward Substitution but from bottom to top, we can modify our blocked Forward Substitution implementation to perform the blocked Backward Substitution. For this, we need a function similar to `solve_forwardsub` but backwards. This is relatively simple to achieve by just flipping the direction of the for loops and, in the end, dividing the top row by the diagonal. With these changes to

Listing 6.1 Blocked forward substitution

```
for (Step through the vector matrix from top to bottom with WIDTH stepsize) {
  q.submit([&](sycl::handler &cgh) {
    auto triagblk = sycl::local_accessor<real_type, 2>(sycl::range(WIDTH, WIDTH), cgh);
    auto prevrslt = sycl::local_accessor<real_type, 1>(sycl::range(WIDTH), cgh);
    cgh.parallel_for(
      sycl::nd_range{
        sycl::range(All top blocks of size 32 in the vector matrix),
        sycl::range(WIDTH, WIDTH)
      }, [=](sycl::nd_item<2> idx) {
        triagblk[locali][localj] = gpuA[Load in the diagonal on the same height as the top];
        sycl::group_barrier(g);
        real_type x = gpuB[Load in a vector block along the top];
        solve_forwardsub<real_type, WIDTH>(..., triagblk, prevrslt, x);
        gpuB[Store the vector block back] = x;
      });
  });

if (The top is on the bottom block) break;

q.submit([&](sycl::handler &cgh) {
  auto topblk = sycl::local_accessor<real_type, 2>(sycl::range(WIDTH, WIDTH), cgh);
  auto leftblk = sycl::local_accessor<real_type, 2>(sycl::range(WIDTH, WIDTH), cgh);
  cgh.parallel_for(
    sycl::nd_range{
      sycl::range(The entire body of the vector matrix under the top),
      sycl::range(WIDTH, WIDTH)
    }, [=](sycl::nd_item<2> idx) {
      leftblk[locali][localj] = gpuA[Load a block from the row of L left the diagonal based on
 group i index];
      topblk[locali][localj] = gpuB[Load a block from the top based on group j index];
      sycl::group_barrier(g);
      real_type sum;
      trans_matmul<real_type, WIDTH>(..., leftblk, topblk, sum);
      gpuB[The relant block in the body based on group i and j index] -= sum;
    });
});
```

`solve_forwardsub`, we then get `solve_backwardsub`. With the function at hand, we can now describe the blocked Backward Substitution. It can be done in two steps. First, we take the bottom diagonal matrix and perform our new function `solve_backwardsub` with the bottom row of the vector matrix. This ensures that the bottom vector matrix now contains the solution. This then leads to the second step of using the calculated solutions and, this time, the column above the lower diagonal to update the body of the vector matrix above.

The need for the block column above the diagonal, rather than a block row, complicates things, as the blocks are already transposed, so just immediately using `trans_matmul` does not work. A fix can be to implement a `matmul` function; however, we decided to just transpose the blocks of the column as we read them. This way, we have code reuse and do not have to implement anything new. With the transposition, we can use `trans_matmul`, where the `leftblk` parameter is the matrix L transposed

column block, since which block we use along the column is dependent on the work-group index i in the vector matrix body. The other parameter, `topblk`, is then a bottom block of the vector matrix, since these blocks along the bottom are dependent on the work-group index j in the vector matrix body. The result is then used to subtract the relevant vector matrix block.

For the implementation of the Backward Substitution, we will just detail what to change in listing 6.1. The outer for loop now goes in the reverse direction and starts from the bottom block row in the vector matrix. The first task's ND-range is the same in size but intended for the bottom row, and in the task, we load the diagonal at the same height as the bottom vector matrix row, from where we also load in as `x` and write back to the same spot that was loaded. The `if` statement now breaks if the bottom is at the top. It will be reached since, remember, the for loop pushes up the bottom with each iteration. The second task's ND-range is now the body over the bottom row blocks, and we load a transposed (flipping the indices) column block from L based on the work-group index i along the column above the diagonal, and a block along the bottom of the vector matrix based on the work-group index j into their respective local memories.

6.1 Integrating Everything into PLSSVM

Now, with a Cholesky Decomposition implementation and Forward and Backward Substitution implementations, we have everything we need. Lastly, we need to integrate everything together and insert it as an additional solver type into PLSSVM. For the integration part, we have made it simple, as it's just enough to call each code in order. Our whole system for solving $AX = B$ works as follows. First, we load the symmetric matrix A adhering to the memory layout with padding. In our case, we also strip the lower inner triangular matrix from A. Then, we call our Cholesky Decomposition implementation. At this stage, A has become the decomposed L. Afterward, we load the vector matrix B, also adhering to the memory layout with padding. With it, we call our Forward Substitution code and immediately after, our Backward Substitution code. At this stage, B has become X, so our final step is to copy X out onto the host, and we can deallocate everything on the device.

In PLSSVM, as we are creating a new solver type, we thus create a new method reflecting the one implementing CG, that calls our Cholesky decomposition implementation with the signature `std::pair<soa_matrix<real_type>, std::vector<unsigned long long>> cholesky(const std::vector<detail::move_only_any> &A, const soa_matrix<real_type> &B, real_type jitter);`. It takes as input a list of devices with the matrix data A called `A`. Note: this input parameter always holds one element, as our implementation will not support having the matrix A on multiple devices. The matrix A is already preloaded onto the device with the data layout specified at section 5.5, and the pointer `gpuA` to the data on the device can be retrieved from the device datatype. Our method also gets the parameter `B`, which holds the vector matrix; however, unlike `A`, it is on the host. This means we do have to update it ourselves. Then there is the last parameter `jitter`. It was introduced rather late as of writing, so we will explain it here. The `jitter` parameter is a value that gets added to the diagonal values as they are used. This does skew the solutions, but it is meant as a last resort to help the Cholesky decomposition save itself from NaN values. In our case, the NaN values happen due to the square root that needs to be calculated on the diagonal. If a value is negative, then we have a problem. Normally, negative values should not happen; they occur only if the input matrix is not positive definite. However, if a positive definite matrix is close to being not positive

definite, then due to rounding errors because of floating point types, we can get a negative diagonal element. `jitter` fights this by ”lifting” up the diagonal, making it more positive definite; however, again, the addition of the `jitter` values will worsen results. We look more into `jitter` and its effects in chapter 7.

We then place our `cholesky` function next to where CG is called, but with an `if` statement to differentiate between the two, since we want to call only one of them. What the `if` statement takes depends on the solver type that is given by the user. In our case, we use the same name `cholesky` for the solver type. Training using the Cholesky Decomposition can then be done with `../PLSSVM/build/plssvm-train -l cholesky -b sycl --sycl_implementation_type adaptivecpp [dataset]`, with, for example, AdaptiveCpp.

7 Benchmarks and Results

All benchmarks and evaluations are done here using a NVIDIA A30 GPU with 24576MiB global memory and a AMD EPYC 9274F 24-Core Processor CPU on 409Gi RAM.

01	02	03	04	05	06	07
$5600s \pm 0.2$	$643s \pm 0.5$	$660s \pm 0.1$	$454s \pm 0.01$	$48s \pm 0.04$	$27s \pm 0.05$	$25s \pm 0.3$

Table 7.1: Progress overview up to 07 with $N = 60000$ and single-precision.

7.1 Methodology

For gathering the times for each milestone Cholesky Decomposition implementation, it would be possible to just run each implementation for each square matrix size N from 1 up to a certain limit and call it a day. Yet, such a benchmark strategy, while delivering the best results, would make the process take a long time. To avoid waiting long, we use the fact that it takes N iterations to calculate a Cholesky Decomposition for a square matrix of size N. For size $N + 1$, it then takes $N + 1$ iterations, resulting in a one-iteration difference. Adding the time for the iteration to $N + 1$ to the previous cumulative time for N would approximate the total time for $N + 1$. However, this method does not take into account the tasks after the loop, so it's an approximation. Due to the fact that the tasks afterward only have a complexity of up to $O(n^2)$, the approximation should still be relatively accurate, as the loop in total is $O(n^3)$. So, to save time, we opted to use this strategy for benchmarking.

For our SYCL code, we use the AdaptiveCpp compiler version 24.02.0 for the reason of being relatively simple to set up and the CUDA version for which it compiles is 11.8.0. By using its JIT compiler, we can greatly simplify the search for optimal compile flags, as the JIT compiler itself will optimize the code during runtime, eliminating the need for compile flags. For the C++ code, we use the compile flag `-Ofast` for best performance. This will also include `-ffast-math`, so the compiler is allowed to reorder floating-point operations, which may lead to problems. However, in our tests, the accuracies of the results stayed in an acceptable range, so we leave it on for better performance. We compile and evaluate both for `float`, which is single-precision, and `double`, which is double-precision.

For the data, we generated 5 random datasets of floating-point numbers, each representing an L matrix of width $N = 60000$. The values are between 0.5 and 2 on the diagonal and between -1 and 1 on the lower triangle. The values are chosen in this fashion to try to prevent large floating-point values for A, keeping the entries in A from spreading out too much due to matrix multiplication. We include multiple L matrices to see if different values change the runtime of the algorithm. This should be expected but we will then see it, if is does change something. A is then calculated using

$A = LL^T$, resulting in a positive-definite and symmetric square matrix. This is the case due to the Cholesky Decomposition theorem, as L contains positive diagonal entries. Numerical errors aren't of great importance here, but we would like to ensure that the calculation doesn't create any floating-point NaN values, as it may change the cycles taken for calculation on the GPU. So, we set the `jitter` parameter to 0.1, as it is enough to avoid NaN values for $N \leq 60000$. Note that the `jitter` parameter "liftsüp the diagonal, helping us avoid negative values under our square roots. Each implementation measures each dataset once for $N = 60000$ and then exponentially increases from $N = 1$, $N = 2$, ... up until $N = 2^{\lfloor \log_2(N) \rfloor} = 32768$. So, each implementation has 5 time measures for the same N. We then take the mean of each of the 5 measurements. Each algorithm measures their task launches individually, meaning the time a single iteration took is known. Also the time of the Forward and Backward Substitution is measured. One of the 5 datasets was taken with its associated L matrix and fed into 06. Its output values were recorded and compared with the original L matrix, calculating the Frobenius Norm Error. This was repeated with multiple jitter values to see how the parameter affects the output.

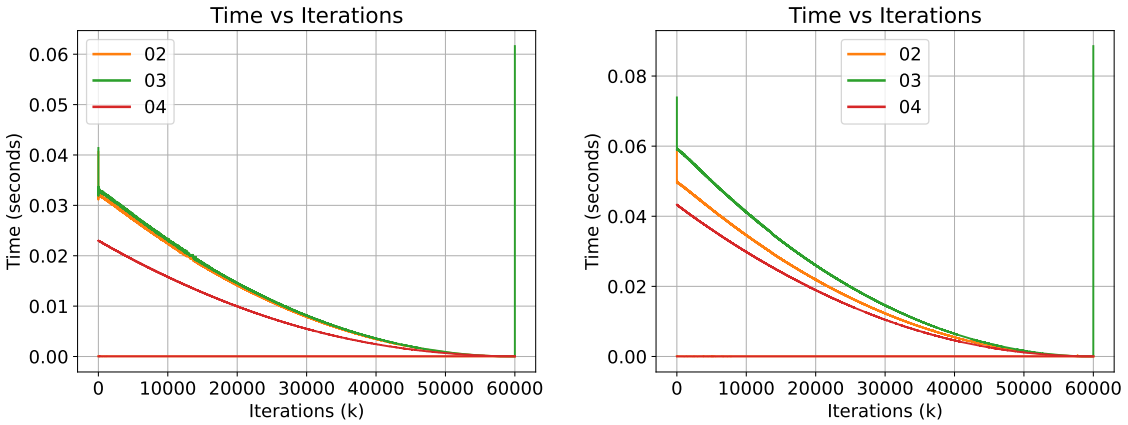
We will compare one dataset between the Cholesky Decomposition and CG using PLSSVM. We chose the MNIST dataset, which contains 60000 features, as it is a sufficiently large dataset comparable to our other datasets. We will compare both the runtime and accuracy. Training will be done 5 times, with one as a warm-up and the rest contributing to the mean. No hyperparameters were set for the CG, and it was using the SYCL backend with AdaptiveCpp for the best comparison.

Another factor we should consider is that by using AdaptiveCpp, we involve a JIT compiler, which, as previously stated, optimizes the code. This means it modifies the code, resulting in different runtimes. To obtain the best results, we first "warm up" the GPU by repeatedly running the program until the runtime stabilizes. We achieve this by running our task 5 times using $N = 8192$ before each measurement. This approach resulted in the runtime no longer changing by significant factors.

7.2 Cholesky Decomposition Benchmarks

Figure 7.1 shows the results measured for the parallel algorithms implementing the standard Cholesky Decomposition. Algorithm 01 is excluded, as it took $0.4s$ for double precision and $0.26s$ for single precision, which detracts from the readability of the other results. The figure consists of mainly two curves. The non-linear curves represent the *update submatrix* tasks, which run with a complexity of $O(n^2)$, as shown in the figure. In contrast, the linear curves, tightly packed at the bottom, correspond to the *update column* tasks, which run in $O(n)$ time. As observed in both figures, the *update column* tasks require significantly less runtime compared to the *update submatrix* tasks.

Examining algorithms 02 and 03, we observe that 03 performs worse, as illustrated in 7.1a. The difference is even more pronounced in 7.1b. Since we are targeting an NVIDIA GPU and SYCL's runtime translates our code into a CUDA kernel, we can leverage the tool *NVIDIA Nsight Compute* [25d] to investigate the underlying reasons why this happens. Using the tool, it becomes clear that our tasks are memory-bound, meaning the SMs of the GPU must wait for I/O operations and cannot proceed with computations. This explains the performance difference between 02 and 03. In 03, the additional memory load from accessing the top value in the diagonal of the *update submatrix* task results in even more memory accesses before any computation can occur. In contrast, the *update submatrix* task in 02 does not require the top diagonal value, as it calls the *update column*



(a) Using single-precision

(b) Using double-precision

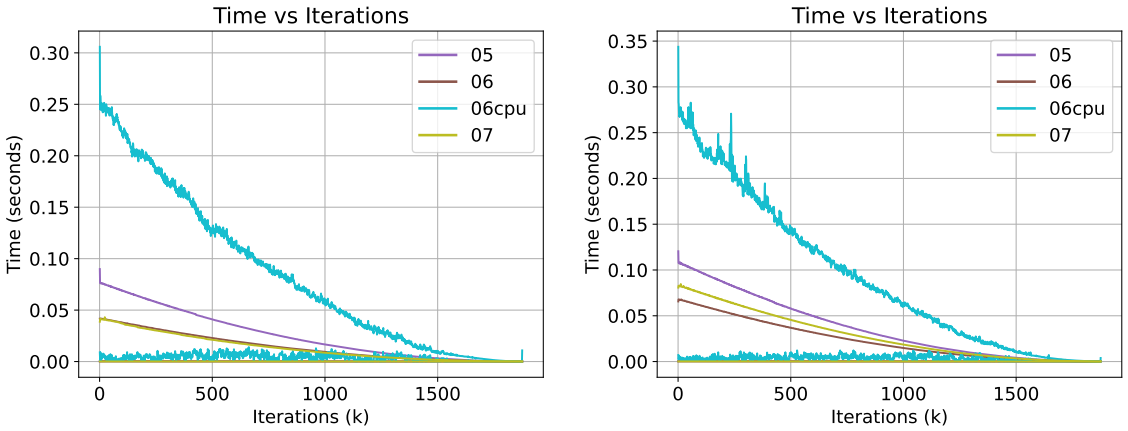
Figure 7.1: Time used for each iteration of the outer k loop for Algorithm 02, 03, and 04. Implementations are split into two parts: the *update submatrix*, which runs curves down from left to right, and *update column*, which is at the bottom. The horizontal line at the end is also valid time, representing a larger than normal task. The line at the beginning is probably a spike caused by the JIT.

task beforehand. This makes the requirement for the top diagonal value only $O(n)$, significantly less than the $O(n^2)$ time complexity associated with the *update submatrix* task in 03. Therefore, deferring the *update column* task until the end in 03 has actually worsened performance.

In 7.1, we observe a sharp incline at both the beginning and the end of the measurement for 03, and to a lesser extent for 02. The incline at the end for 03 corresponds to the *update column* task. Since the loop no longer performs *update column*, each column must be updated individually, which transforms the task into one with a complexity of $O(n^2)$. Both figures in 7.1 even show that *update column* is more expensive than an instance of the *update submatrix* task. However, since it is only executed once, it does not significantly impact overall performance. The incline at the start is less obvious to explain, but we believe it is most likely due to JIT compilation, as it smooths out immediately afterward. Again, since this is just a minor bump, it does not have a substantial effect on performance.

Algorithm 04, which utilizes local memory, performs the best, as expected due to the previously memory-bound nature of the task. By using local memory, 04 temporarily stores its global memory loads, which have a much faster access time, leading to the performance boost observed in 7.1. However, its performance is lower in 7.1b compared to 7.1a. This can be explained by the fact that double precision requires more storage. Therefore, for the same amount of local memory, fewer values can be stored compared to single precision. As a result, fewer values can be used simultaneously, leading to increased global memory usage, which ultimately results in worse performance.

In conclusion, the results from 7.1 demonstrate the success of our optimizations.



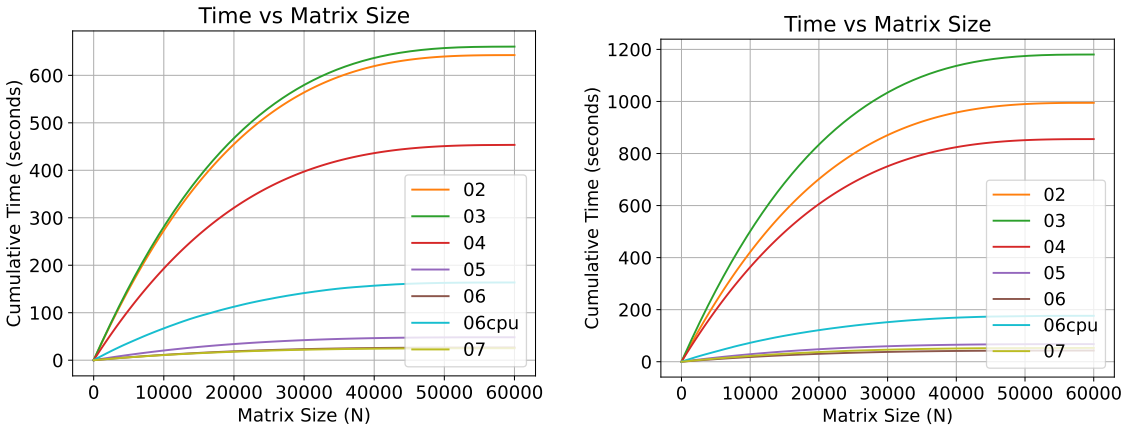
(a) Using single-precision

(b) Using double-precision

Figure 7.2: Time used for each iteration of the outer k loop for Algorithm 05, 06, 06cpu, and 07. Again, the implementations are split into two parts: the *update submatrix*, which runs curves down from left to right, and *update column*, which is at the bottom. The plot now also includes `06cpu`, which ran exclusively on the CPU.

Figure 7.2 displays the results for the algorithms implementing the blocked Cholesky Decomposition. In contrast to 7.1, the runtime appears to be larger. However, the algorithms require significantly fewer iterations, as indicated on the iterations axis. Since we are using blocking, we only need $60000 \div 32 = 1875$ iterations, due to the blocksize 32. This means we have significantly fewer terms to sum for the final runtime.

In 7.2, we see that algorithm 05 performs worse in both single- and double precision, while algorithms 06 and 07 achieve higher performance by utilizing their local memory and registers more effectively. In 7.2a, 06 and 07 are very close in performance, with 07 barely edging out as the better option, but only in the range from about $k = 100$ to $k = 1000$. It is possible that 07 could also perform better at $k < 100$, as the JIT compilation likely needed some iterations to identify the best-performing optimization. However, the relationship is inverted in 7.2b. Here, we see that 06 performs significantly better than 07, indicating that 06 handles larger value sizes more effectively. The reason for this is not immediately obvious, but it is likely due to the heavier access to local memory in 07 compared to 06, which negatively impacts performance in double precision, since larger value need to be read per access. As implemented, 06 is optimized for fewer local memory loads by storing intermediates in register memory and performing the entire calculation there, allowing each work-item to cover a rectangular area. In contrast, 07 loads a single row and then loads new subsequent rows only after the work-item has finished with the previous row, resulting in a higher number of local memory operations. This explains the results in 7.2b, but not those in 7.2a. It is likely that the JIT compiler optimized both 06 and 07 similarly, which may have only been possible in single precision, accounting for their similar performance. Most notably at the start. This behavior was not observed in other similar cases, such as in 7.1a for algorithms 02 and 03, where the runtimes, while close, diverged most at the beginning. For 07, the JIT likely provided a slight performance edge at $k = 100$.



(a) Using single-precision

(b) Using double-precision

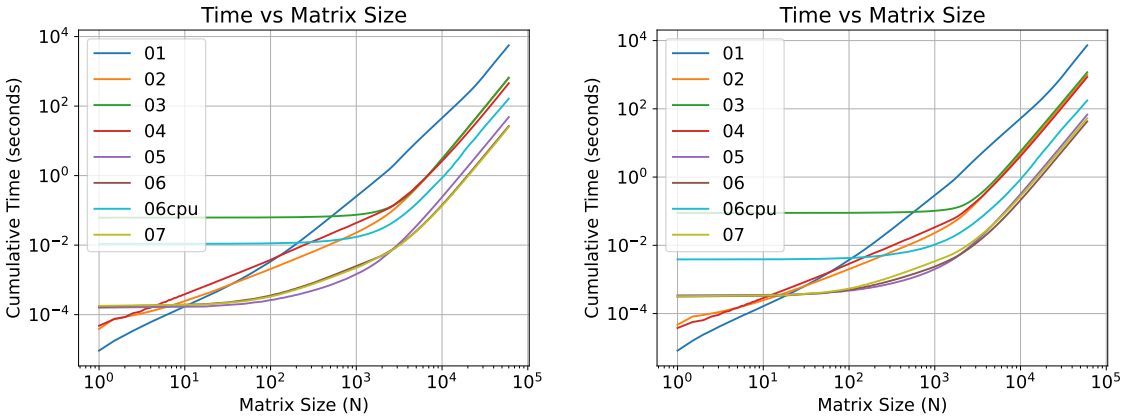
Figure 7.3: Cumulative time for each algorithm, meaning given a iteration we sum up all the time needed of all previous iterations up to that iteration. So where to line ends is the total time. These plots show us how the standard and blocked cholesky runtimes relate to each other.

For 06cpu in 7.2, we observe a similar curve to the measurements on the GPU, although it appears more fuzzy. We believe this is due to other processes interfering with the measurement, as the operating system itself needs to run, and since SYCL utilizes all CPU threads, there is little room left for other processes. The performance is significantly worse than the other measurements, but this is expected, as the GPU will generally outperform a CPU in terms of throughput. As seen in 7.1, we also observe starting bumps in 7.2 and the runtimes of the *update column* tasks positioned at the bottom. This is likely due to poorer JIT compilation and smaller time complexity, similar to the previous observations.

In conclusion, just as in 7.1, our optimizations for the blocked Cholesky Decomposition in 7.2 were also worthwhile for the most part.

Figure 7.3 combines both Figures 7.1 and 7.2 and accumulates the time, allowing us to see how much better fewer iterations are. We observe that the blocked variants perform significantly better in both single precision and double precision. Algorithm 01 is excluded yet again to enhance the readability of the figure. We even see that 06cpu runs much faster than the non-blocked algorithms.

The blocked Cholesky Decomposition, as seen in 7.2, requires fewer iterations, which reduces the number of tasks launched and consequently minimizes the overhead associated with task launches. This was a key reason for deciding to implement the blocked approach. However, as shown in 7.1a for implementations 03 and 02, algorithm 03 actually launches half the number of required tasks compared to 02, yet 02 is faster. Our microbenchmarking in the implementation indicates that more tasks tend to lead to longer runtimes, but in the case of 02 and 03, it seems that the time spent on memory operations is likely more costly here than doubling the number of task launches. Thus, while the number of tasks launched does play a role in runtime, it does not account for the substantial performance gain achieved with the blocked Cholesky Decomposition. Memory usage also plays a significant part in this context. Since we are only using 1875 iterations, we do not have to access memory as frequently as with the standard Cholesky Decomposition implementations.



(a) Using single-precision

(b) Using double-precision

Figure 7.4: Reversed cumulative time for each algorithm. This approximates how would have the algorithm acted it measuring every single $1 \leq N \leq 60000$ value.

This results in a significant reduction in global memory loads. For example, with a block size of 32, the blocked Cholesky Decomposition only requires a single pass through the entire matrix in the *update column* and *update submatrix* tasks. In contrast, the standard Cholesky Decomposition necessitates multiple passes through matrices of sizes N, $N-1$, $N-2$, ..., down to $N-31$. This quickly adds up and becomes costly in terms of runtime.

In conclusion, our focus on the blocked Cholesky Decomposition variants has paid off and should be preferred over the standard Cholesky Decomposition implementations when the matrix is large.

Due to the difficulty of measuring performance for every $1 \leq N \leq 60000$, we opted to use a different approach. Since we observed in 7.3 that the curve flattens out at the end of the algorithm, indicating that what happens after the loop has little effect, we can use the loop to obtain a close approximation for $N < 60000$. For both the standard and blocked Cholesky Decomposition, we first start with tasks involving the whole matrix and then decrease the size after each iteration. By reversing the order of the times taken and then accumulating them, we can derive a similar runtime to the first iteration for $N = 1$ in the standard variant and for $1 \leq N \leq 32$ in the blocked variant. The subsequent iterations continue to increase N by the iteration step until $N = 60000$. Using this method, we obtain the figure 7.4 from benchmarks of $N = 60000$. However, one thing to consider is the end of the loop. Nearly all loops in the algorithms terminate at an iteration step before reaching 60000, simply because the *update submatrix* task is not needed anymore. Thus, in 7.4 for the standard Cholesky Decomposition implementations, we load not just the value on the diagonal for $N = 1$, but also a single element below it. For the blocked variant implementations, we load the entire diagonal block along with a single block beneath it. This could lead to deviations from the actual performance. Additionally, this time we included results from 01, as the time-axis scaling does not become excessively large with its inclusion.

In both 7.4a and 7.4b, we observe that for lower values of N, the implementations deviate significantly in terms of runtime, with some implementations being unexpectedly faster than others. We notice that 03 is very similar at the start to the blocked variants 05, 06, and 07, but differs from the other standard variants. The reason for this is not immediately obvious, but we suspect it is due to JIT compilation. It seems that the compiler may have attempted to optimize 03 as if it were a blocked

algorithm, which could have improved performance. The constant reuse of the diagonal throughout the *update submatrix* task may have triggered such optimization. However, as we compare it to 02, 04, and even 01, it becomes clear that the optimization is not as effective, as it likely continues to expect larger N sizes for which it was optimized.

Among all the standard Cholesky Decomposition implementations in 7.4, 01 is the fastest, probably due to the very low overhead in the task, as it simply performs a `single_task` with a square root, thus not really utilizing the GPU at all, which it doesn't really need to. In comparison, 02 calculates a square root multiple times. Our previous favorite, 04, does not perform as well for smaller sizes, being close to 02 at the start but quickly deviating. It only becomes the fastest of the standard variants again at around $N = 1300$. This is likely due to the heavy padding and local memory overhead, which, for smaller values of N, are not used effectively. JIT compilation may also be a factor for values between $4 < N < 1300$, as it was optimized for larger N, which is how the measurements initially began.

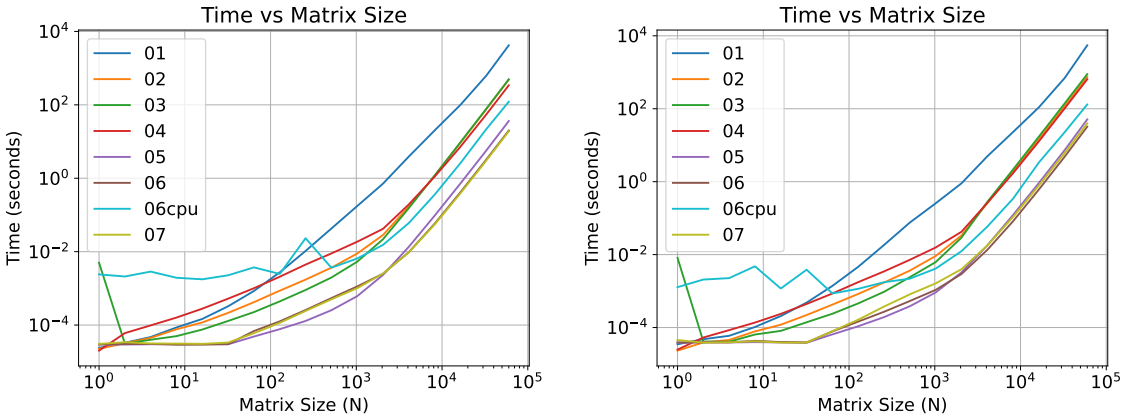
For the blocked variants, we also obtain unexpected results. Here, 06 and 07 do not perform well, while the simpler 05 emerges as the better option. In 7.4a, 06 becomes the best option again at around $N = 1200$, similar to the standard variants. This is likely due to JIT compilation, but also because of the heavy usage of registers, allowing work-items to calculate multiple entries per task invocation. While the work-items in 06 stop at loading unallocated memory due to the data not being large enough, the internal calculations still run, albeit with undefined register values. However, as we observe at $N = 1$, there is not a significant disparity in runtime. This makes sense, as stated before, since register memory is very fast, so the calculations should not have a substantial impact.

After about $N = 10000$ in 7.4, the runtimes begin to exhibit a linear trend with the expected incline of 3 in the log-log scale, indicating that our implementation is correct. At this point, the implementations start to sort themselves by the runtime seen in Figure 7.3.

In conclusion, reversing the data did not provide much value for analyzing the values at $N < 10000$, due to the theorized interference of JIT compilation, which will become clearer in 7.5. However, we do include Figure 7.4 as it highlights how significantly JIT compilation can affect the results. This suggests that our "warm-up" could have also worsened the results. However, we do not believe this is the case, as $N = 8192$ is still a relatively small size.

We have observed in 7.4 that JIT compilation could significantly influence how our implementations perform for smaller values of N and for very small N, our trick resulted in poor approximations. With true measurements for lower values, we can now see a more accurate picture in Figure 7.5. Here, we see that JIT compilation has made substantial changes, as the curves in Figure 7.5 are more compact at the start. Only the blocked Cholesky Decomposition implementations seem to follow a similar curvature to that in Figure 7.4.

For the standard variants, we see that 04 is even worse off than before, likely due to issues with heavy padding and local memory, causing 04 to perform more operations than necessary. Algorithm 03 stands out from the rest in both 7.5a and 7.5b, exhibiting a significant spike in runtime at the start. A possible explanation for this is the need to call two tasks, and as we know from the implementation microbenchmark, calling tasks incurs overhead. However, this explanation does not account for the performance at $N = 2$, where 03 is actually faster. The reason for this is not obvious, but it could be related to SYCL and the JIT compilation. Initially, we call a `parallel_for` with a single dimension, similar to the other implementations, but then 03 calls another `parallel_for` with two dimensions,



(a) Using single-precision

(b) Using double-precision

Figure 7.5: Runtime for each algorithm based on a matrix size

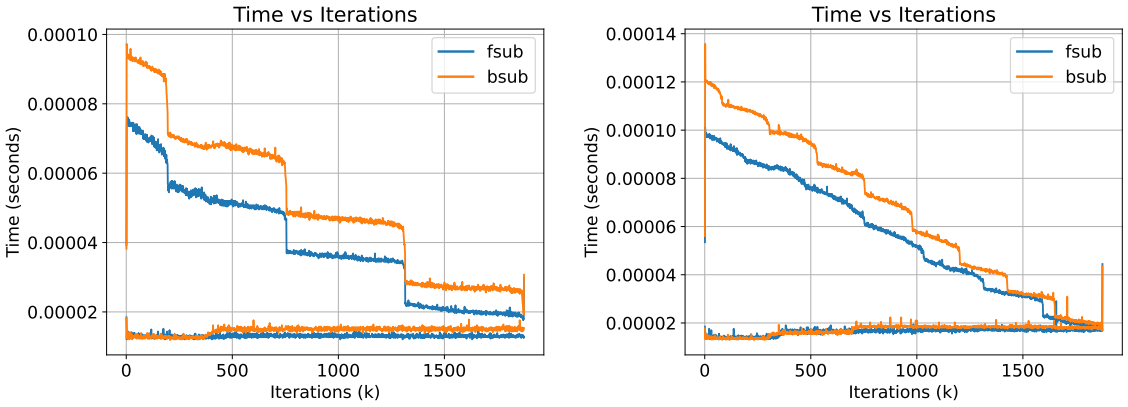
which may invoke a higher cost if it is being used for the first time. Yet, it still raises the question of why later iterations do not require as much runtime. Many factors influence GPU performance and it could simply be a peculiar hardware quirk affecting 03. However, this behavior is only observed for $N = 1$ and we are discussing milliseconds, so in the grand scheme, it is not a significant issue. Afterwards, 03 becomes the best-performing standard variant for both single and double precision, only later coming very close to 02 especially in 7.5a. This is expected, since there is an additional memory cost for double precision with larger N, which is where 03 starts to perform worse than 02.

Surprisingly, the blocked variants perform very well in Figure 7.5 compared to Figure 7.4, even though the blocked variants should involve much more memory.

In conclusion, we see that it is important to use an N in the ”warm-up” phase that is close to the value of N for which we want to measure performance. However, this consideration primarily affects lower values of N and if runtime measurement is important, warming up for value calculation is not really necessary. It appears that our 06 algorithm performs best or near best for both large and small matrices, as well as for single precision and double precision. Based on these results, we have decided to use the 06 implementation as our main Cholesky Decomposition implementation. As the value of N increases in Figure 7.5, we can draw the same conclusion as in 7.4: our specially optimized algorithms perform well for large values of N.

7.3 Forward and Backward Substitution Benchmarks

In Figure 7.6, we present the runtime for each iteration of our Forward and Backward Substitution implementation, similar to Figures 7.1 and 7.2. By fixing the number of B vectors to 32, we effectively achieve a time complexity of $O(n)$ for our reused *update submatrix* and $O(1)$ for our reused *update column*. This results in the small values observed in the figure.



(a) Using single-precision

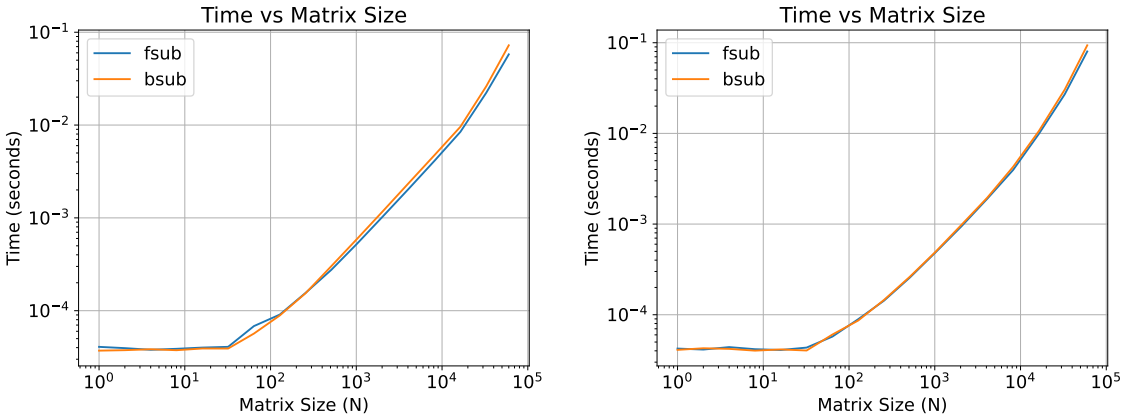
(b) Using double-precision

Figure 7.6: Time used for each iteration for the implementations of the Forward Substitution and Backward Substitution

The most obvious observation is that we can see a sort of stair step pattern. This could stem from the blocked nature of the algorithm, but the iterations are too wide. A case could be made as a result of JIT compilation, as the initial steps are much shorter than the subsequent ones, and at the very beginning, we have a runtime more akin to the time complexity of $O(n)$. JIT compilation most likely applied a blocked algorithm on top, as the runtime appears very similar to what we observed in 7.4. However this probabily isn't the case, since a JIT should be able to do such transformations, as it would change the algorithm and also likely the values which it isn't supposed to do. The `fsub` implementation also seems to be more efficient than `bsub`, which makes sense because `bsub` involves an additional block transpose and safeguarding bitwise operations, which require more time. While these operations are inexpensive, they likely have a non-trivial impact in this small time context. In 7.6b, the difference is not as pronounced, indicating that the larger values bring the algorithms closer in terms of performance. Additionally, in 7.6b, we observe the same stair-step pattern, but it is shorter, further validating our hypothesis that the JIT implementation is using some form of blocked algorithm. This is likely because, due to the larger size of double precision, fewer values fit in a block.

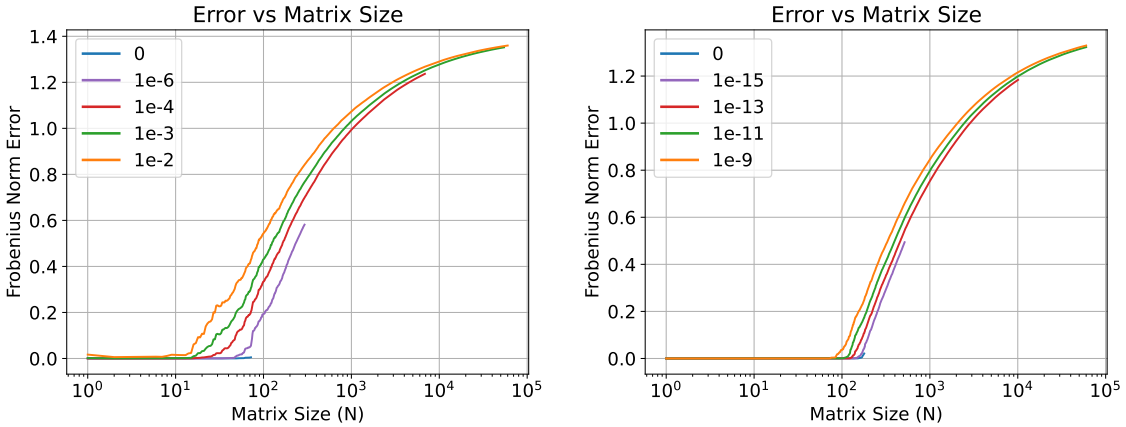
The runtime curve also appears fuzzy, which is expected, as we are in the time range of microseconds where the physical attributes of the hardware start to play a more significant role. The beginning bump of the curve shows, in contrast to previous measurements, much better performance. Again, this is likely due to factors outside our control. Looking at the *update column* algorithm, we see that both implementations are relatively similar, but `fsub` is slightly faster later on. In 7.6b, it even seems as if the JIT compilation has made performance worse. The reason for this is not obvious, but we suspect it is due to the optimization for the *update submatrix* part. As we observe in 7.6a, the moment `bsub` becomes more expensive for *update column*, there is a noticeable change in *update submatrix*, where the runtime becomes more blocky. In 7.6b, there appear to be two of these jumps. The fact that `bsub` is more expensive than `fsub` is likely due to the additional bitwise filtering that `bsub` must perform on its data.

We skipped the plot depicting the cumulative times, as it can be clearly deduced which algorithm is faster from Figure 7.6. However, we include a plot showing the algorithm runtimes for some N values, presented in Figure 7.7. In this figure, we observe a surprising trend of the runtime growing



(a) Using single-precision **(b)** Using double-precision

Figure 7.7: Runtime for `fsub` and `bsub`



(a) Using single-precision **(b)** Using double-precision

Figure 7.8: Resulting matrix error from different jitter values depending on the input matrix size A

by a factor of one in the log-log scale, because it should grow by a fector of two just like the time complexity says. However, it seems that as we increase N, the factor also increases, likely reaching two, as similarly observed in Figure 7.5. We also see in the figure that, in general usage, both algorithms do not exhibit significant differences in runtime, which is expected since they essentially use the same algorithms from our Cholesky Decomposition implementation.

In conclusion, our Forward and Backward Substitution implementations are quite fast, significantly faster than our Cholesky Decomposition implementation, so they do not have much effect on total runtime performance.

7.4 Numerical Stability of the Cholesky Decomposition

Figure 7.8 depicts our observation with the jitter parameter we used for the Cholesky Decomposition implementation. The plot displays how the Frobenius norm error between the actual L and the L calculated through the Cholesky Decomposition lies in relation to the matrix size. We notice first that the curves just stop at a point. This happens since, at that point, the matrix L starts to contain NaN values, most likely due to the square root operation that occurs on the diagonal. We see that the chosen matrix datasets do not allow the Cholesky Decomposition without any jitter value to reach a large N. However, if we start choosing higher jitter values, then the NaN values are propagated back. This is expected, as the reason for the jitter values is to ”lift” the diagonal, meaning that by adding a small positive value under the square root, we potentially save it from becoming a negative value, thus prolonging the computation.

We notice the curves are flat in the beginning and then there is a point where they start to rise very fast. The point at which the curve starts to rise depends on the jitter value. However, in Figure 7.8a, we see that at the start, we have a slightly higher error than immediately after. We think this is due to how the error generally works, as with a higher N, the erroneous values become more sparse since the matrix is generally correct, making the error more watered down.

But that certainly does not happen after the error starts to rise; instead, it converges together, probably due to the error becoming so bad that it becomes harder to get worse, along with the fast-growing number of elements in the matrix. However, in Figure 7.8b, we also see the error curve for the jitter value of zero bump up. Our theory is that the jitter value is not itself responsible for the big leap in error but helps to kickstart it. The big spike in error is a trait of the dataset itself, as it is probably ill-conditioned, which could be tested. Testing jitter values on a perfect matrix, like the identity matrix, or a matrix that is not the identity matrix but has a good condition would help to clarify what exactly is going on.

In conclusion, the jitter value is a helpful tool to prolong the values from becoming NaN during computation. However, it does lead to a faster increase in error for some datasets. The consequences of changing the jitter on the Cholesky Decomposition, in general, we cannot yet determine, but this is outside the scope of this thesis.

7.5 PLSSVM Benchmarks and Accuracy

Running the Cholesky Decomposition in PLSSVM on the MNIST data gave us a time of 47905 ms ± 585 ms for double precision and 42436 ms ± 55 ms for single precision. Checking the accuracy using the training dataset gave us 100% for both floating-point number types, meaning the model learned the training data correctly.

Now, running the CG in PLSSVM without tuning any parameters, also on the MNIST data, gave us a time of 3214 ms ± 45 ms for double precision and 2434 ms ± 19 ms for single precision. We see that CG has a 93% decrease in runtime for double precision and a 94% decrease for single precision. This makes CG significantly faster. However, checking the accuracy gave us only 11.24% for both single precision and double precision, and `plssvm-predict` warned about ill-defined results.

These results suggest that CG probably needs its hyperparameters tuned for it to produce an acceptable result. However, due to the warning from `plssvm-predict`, it could also be the case that the MNIST dataset was too large and resulted in NaN values. However, that is very unlikely, as MNIST is a very popular dataset, so CG has definitely been tested with it and should be able to pass with the correct hyperparameters.

8 Conclusion and Outlook

The goal of this thesis was to implement a Cholesky Decomposition using SYCL and integrate it into PLSSVM, which has been successfully achieved through the implementation of multiple versions of the Cholesky Decomposition algorithm and the selection of the best one for PLSSVM.

The project began by first learning about the Cholesky Decomposition algorithm itself, and we created initial basic prototypes that ran on the CPU. Shortly after, we started to learn about SYCL and how GPU execution works. To learn SYCL, we used a task for matrix multiplication as an example. It started off simply, with each work-item handling a single cell, but we ended up with an implementation that successfully utilized local memory to speed up its operation.

Using the background knowledge gained, we started implementing the various Cholesky Decomposition implementations presented in this thesis. 01 was quick to implement but quickly showed that a GPU isn't ideal for latency. Implementation 02 followed suit and was surprisingly an implementation that was very hard to beat in terms of our knowledge at that time. So, SYCL with AdaptiveCpp can be quite effective at optimizing simple implementations. However, many implementations later, the shell of 02 was cracked, and by finally using local memory correctly, it was then surpassed. Through much fine-tuning, we later achieved implementation 04. Many more attempts were made to beat 04, but they were not successful. However, we can safely say that effectively using local memory, if value sharing for a task is possible, is nearly essential if one wants to achieve speedy runtimes.

We then discovered the blocked Cholesky Decomposition, but unfortunately, it was a little late, leaving very little time for exploration. However, we carried on because it looked very promising. We managed to create a prototype; however, we were still using row-based indexing, which resulted in a bank conflict that made the implementation even worse in terms of runtime than 04. But after changing the data layout and removing any other bank conflicts, we achieved 05, which delivers a runtime much faster than 04. A while was then spent trying to find an even better implementation, which, in retrospect, we think could have been better used for the PLSSVM implementation and for writing this thesis. However, without that effort, 06 would not have existed, which is our fastest implementation.

With the implementation of 06 and the successful integration of everything into PLSSVM, the project has been successfully completed.

Looking further, the jitter parameter needs to be examined more closely, along with how the errors behave across multiple datasets, since our evaluation did not cover much. In terms of performance, there is probably nothing more we can do to 06 for a considerable time, but an interesting prospect is to explore SIMD operations within a work-item.

Bibliography

[18] *GPU optimized math routines in the Stan math library*. Aug. 2018. DOI: 10.5281/zenodo.1471582. URL: https://zenodo.org/records/1471582 (visited on 03/03/2025) (cit. on p. 31).

[25a] *AdaptiveCpp/AdaptiveCpp*. original-date: 2018-07-15T00:12:40Z. Mar. 2025. URL: https://github.com/AdaptiveCpp/AdaptiveCpp (visited on 03/03/2025) (cit. on p. 25).

[25b] *cuSOLVER*. en. 2025. URL: https://developer.nvidia.com/cusolver (visited on 03/03/2025) (cit. on p. 31).

[25c] *Khronos SYCL Registry - The Khronos Group Inc*. 2025. URL: https://registry.khronos.org/SYCL/ (visited on 03/03/2025) (cit. on p. 17).

[25d] *NVIDIA Nsight Compute*. en. 2025. URL: https://developer.nvidia.com/nsight-compute (visited on 03/05/2025) (cit. on p. 64).

[25e] *oneMath — oneAPI Specification 1.4-provisional-rev-1 documentation*. 2025. URL: https://oneapi-spec.uxlfoundation.org/specifications/oneapi/v1.4-rev-1/elements/onemath/source/#onemath-section (visited on 03/03/2025) (cit. on p. 32).

[25f] *SYCL — oneAPI Specification 1.3-rev-1 documentation*. 2025. URL: https://oneapi-spec.uxlfoundation.org/specifications/oneapi/v1.3-rev-1/elements/sycl/source/ (visited on 03/04/2025) (cit. on p. 25).

[BAB11] C. Brunelli, E. Aho, H. Berg. "OpenCL implementation of Cholesky matrix decomposition". In: *2011 International Symposium on System on Chip (SoC)*. Oct. 2011, pp. 62–67. DOI: 10.1109/ISSOC.2011.6089694. URL: https://ieeexplore.ieee.org/document/6089694 (visited on 03/03/2025) (cit. on p. 31).

[BJN+23] J. Bellavita, M. Jacquelin, E. G. Ng, D. Bonachea, J. Corbino, P. H. Hargrove. "symPACK: A GPU-Capable Fan-Out Sparse Cholesky Solver". In: *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1171–1184. ISBN: 9798400707858. DOI: 10.1145/3624062.3624600. URL: https://dl.acm.org/doi/10.1145/3624062.3624600 (visited on 03/03/2025) (cit. on p. 31).

[CJS+13] J. Chen, Z. Jin, Q. Shi, J. Qiu, W. Liu. "Block Algorithm and Its Implementation for Cholesky Factorization". In: 2013. URL: https://www.semanticscholar.org/paper/Block-Algorithm-and-Its-Implementation-for-Cholesky-Chen-Jin/2ed0daf1138e60271db772c4884f5277b19f50f9 (visited on 03/04/2025) (cit. on pp. 22, 48).

[TNLD10] S. Tomov, R. Nath, H. Ltaief, J. Dongarra. “Dense Linear Algebra Solvers for Multicore with GPU Accelerators”. In: *Proc. of the IEEE IPDPS’10*. DOI: 10.1109/IPDPSW.2010.5470941. Atlanta, GA: IEEE Computer Society, Apr. 2010, pp. 1–8 (cit. on p. 31).

[VBP22] A. Van Craen, M. Breyer, D. Pflüger. “PLSSVM: A (multi-)GPGPU-accelerated Least Squares Support Vector Machine”. In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 818–827. DOI: 10.1109/IPDPSW55747.2022.00138 (cit. on p. 17).

All links were last followed on March 04, 2025.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature