UNIVERSITY OF TWENTE

# An API for Intelligent Deployment of Numerical Calculations

Master Thesis
Embedded Systems - CAES

March 24, 2020


| | |
|---|---|
| Author | Navoda Perera (s2005336) |
| Supervisors | Dr. Daniel Ziener (UT) |
| | Dr. Ignacio Alonso (ASML) |
| | Dr. Artem Ivashko (ASML) |

# Abstract

Linear algebra is widely used in many scientific and engineering applications, where Basic Linear Algebra Subroutines (BLAS) and Linear Algebra PACKage (LAPACK) libraries are generally used in the back end for efficient calculations. There are hardware optimized versions of BLAS and LAPACK provided by vendors such as Intel Math Kernel Library (MKL) for Intel processors and Nvidia CuBLAS/CuSolver for Nvidia Graphical Processing Units (GPUs) respectively. As most of these routines can benefit from parallel processing, they have been shown to yield better performance on platforms that exploit parallelism such as GPUs and Field Programmable Gate Arrays (FPGAs) than general purpose processors. However, exploring these gains is not a trivial task in terms of software implementation.

This project explores the concept of an Application Programming Interface (API) for BLAS and LAPACK routines which abstracts the hardware specific details from the user and provides a convenient interface for deployment. A compute node with a CPU and a GPU connected through a PCI Express link is considered in the study. In addition to the convenience of deployment, it employs a model based approach to predict execution times and data transfer overheads based on input/output data sizes. These predictions are used in the optional dynamic deployment mode, where the deployment of a given routine will be decided at run time by the API. In addition, a data flow analysis method, which is also based on the mentioned prediction models, is presented to further improve the execution time of a given application code.

The API is shown to perform as expected with user-specified deployment, dynamic deployment and the data flow analysis modes. The two latter modes use performance models derived through an empirical approach to predict execution and data transfer times. Although considerable variations in execution time are observed for arbitrary sized inputs/outputs, all models are shown to predict with mean absolute percentage errors less than 20%.

# Acknowledgements

I would like to sincerely thank my supervisors Dr. Daniel Ziener from the University and Dr. Ignacio Alonso and Dr. Artem Ivashko from ASML for guiding me and providing me valuable advice from the onset of my project. It had been a very interesting experience working at ASML in a corporate environment, and I believe that it will greatly benefit my future career.

My heartfelt thank goes to Steven Van der Vlugt and Lennart Noordsij for the continuous feedback and encouragement throughout the project. I am also grateful to the other students and colleagues at ASML whose feedback and support had been very helpful. I take this opportunity to thank my graduation committee members Dr. André Kokkeler and Dr. Arnd Harmanns for the feedback provided on the project and the report.

Finally, I lovingly thank my family and friends for all the support given to me during my graduation project as well as in every step of my Master's program.

Navoda Perera
Eindhoven, Netherlands
March 24, 2020

# Contents

# Chapter 1

# Introduction

Linear Algebra is at the base of most science and engineering applications as it allows modeling and computing many natural phenomena efficiently. Today, linear algebra is being used in applications including but not limited to control systems, data science, computer graphics and machine learning.

The Basic Linear Algebra Subprograms (BLAS) [1] and Linear Algebra Package (LAPACK) [2] are two libraries that implement frequently used calculations in linear algebra. These are widely used in the backend of programs such as MATLAB [3] and libraries like Numpy [4] in Python. Reference designs for BLAS/LAPACK based on FORTRAN are available from Netlib [5], which is a common repository for numerical computing software. It is maintained by AT&T Bell Laboratories, University of Tennessee and Oak Ridge National Laboratory.

However, as different computer architectures emerged over time, vendors started introducing optimized implementations of BLAS and LAPACK for their hardware. With the advent of heterogeneous computing, this has expanded from general purpose processors to other accelerator platforms such as Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Some prominent examples of customized libraries are Intel Math Kernel Library (MKL) [6], AMD Optimizing CPU Libraries (AOCL) [7], Nvidia CuBLAS [8] and CuSolver [9] and Oracle's Sun Performance Library [10]. In addition, the active research in this area has further produced interesting work to include and improve BLAS/LAPACK for CPUs, GPUs and FPGAs alike. These will be further discussed in Chapter 3.

ASML [11] is a Netherland based company that manufactures photolithography systems for the semiconductor industry. These are highly complex machines that consist of a number of physical systems working in synchronization. Consequently, there is a large number of numerical calculations that happen in the software of the machine. This involves solving systems of linear equations, linear least square problems and other linear algebra calculations such as matrix multiplication. Hence, ASML maintains an internal *Numerical Library* that acts as a wrapper for BLAS and LAPACK routines. This library takes care of memory management and error handling, while providing a convenient interface for the programmers.

## 1.1 Problem Statement

With each new generation of ASML lithography machines, their computational requirements grow. Hence, hardware acceleration of software components is considered in order to achieve better performance. However, porting an application to a new hardware platform involves considerable effort. In case of platforms such as FPGAs, this requires more specific knowledge for development.

Due to the compute intensive nature of linear algebra operations, for large data sizes, they are

often the bottleneck of an application. Therefore, considerable gains can be expected even by accelerating just BLAS/LAPACK routines, instead of trying to port a whole application to a new platform. This expectation is based on the fact that parallel architectures such as GPUs and FPGAs have shown to produce gains of multiple orders of magnitude for most routines [8] [9] [12].

In addition, a key factor that affects performance gains with hardware deployment is data transfer times. In most cases, the data for the calculation needs to be transferred to the specific hardware from the host (CPU) through a high speed interconnect like PCI Express (PCIe). For smaller data sizes, usually the data transfer time dominates the execution time. That is, it may be faster to do the execution on the CPU, than to transfer to the GPU, execute and get the data back.

This phenomenon can be observed in Figure 1.1, in which the CPU and GPU execution times of matrix-matrix multiplication is plotted for increasing input sizes. The computational complexity ($m \times n \times k$) on the x-axis determines the number of operations involved in the calculation when two matrices of sizes $m \times k$ and $k \times n$ are multiplied. An additional curve (green) is plotted, which is the execution time on the GPU added with the input and output transfer times through the PCIe link. In the highlighted region the execution time on the GPU is less than that of the CPU. But when transfers are considered, the total time exceeds the CPU time. To the left of this region, the CPU execution itself is faster than the GPU.

In applications with multiple routines, some of these data transfers may become redundant depending on how they are deployed. For example, when the output of one routine is consumed by another, executing both of them on the same platform will avoid the need to transfer the output of the first routine. But this can be more complex when there are more routines and operations with such data dependencies, specially when some of them are required to always execute on a specific platform (e.g. control logic executing on the CPU, which use the output values of a routine). The design space becomes too large to explore different combinations manually as the number of routines in an application increases.

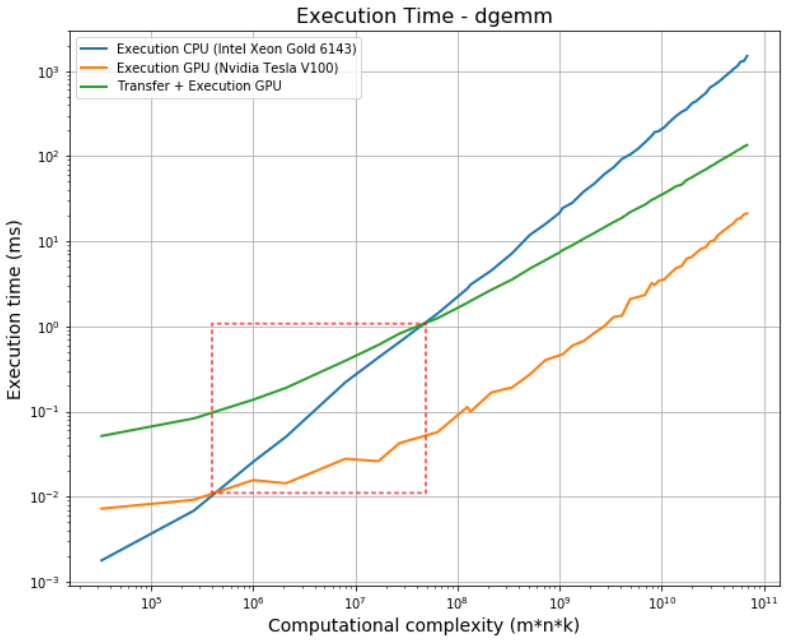



Figure 1.1: Execution time of matrix-matrix multiplication (`dgemm`)

In this context, ASML is interested in exploring the feasibility of an Application Programming Interface (API) which provides the convenience of quickly testing deployment options of BLAS/LAPACK routines. The aim of the API is to abstract all hardware implementation details from the developer, providing a simple interface to select the deployment platform for a given routine. The API needs to handle all data transfers involved, ideally avoiding redundant ones.

In addition, the API intends to explore multiple methods to make deployment decisions automatically. In our study we propose a runtime deployment feature and an offline deployment schedule generation feature based on profiled data of an application. The schedule intends to improve the performance of applications by minimizing the overhead of data transfers.

As mentioned earlier, the performance gains from deploying only BLAS/LAPACK routines may not be as optimal as a hardware implementation of the full application. But the effort involved by the application programmer is greatly reduced. This trade-off between design time and performance is justified in most applications where time-to-market is critical.

This study explores the feasibility of implementing such an API, that bases its decision making on performance models derived by an empirical method. The main research questions involved are:

1. How accurately can we predict the run time of BLAS/LAPACK routines for arbitrary input sizes?
2. To what extent can we use model based predictions for runtime deployment decisions?
3. How can we minimize the overhead of data transfers to improve total execution time of an application?

## 1.2 Approach

The scope of this project is limited to CPU and GPU platforms, although the implementation is such that it can be easily extended to other platforms. The CPU will be considered as the baseline for performance and as the main execution platform. The GPU is the hardware accelerator, which is to be utilized when beneficial.

In the project, the ASML Numerical Library (NL) is extended in three stages to develop the aforementioned API. First, for a selected set of functions, GPU implementations are introduced alongside their CPU counterparts. A convenient interface allows the user to select the platform of execution when calling these functions.

Then, the dynamic deployment feature is added, where the API will decide the best deployment option for a given function at runtime. These decisions are based on prediction models derived for the execution time of each routine on the CPU and the GPU, along with another model for the data transfer overhead. An empirical method of performance modeling is used to obtain these models.

Finally, a data flow analysis method is used in an effort to minimize total execution time of an application. The API will generate a data dependency graph of the linear algebra routines for a given segment of the application code. Based on that, a graph optimization algorithm is used to determine an optimized schedule of the routines to minimize total execution time, including data transfers. The API can then use this schedule information in a subsequent run for deployment.

# Chapter 2

# Background

This chapter introduces a number of key concepts that are relevant for this project. Section 2.1 gives a brief introduction and history of the BLAS and LAPACK libraries and the routines that will be used in this project. Sections 2.2 discusses the concept of performance modeling, while Section 2.3 provides a short description about ASML and a previous project in the company that is related to this project. Finally, Section 2.4 explains the hardware platforms and their architectures that are relevant to the study.

## 2.1 Linear Algebra libraries

As the use of numerical programming gained momentum in the 1970s, the usefulness of standard libraries was identified. As a result, BLAS was first published in 1979 for scalar-vector and vector-vector operations (BLAS level-1) as a Fortran library. Subsequently, during the 1980s, BLAS was updated with level-2 and level-3 operations for vector-matrix and matrix-matrix operations respectively.

The linear algebra library LINPACK was developed on top of BLAS for analyzing and solving linear equations and linear least-square problems. While BLAS was usually tailored for performance in specific hardware platforms, LINPACK was able to benefit from these optimizations without any modifications. Similarly, EISPACK was developed for numerical computation of eigenvalues and eigenvectors of matrices.

However, with the introduction of level-3 BLAS routines, LAPACK was designed as a successor to the above high level libraries. It combined the linear equation and linear least-square routines of LINPACK and eigenvalue routines of EISPACK. LAPACK was later extended to include ScaLAPACK [13], a subset which targets parallel distributed memory machines.

As the performance of BLAS and LAPACK routines became more critical, platform specific libraries such as Intel MKL, AMD AOCL and Nvidia CuBLAS/CuSolver were introduced by hardware vendors. There are also open source libraries like OpenBLAS [14] and ATLAS [15] with performance comparable to those of vendor implementations.

In addition, there are a number of libraries and frameworks developed by the research community targeting various hardware architectures. MAGMA [16] is one such project which is now available as a library [17] targeting heterogeneous architectures with multi-core CPUs and multi-GPU systems. BLASX [18] is another such library focusing level-3 BLAS on Multi-GPU systems. Furthermore, there are other implementations targeting FPGAs, GPUs and comparisons among the platforms, which are discussed in Chapter 3.

In this project, only vendor libraries are considered for the CPU and GPU due to two reasons. One

is that they yield good performance on their respective platforms. The other is reliability, because for ASML as a company, continuous maintenance and support of off-the-shelf libraries is important when maintaining its own software. Thus, Intel MKL and Nvidia CuBLAS/CuSolver libraries were used within the API.

### 2.1.1 Linear Algebra Routines

BLAS and LAPACK routines share similar naming structures for routines. The first character of the name identifies the type of data the routine operates on. Thus, there are 4 versions of each routine to handle each data type.

| Character | Data Type |
|---|---|
| `s` | real, single precision |
| `d` | complex, single precision |
| `c` | real, double precision |
| `z` | complex, double precision |

Table 2.1: Data types in routine interfaces

In addition, there are more information included in the names such as whether the operation is on a general (`ge`), triangular (`tr`) or Hermitian positive definite (`po`) matrix, or whether the routine involves matrix-matrix (`mm`) or matrix-vector (`mv`) operations. A detailed explanation of this naming scheme can be found in the Netlib website [5].

In this project, only double precision routines are considered mainly due to precision requirements. A subset of the routines was selected based on the their usage in the example application (explained in Chapter 4). The routines are introduced in the below subsections as relevant to the project scope. The full explanations of all features of each routine can be found with Netlib or MKL documentation.

#### 2.1.1.1 Matrix-matrix multiplication (`dgemm`)

The `dgemm` routine computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product. The operation is defined as:

$$C = \alpha A \cdot B + \beta C$$

where $A$, $B$ and $C$ are $\texttt{m} \times \texttt{k}$, $\texttt{k} \times \texttt{n}$ and $\texttt{m} \times \texttt{n}$ matrices respectively. These three dimension values (`m`, `n` and `k`) are inputs to the routine. $\alpha$ and $\beta$ are scalars, of which only $\alpha = 1.0$ and $\beta = 0.0$ cases are considered in the project scope. Matrices $A$ and $B$ can be optionally transposed at input using routine arguments `transa` and `transb`. The computational complexity of `dgemm` is given by $O(mnk)$.

#### 2.1.1.2 Matrix-vector multiplication (`dgemv`)

This routine calculates a scalar-matrix-vector product and adds a scalar-vector product to it. The general definition is given as:

$$y = \alpha A \cdot x + \beta y$$

where $A$ is an $\texttt{m} \times \texttt{n}$ matrix and $x$ and $y$ are `n`-element vectors. Here, `m` and `n` values are dimension inputs. $\alpha$ and $\beta$ are scalars of which we consider cases with $\alpha = 1.0$ and $\beta = 0.0$. Similar to `dgemm`, the matrix $A$ can be optionally transposed at input using a flag argument. The computational complexity of `dgemv` is $O(mn)$.

#### 2.1.1.3 Triangular matrix solver (`dtrsm`)

The `dtrsm` routine solves one of the following matrix equations:

$$A \cdot X = \alpha B$$
$$X \cdot A = \alpha B$$

where $X$ and $B$ are $\mathtt{m} \times \mathtt{n}$ matrices and $A$ is a upper or lower triangular matrix of size $\mathtt{m} \times \mathtt{m}$ or $\mathtt{n} \times \mathtt{n}$ (depending on whether $A$ is on the left or the right). The position of $A$ matrix and upper/lower property are given by an additional `side` and `uplo` flags. `m` and `n` values are dimension inputs. $\alpha$ is a scalar with $\alpha = 1.0$ in the example application. Matrix $A$ is only used in the lower triangular arrangement and on the left side (size $\mathtt{m} \times \mathtt{m}$). The computational complexity of `dtrsm` is defined as $O(m^2 n)$.

#### 2.1.1.4 Cholesky factorization (`dpotrf`)

The cholesky factorization of a Hermitian positive definite matrix $A$ can be obtained using `dpotrf`.

$$A = L \cdot L^T \; or$$
$$A = U \cdot U^T$$

The output will be overwritten on the input matrix $A$ of dimensions $\mathtt{m} \times \mathtt{m}$. The lower or the upper triangular matrix can be chosen for the output using the `uplo` flag. The other half of the matrix will contain garbage values. The computational complexity of `dpotrf` is $O(m^3)$.

#### 2.1.1.5 Matrix copies (`dlacpy`, `mkl_domatcopy` and `cublasDgeam`)

In the example application there are a number of matrix copies that involve additional actions. The `dlacpy` routine is used to copy all or part of a triangular matrix to another. The `uplo` flag can be used to specify whether to copy the upper or the lower triangle. `dlacpy` requires only one dimension argument as triangle matrices are square by definition. However, with current CuBLAS version (CUDA 10.1), there is no implementation of `dlacpy` for the GPU. Therefore, a naive implementation was used for the GPU, which performs a column-by-column copy of the triangular matrix using the vector copy (`cublasDcopy`) routine. For convenience, this implementation would be referred to as the GPU version of `dlacpy` in the report. However, it is to be noted that there certainly may be better implementations.

The `mkl_domatcopy` and `cublasDgeam` routines are used for copying parts of matrices to another, while optionally transposing and scaling the source matrix. Since there is no common BLAS routine that performs this, the additional routines provided by Intel MKL and CuBLAS are used in this project. The two routines are not identical in all their functionality. For example, `cublasDgeam` can optionally perform matrix addition as well. However, they were analyzed as each other's counter part because they were used for the same functionality in the project. To maintain consistency, the two routines will be commonly referred to as "Extended Copy (`excopy`)" in the report.

As all three routines involve copying 2D arrays, the computational complexity is taken as in the order $O(mn)$, where the (sub)matrix being copied is of size $m \times n$.

## 2.2 Performance Modeling

Performance modeling [19] in the context of this project is the process of mathematically modeling the behavior of software functions against various configurations. There are two main approaches to performance modeling: *Analytical* and *Empirical*. In analytical modeling, knowledge about the algorithm, underlying system and its architecture is used to derive a model. For example, the

execution time of a set of multiplications on a processor may be calculated by taking into account the processor frequency, the instruction set architecture (ISA) of the processor, memory hierarchy and speeds, etc. This obviously requires a deep understanding of the system and the models will largely vary from one platform to another. In addition, an understanding of the steps and operations in the algorithm is also required. However, when using closed source libraries such as Intel MKL and Nvidia CuBLAS, this knowledge is not available.

Empirical modeling on the other hand aims at deriving a mathematical model based on past measurements. For the above example, a set of execution time measurements may be taken for different numbers of multiplications and a model can be approximated based on them.

In this project, we use the empirical approach to model the execution times of a selected set of BLAS and LAPACK routines. This follows the work of a previous project at ASML which is explained in Section 2.3.1

## 2.3 ASML

ASML develops a range of lithography systems, which addresses a critical step in the silicon fabrication process. It essentially projects light through a blueprint of the pattern that needs to be printed on a silicon wafer. These blueprints are known as "masks" or "reticles". After the pattern is printed/exposed, the wafer is moved slightly and another copy is made. This process is repeated to cover the wafer and complete a single layer. In a usual microchip, there can be many such layers. Modern EUV lithography machines use Extreme Ultra Violet (EUV) light which has a wavelength of 13.5 nm, while older DUV machines use Deep Ultra Violet (DUV) light with a typical wavelength of 193 nm. Both these systems are used in the chip making process today.

The precision of these patterning techniques has become one of the defining factors for keeping up with the Moore's Law [20], a prediction made by a co-founder of Intel, Gordon Moore in 1965. His prediction was that the number of transistors placed in an IC will double about every two years. This reduction in size is largely influenced by the wavelength of the light source used in lithography.

In keeping up with this scaling and performance requirements, it has become critical for lithography system manufacturers to squeeze out as much performance as possible from the state-of-the-art technologies. A good example is this published study [21] of GPU acceleration for a real-time control loop in the ASML EUV lithography machine. The calculations that are being offloaded to the GPU predict the deformations on the wafer due to heating. The study shows that with the large amount of data and calculations involved, it is still more beneficial to compress the data, transfer to the GPU, decompress and process, rather than to execute everything on the CPU.

### 2.3.1 Previous Work

Prior to this project, there was a previous study done at ASML on performance modeling of linear algebra routines for the GPU [22]. It targeted the modeling of three selected BLAS/LAPACK routines: `dgemm`, `dpotrf` and `dpotrs`, which are matrix-matrix multiplication (BLAS), Cholesky factorization (LAPACK) and Cholesky solver (LAPACK) respectively. An empirical approach was used to obtain data transfer and execution time models for the GPU.

Measurements were made for square matrices of sizes ranging from $32 \times 32$ to $10000 \times 10000$. Then polynomials were fitted to the measurements using a $\chi^2$ minimization method. For all three routines, polynomials of third order have given reasonably good fits, while the independent variable used was the row/column size of the square matrices. This also falls in line with the fact that the selected routines have cubic computational complexity.

In addition, it was shown that a linear model fits fairly well with the measured data transfer times

over the PCIe link. This was also supported by the work of Boyer et al. in [23]. At the end, the models were able to predict with less than 20% relative error both for execution times and data transfer times. Therefore, the method has proven to yield reasonable results, without the need of having hardware specific knowledge as in the case for analytical modeling. More details about the outcome of this study and how it is being used in this project is discussed in Chapter 4.

## 2.4 Hardware Platforms

The two main hardware platforms considered in this study are the CPU and the GPU. Our test setup consists of an Intel Xeon processor and an Nvidia Tesla GPU, specifications of which are provided in Section 4.1. A general introduction to CPUs, GPUs and the Xeon and Tesla architectures is given in this section.

CPUs, also referred to as General Purpose Processors are designed to handle multiple types of tasks sequentially. Derived from the Von Neumann architecture [24], CPUs commonly share the concept of a centralized processing unit that executes a set of instructions from a stored program. They are usually designed to finish a task at as low as possible latency while keeping the ability to quickly switch between operations. Although there is a certain amount of parallelism introduced by multiple processing cores in today's CPUs, it is still comparatively limited.

From the initial single core CPUs operating in the megahertz frequency range, they have now evolved to having multiple cores operating in the gigahertz range. Modern processors employ techniques such as branch prediction, pipelining, speculative execution and vector instruction units to gain performance improvements. The Intel Xeon [25] is a series of processors targeted at workstation and server market, with up to 28 processor cores on a single chip. Each core in Xeon processors has access to its own L1 and L2 caches and a shared L3 cache, also known as the Last Level Cache (LLC).

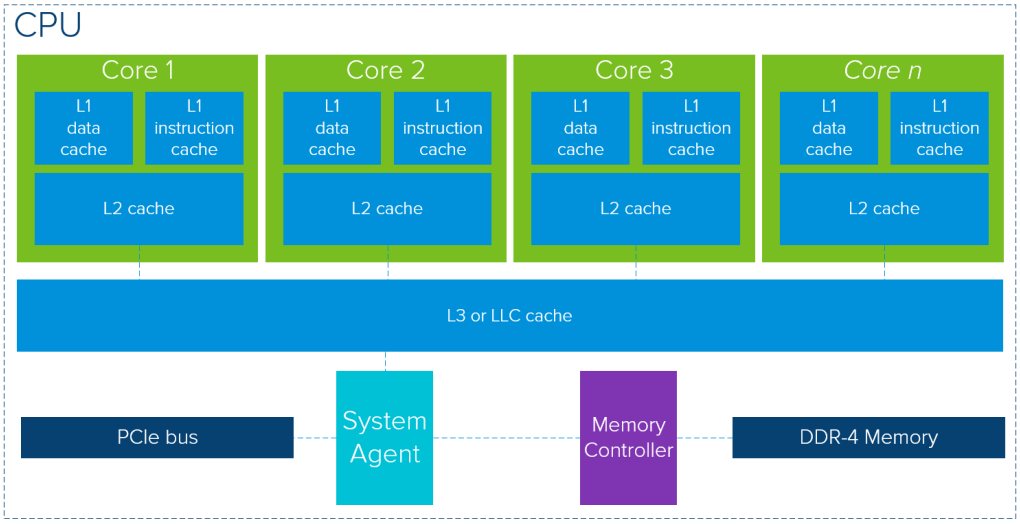



Figure 2.1: Generic block diagram of a CPU [26]

GPUs on the other hand were originally introduced for processing graphics, which involves a large number of calculations that could be done in parallel. Thus, they were designed to maximize the throughput by executing similar operations on thousands of processing cores. Over time GPUs were utilized in other applications that have massive data parallelism as well.

Nvidia Tesla is a range of GPUs targeting acceleration on data centers and servers. The Tesla V100 GPU that we use is based on the Volta architecture, which was introduced in 2017. The architecture

is based on an array of Streaming Multiprocessor (SM), each with 64 single precision (FP32) and 32 double precision (FP64) cores [27] (Figure 2.2). In addition, each SM contains multiple mixed-precision Tensor Cores purpose-built for deep learning matrix arithmetic. The Tesla V100 contains 80 such SMs.

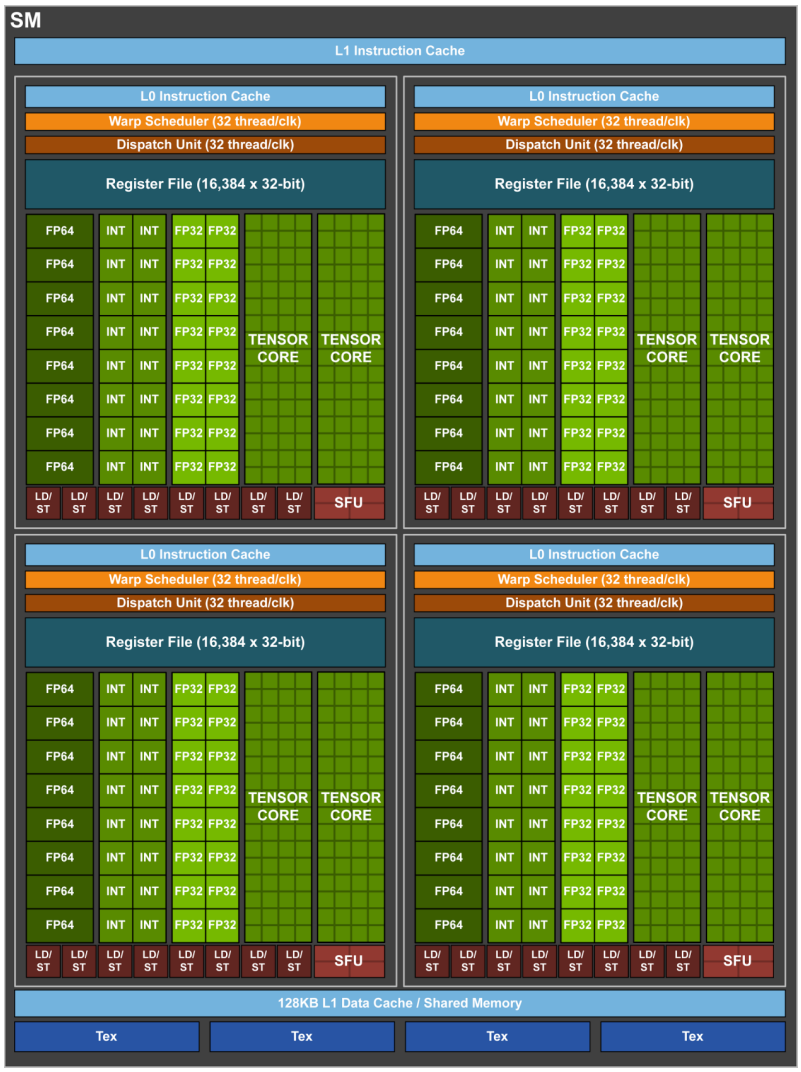



Figure 2.2: Volta GV100 Streaming Multiprocessor architecture [27]

In order to benefit from high processing power, it is also important to have complementing memory bandwidths. The Intel Xeon Gold 6134 processor that we use supports DDR4 (Double Data Rate) memory protocol, with a supported maximum bandwidth of up to 207.425 GB/s. In contrast, the Tesla V100 GPU has HBM2 (High Bandwidth Memory) interfaces to access the memory. It deliver memory bandwidths of up to 900 GB/s.

Another main factor influencing the performance of hardware accelerators is the interconnect used for communication between the platforms. PCI Express is a commonly used bus interface for communication on motherboards between the CPU and other peripherals such as GPUs. Multiple generations of the standard had been introduced, with 3.0 being the commonly supported one in latest devices. PCIe is able to detect the maximum version and the number of data lanes supported by two devices that are communicating and work in a compatible way. So even if a single lane device is connected to an 8 lane connector, the interface will work with the bandwidth of the single lane. Similarly, a generation 3 device connected to a generation 2 bus will work with the data rates of the latter. The PCIe generation 3.0 supports a single lane throughput of around 1.969 GB/s, with a combined throughput of 31.51 GB/s for 16 lanes [28].

# Chapter 3

# Related Work

## 3.1 Hardware acceleration of BLAS/LAPACK

Hardware acceleration of BLAS/LAPACK routines is an actively researched area. Apart from the libraries provided by vendors, there are a number of libraries with full or part implementation of BLAS/LAPACK developed by the open source community. Most of them are continuations of research work. In addition, there are also a number of standalone studies that discuss hardware acceleration of a few selected routines.

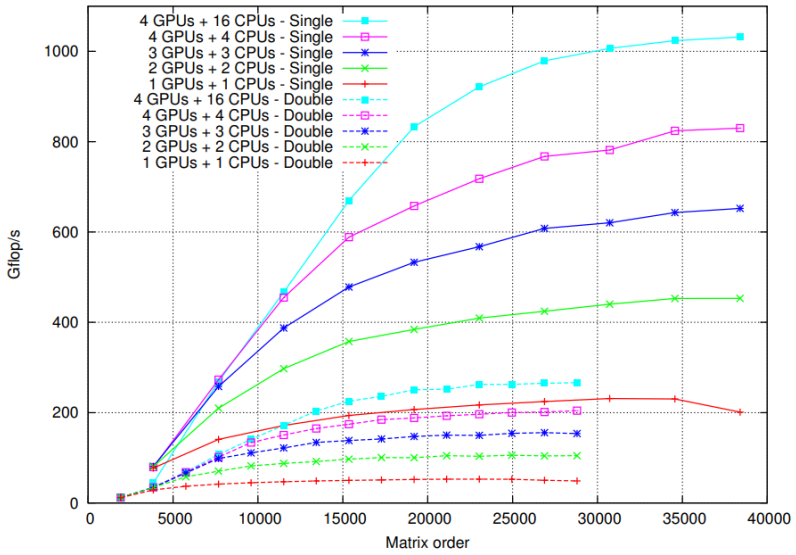



Figure 3.1: Scalability of `sgeqrf` and `dgeqrf` with multiple compute nodes using MAGMA [29]

MAGMA (Matrix Algebra on GPU and Multicore Architectures) [16] [17] is a library actively developed by by the team that developed LAPACK and ScaLAPACK (University of Tennessee), in collaboration with industry partners such as Nvidia, Intel, AMD and MathWorks. It focuses on exploiting the compute power of heterogeneous systems with multiple many-core CPUs and multiple GPUs. There are multiple implementations supporting the usage of GPUs with Nvidia CUDA and/or many core CPUs with OpenCL and Intel Xeon Phi support. The library executes small non-parallelizable tasks on the CPU and more parallelizable ones on a parallel architecture. Figure 3.1 shows how the performance of QR factorization for single and double precision scales with the number of CPUs (AMD Opteron) and GPUs (Nvidia Tesla) using MAGMA. However, this dynamic task scheduling in MAGMA is limited to deploying BLAS routines that are being used inside LAPACK functions to

improve performance. In contrast, in our project we explore the concept of deploying BLAS and LAPACK routines of an application in available platforms so that their individual execution times are minimized. That is, both BLAS and LAPACK functions are considered as similar indivisible execution units.

The MAGMA library also provides batched implementations for all BLAS level 3 routines and a selected set of LAPACK routines. This is shown to yield a few orders of magnitude of performance gains [30] compared to non-batched implementations. Applications which require calculations with large numbers of small matrices (e.g. Deep learning, astrophysics, structural mechanics, etc.) can specially benefit from batched calculations. In literature more studies can be found that try to exploit performance gains from batched executions [31] [32] [33].

In [31] the authors propose a standardized interface for Batched BLAS, so that developers are able to express many small BLAS operations as a single call. A comparison of performance between non-batched and batched implementations of matrix multiplication is provided for the CPU and GPU using MKL, CuBLAS and MAGMA libraries. The study makes use of batched BLAS implementations in CuBLAS [8] and batched matrix-matrix product (Batch GEMM) [34] in MKL.

In [32], Tabik et al. explore the impact of fusing vector operations (level 1 BLAS) to improve global runtime of iterative solvers on the GPU. The motivation is to reduce the inefficiency of doing those computations individually. The study shows up to 1.27x speed up for an iterative biconjugate gradient method (BCG) solver using this approach.

CLBlast [33] is an open-source library [35] providing optimized OpenCL BLAS routines operating in multiple precision levels. It is claimed to have been tested with a large variety of OpenCL devices and has an optional CUDA back-end. The routines can be auto-tuned or explicitly tuned for specific problem sizes and platforms. In addition the library supports batched execution of small calculations as well.

In our study, batched execution of routines was not included in the scope for manageability reasons. However, it is suggested as a possible addition to the API as future work in Section 8.1.

PLASMA (Parallel Linear Algebra Software for Multicore Architectures) [36] [37] is another project that is implemented by University of Tennessee along with other collaborators, including University of Manchester (UK), University of Colorado Denver and King Abdullah University of Technology. PLASMA focuses on using tiling algorithms to exploit parallel architectures more efficiently. The library uses a Directed Acyclic Graphs (DAG) representation of LAPACK routines to asynchronously schedule multiple operations on the available hardware. Similar to MAGMA, it only deals with BLAS calls within LAPACK routines, and thus differs from our work. However, we use a similar DAG related method to obtain deployment schedules for routines in an application.

When considering multi GPU systems, Nvidia provides its own implementation for level 3 BLAS called NVBLAS [38]. The library claims to dynamically route BLAS calls to multiple GPUs, while deploying to the CPU if the data transfer to the GPU is expected to be too high.

BLASX [18] is a similar implementation aimed at multi-GPU systems only for level 3 BLAS routines. In [18] the authors address techniques such as Peer-to-Peer (P2P) communication between GPUs and overlapping communication with computations. The authors present experiment results with up to 3 GPUs in the system, where BLASX had shown to out-perform MAGMA, cuBLAS-XT and PaRSEC implementations. PaRSEC (Parallel Runtime Scheduling and Execution Controller) [39] [40] is another framework maintained by Innovative Computing Lab (ICL) of University of Tennessee, which is used for scheduling micro tasks on distributed many core systems

In our study we work with a hardware setup with a single CPU and a GPU. Therefore, multi-GPU implementations were not considered in the scope. However, the API provides a generic interface for the routines. Therefore, new implementations can easily be introduced in the API, with additional selection options to the user. For an example, a MAGMA or NVBLAS implementation can be used if there are multiple GPUs in the system, so that it will utilize them for LAPACK calls with multiple BLAS routines. Similarly, the API can easily be extended to other computing platforms as well. Therefore, for the sake of completion, here we include a few studies on FPGA implementations of

BLAS routines.

An open source FPGA implementation of all BLAS routines is provided in [41]. It is available as a library developed using High-level Synthesis (HLS) tools, enabling better reusability, maintainability and portability across FPGAs. Moreover, the interfaces is designed to reduce off-chip communication when possible. The library yields execution times comparable with that of CPU, although it performs worse for smaller input sizes. The FPGAs rank better in terms of energy efficiency when compared against CPUs. The authors conclude that kernels that can be pipelined in multiple stages generally perform better on the FPGA.

An FPGA implementation of matrix multiplication (`dgemm`) is presented in [42] along with a software wrapper to transparently act as an accelerator. The design makes use of parallel multiply-accumulate (MAC) units and blocking algorithms on a dual-FPGA configuration. The study demonstrates a 60% performance gain over a CPU version of `dgemm` from the optimized ATLAS library. The performance is measured taking the data transfers into account as well.

Partial reconfiguration in FPGAs is a topic that is gaining popularity in the research community [43]. The work in [44] implements `sdot` and `sgemm` operations on a Xilinx Zynq SOC as a framework for dynamically configuring the FPGA for acceleration. It is shown that the FPGA yields speed ups of up to 7.26x compared to the CPU, while being more energy efficient.

A comparison of double precision performance and energy usage of BLAS on CPU, GPU and FPGA is presented in [45]. For the CPU and GPU standard Intel MKL and CuBLAS libraries are used. For the FPGA, custom implementations of dot product and matrix-vector multiplication are developed. With the test setup of an Intel Core 2 Duo 3.16 GHz processor and an Nvidia Tesla C1060 GPU, the CPU performs better for the `gaxpy` routine. However, only a limited range of input sizes are presented in the results. More focus is given to the FPGA implementation and its power efficiency.

Thus, it is clear that hardware acceleration of linear algebra calculations is a topic with active research and development. However, the concept of making deployment decisions for individual routines at runtime based on available hardware is not explored much to the best of our knowledge. This project explores the feasibility of the idea, using performance models to predict the execution times in the available platforms. A brief discussion of related work in the area of performance modeling is presented in the next section.

## 3.2 Performance Modeling

A comprehensive study on empirical modeling of BLAS/LAPACK routines for the CPU was done by Elmar Peise in [46]. In his work, Peise extensively explores different factors affecting the execution time of linear algebra routines including different input configurations and CPU conditions.

In the study, arguments to routines are broadly categorized into a few groups for analysis: *flag*, *scalar*, *leading dimension*, *increment*, *data* and *size*.

***Flag*** arguments usually specify actions or variations on the input/output matrices. They have 2 or more discrete values such as the transpose flag (`T/N`) on input matrices of `dgemm` or the upper/lower triangle (`U/L`) flag on the output matrix in `dpotrf`. As shown in [46], the flags have only a very slight influence on the execution time. However, in our project, we take the `trans` flag into account when modeling `dtrsm` and `mkl_domatcopy` routines as both values of the flag are used with those routines in the example application.

***Scalar*** arguments are used to multiply vectors or matrices with scalar values. In [46], it is shown that values `0` and `1` are treated as special cases in most routines. Of these, `0` appears to have the most effect where they seem to avoid certain calculations altogether. However, since in this project the scalar values are kept constant in the routines (`0` and `1`), the effect is not explicitly explored.

***Leading dimension*** determines the memory access strides of data. Since all data are represented as 1-D arrays, when working on sub-matrices, leading dimension is used to identify row/column boundaries. These arguments have very little influence on the execution time.

***Increment*** argument is similar to *leading dimension*. But it is used in the context of vectors. So for general cases, the stride length specified by *increment* would be `1`.

***data*** arguments are pointers to data arrays. As studied in [46], the actual numerical values in the data generally do not affect the runtime of routines. However, the fact whether the data resides in the cache or the main memory has some effect. However, studying the influence of caching is not included in the scope of this project to keep it manageable.

***size*** arguments specify the dimensions of the input and output matrices/vectors. They determine the minimal FLOP count of the routine and therefore directly influence the execution time. These are the arguments used in our study for modeling and predictions.

These groups were used in our study when implementing the test framework to collect measurement data for performance modeling. The grouping helped to make the C++ code more generic, so that new routines could be introduced for measurement with minimal effort. The open-source test framework [47] related with the above study was also referred during implementation.

In addition, the author uses a complex model generation process that derives multiple models for the same routine based on discrete flag and scalar argument combinations. For example, if a routine has two flag arguments: `trans` and `uplo`, the framework generates $2^2 = 4$ models with combinations of all the flag values (`trans=(T,N)`, `uplo=(U,L)`). Similarly, for `dtrsm` and `mkl_domatcopy` routines we use two models each for the two values of the `trans` flag, as the execution times were found to be considerably different and both values are used in the example application. However, as the application does not use differing flag values in the rest of the routines, we do not generate multiple models for them.

The modeling process in [46] recursively determines a set of dimension values in a range, makes measurements, fits a polynomial, checks if the model relative error is below a threshold of 1% and if not, divides the input range in two and continues for the two halves. However, this measurement and model generation method can be time consuming and when the framework tries to continuously divide the ranges to achieve higher accuracies it may over-fit the models to the data. Another drawback of having a large number of sub divisions for the same routine in our scope is that it would incur more processing at run time to compare the dimension arguments against the different ranges to determine the applicable polynomial to make a prediction. Due to these reasons and the implementation complexity, we follow a simpler approach of dividing the input ranges based on visual observations, focusing on absolute percentage errors below 20% (Section 5.1).

A comprehensive study on performance modeling for the CPU is also presented in [48] by Roman Iakymchuk. There, a bottom-up approach is used for modeling, starting at the BLAS level and then building up to higher levels using previous models. The author also explores empirical and analytical methods for modelling. However, only small problem sizes with square matrices that fit in the caches is used in the study. For empirical modeling, results for only LU factorization is presented, using models for `dscal` and `dger` routines, which achieve relative errors less than 4%. For modeling, piecewise polynomials are used with 2 parabolas, one for each cache level (L1 and L2) of the processors considered. In contrast, our work follows a more generic approach, considering arbitrary input sizes that at times exceed cache sizes and present models for 7 selected routines. In addition, we model the routines in a platform independent way, focusing on both CPU and GPU in our implementation.

Another focus of [48] is modeling memory-bound algorithms, and thus a detailed study is conducted on modeling memory stalls, which directly affects execution time. Similarly, the work in [46] includes an effort to model the effect of caching on the execution time of routines. However, this memory related modeling is not included in our study to keep the scope manageable. We recommend including it in future work for our API, due to its considerable influence on model accuracies as shown in the results (Chapter 7).

In [49], a supervised machine learning approach is used to predict GPU execution times based on CPU runs. Support Vector Machines (SVM) and Nearest Neighbor with Generalized Exemplars (NNGE) methods are considered for the purpose. Features such as number of ALU instructions, SIMD instructions, memory loads and stores and branches in the program are collected by runtime profiling. With this, Baldini et al. report an accuracy of 77% - 90% with their data sets. However, when a new application is introduced, it needs to be instrumented to run first with the runtime profiler. Then only the collected data can be used to predict the GPU execution time.

A comparison of machine learning based modeling and analytical modeling is presented for the GPU by Amaris et al. in [50]. For machine learning, Multiple Linear Regression, SVM and Random Forest are considered. The tests performed on multiple GPUs result in higher accuracy of analytical models with relative errors below 20%. The Linear Regression and Random Forest methods yield relative errors up to 50%. As both studies above require the application to be executed beforehand to collect profiling data, they are not suitable for our project, specially for dynamic deployment (Section 4.4) where the API predicts the execution times of BLAS/LAPACK routines at run time regardless of the nature of the application.

## 3.3 Task Scheduling with Data Dependency Graphs

Graph optimization is an actively researched area [51] in the scientific world as it can be applied in various fields. It is specially used in modeling the execution of parallel applications in homogeneous as well as heterogenous architectures. In our context, we represent the linear algebra operations in a given application as a Directed Acyclic Graph (DAG), which is then used to obtain an optimized deployment schedule of the routines on the CPU and GPU. Some of the most relevant work in literature is discussed in this section.

In the DAG representation, nodes of the graph are BLAS/LAPACK routines and edges represent data dependencies. The nodes are annotated with two costs: predicted execution time for the CPU and the GPU. The cost annotated on edges represent the predicted data transfer time over the PCIe link for the corresponding data size. This representation is discussed in more detail in Section 6.1.

The studies of scheduling dependent tasks can broadly be divided into the two categories of homogeneous [52] and heterogeneous architectures. We focus on the latter as our application needs to schedule routines between the CPU and the GPU architectures. Finding the optimal schedule of a DAG for heterogenous processors is an NP-complete problem even for two machines [53]. Thus, there are many heuristics in literature for scheduling. A comprehensive comparison of 20 such static heuristics is presented in [51].

Heterogeneous Earliest Finish Time (HEFT) [54] is a an often cited method for scheduling a set of dependent tasks on a network of heterogenous architectures. The algorithm provides an optimized schedule in $O(|V|^2p)$ time, where $V$ and $p$ are number of vertices and processing units respectively. However, being a greedy algorithm it is shown to perform poorly in some cases, where making short-term sacrifices may achieve better long term results [53].

In [55], the authors present a scheduling methodology which is able to take into account alternative implementations of the same component in different processing engines, as well as conditional branches in an application. However, the algorithm also allows preemption of tasks to free computation resources for higher priority tasks, which is not relevant for our application.

An algorithm (PEELSCHED) to schedule a set of tasks with precedence constraints and communication costs is presented in [52]. It is shown to produce schedules that are up to 29% longer than the optimum schedule for 2 processors. The study assumes that the processors are homogenous, meaning that the execution time of a given routines in each processor is identical. Therefore, it cannot be used in a heterogeneous system as ours, with a CPU and a GPU.

SPAGHETtI [53] is another algorithm for static scheduling of tasks on unbounded heterogeneous resources. It works in multiple steps. First the earliest start times of each node/task in the graph

is calculated with $O(|V| + |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges respectively. Then a method is proposed to map the tasks to the available architectures, assuming that there are unlimited instances of each architecture. This step takes $O(|A|^2|E|)$ time, where $|A|$ is the number of architectures (e.g. 2 if only CPU and GPU are considered). Then a final algorithm iteratively reduces the inherent parallelism in the graph by adding extra edges between parallel nodes, and re-calculates the schedule with the first two steps. This is repeated until the resulting schedule can be executed with the available resources. The algorithm is shown to perform better than HEFT, generating schedules with lower makespan, which is the time to execute the total schedule. Thus, this method is closely related to our study.

However, even after reducing the graph to the single CPU and GPU instances, it is possible for the above method to generate a schedule where there are nodes to be executed simultaneously on the two platforms. Implementing such functionality inside the API is non trivial, and therefore is not attempted in our study. Alternatively, it can be done by modifying the application code. But that approach goes against our goal of abstracting hardware specific details from the user by minimizing modifications outside the API. Furthermore, slight changes in input sizes or the application itself may result in different schedules, which will lead to further modifications for parallel deployments. Making such updates for each new schedule can be time consuming. Therefore, we only focus on executing routines one after the other in the order they appear in the application, while deploying each instance in an optimal way. Such a schedule cannot be generated using the SPAGHETtI algorithm.

Thus, in our work we suggest an intuitive algorithm which deploys routines sequentially while minimizing the total execution time of the application. Although it has certain limitations which will cause it not to scale well for larger applications and systems, it is presented as a proof-of-concept to demonstrate that the dependency graph can indeed be utilized to achieve better results than simple dynamic deployment at runtime. The algorithm is presented in Chapter 6, and the limitations are discussed in Section 7.2.1, after presenting the results of the study. However, within the scope of this project an attempt was not made to compare it against implementations of other algorithms in literature. Due to the large amount of research in this area, we recommend it to be done as a separate comprehensive study. It may also be of value to explore methods where parallel execution of multiple routines in available platforms is handled by the API as well.

# Chapter 4

# Approach

In this chapter, the test setup and the example application used to demonstrate the API is briefly introduced first. Then an overview of the API is given, followed by a discussion of a few approaches considered for the GPU implementation. Finally, the dynamic deployment mode of the API is explained.

## 4.1 Test Setup

In this study, all measurements and tests were executed on a High Performance Cluster (HPC) node with an Intel Xeon Gold 6134 [56] CPU and an Nvidia Tesla V100 SXM2 [57] GPU. The hardware and software specifications of the system are summarized in Table 4.1.

| | | |
|---|---|---|
| CPU | Model | Intel Xeon Gold 6134 |
| | Number of cores | 8 |
| | Number of threads | 16 |
| | Processor base frequency | 3.20 GHz |
| | L1 cache size | 32 KB per core |
| | L2 cache size | 1 MB per core |
| | L3 cache size | 24.75 MB shared |
| | Memory interface | DDR4 |
| | Maximum memory bandwidth | 207.425 GB/s |
| GPU | Model | Nvidia Tesla V100 SXM2 |
| | No. of CUDA Cores | 5120 |
| | Architecture | Volta |
| | Memory interface | HBM2 |
| | Maximum memory bandwidth | 900 GB/s |
| Interconnect | PCI Express version | Generation 3.0 |
| | Number of lanes | 16 |
| Software | BLAS/LAPACK Library - CPU | Intel MKL 2019 Update 3 |
| | BLAS/LAPACK Library - GPU | Nvidia Cublas/Cusolver (CUDA 10.1) |
| | C/C++ Compiler | GCC 4.8.5 |
| | Operating system | RedHat Enterprise Linux 7 |

Table 4.1: Test setup specifications

Only single thread executions were considered for the CPU in the project. The main reason for this was to keep the project scope manageable. The previous study at ASML [22] showed that increasing the number of threads does not always reduce the execution of of a routine. For example,

the run time of of `dgemm` with two $128 \times 128$ matrices was more than doubled when the number of threads was increased from 4 to 24. And this kind of behavior differs from routine to routine. Therefore, exploring the effect of multiple threads for all routines would have been a non trivial task.

## 4.2 Example Application

Linear solvers are used in many systems in ASML software. In this project, in order to demonstrate the API, a convex optimization problem using the Quadratic Programming (QP) active-set method was selected as an example application. The objective in quadratic programming is to minimize/-maximize a quadratic function of several variables subject to a set of linear constraints that define the feasible region:

$$\begin{aligned} &minimize && \frac{1}{2}x^T Hx + c^T x \\ &subject\ to && Ax \leq d \end{aligned}$$

where,
$c$: an $n$ - dimensional real vector
$H$: an $n \times n$ - dimensional real symmetric matrix
$A$: an $m \times n$ - dimensional real matrix
$d$: an $m$ - dimensional real vector

It has been shown that the problem gets simplified when $H$ above is positive definite and all constraints are equality constraints. These equality constraints are also called the *active-set*. Then, it can be shown that the solution to the problem is given by the linear system

$$MX = F$$

$$\begin{bmatrix} H & A_{as}^T \\ A_{as} & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -f \\ d \end{bmatrix} \tag{4.1}$$

where, $\lambda$ is a set of Lagrange multipliers which are received along with the solution x. $A_{as}$ is the set of active constraints. In the *active-set* method of solving QP problems, first a feasible starting point is selected in the feasible region. Then after determining the active set for the selected point, the above system is solved to calculate a step within the boundaries. This is repeated until it satisfies all conditions, reaching the optimal solution. Alternatively the algorithm can be stopped after a predefined number of iterations, if the solution is already feasible.

However, in our implementation, we have not implemented convergence and bound checks that are in a typical application of this algorithm [58], as the aim was to demonstrate the use of BLAS/LAPACK routines with the NL API. In the tests, the data was randomly generated and the number of constraints in the matrix equation was continuously increased to demonstrate an increasing problem size.

The straightforward method to obtain the solution of the above problem is through LU factorization. However, doing this in each iteration would be sub-optimal because the upper left part of $M$ with with $H$ matrix remains fixed. However, when the $H$ matrix is positive definite, it can be shown that $L$ and $U$ matrices can be constructed using and re-using a set of smaller matrices:

$$L = \begin{bmatrix} L_c & 0 \\ B_{as} & L_2 \end{bmatrix}$$
$$U = \begin{bmatrix} L_c^T & B_{as}^T \\ 0 & -L_2 \end{bmatrix}$$

Where,
$L_c = chol(H)$

$B_{as} = A_{as} \cdot (L_c^T)^{-1}$
$L_2 = chol(A_{as} \cdot H^{-1} \cdot A_{as}^T)$

Here, $L_c = chol(H)$, which is the cholesky factorization of $H$, can be pre-calculated as $H$ does not change during iterations. The sub-matrix $B_{as}$ can be obtained by pre-solving $B_t \cdot L_c = A$ and selecting the same constraints (rows) from $B_t$ that correspond to the active-set, as $A_{as}$ would have been constructed from $A$. Finally, it can be shown that $L_2$ is obtained by Cholesky factorization of $B_{as} \cdot B_{as}^T$:

$$\begin{aligned} B_{as} \cdot B_{as}^T &= \left[A_{as} \cdot (L_c^T)^{-1}\right] \cdot \left[A_{as} \cdot (L_c^T)^{-1}\right]^T \\ &= \left[A_{as} \cdot (L_c^T)^{-1}\right] \cdot \left[(L_c^T)^{-1}\right]^T \cdot A_{as}^T \\ &= \left[A_{as} \cdot (L_c^T)^{-1}\right] \cdot \left[(L_c^T)^T\right]^{-1} \cdot A_{as}^T \\ &= A_{as} \cdot (L_c^T)^{-1} \cdot L_c^{-1} \cdot A_{as}^T \\ &= A_{as} \cdot (L_c \cdot L_c^T)^{-1} \cdot A_{as}^T \\ &= A_{as} \cdot H^{-1} \cdot A_{as}^T \end{aligned}$$

Matrix inverse and transpose rules used for above derivation are:

$$(A \cdot B)^T = B^T \cdot A^T$$

$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

$$(A^T)^{-1} = (A^{-1})^T$$

**Algorithm 1:** Quadratic Solver algorithm

/* $n$: Size of $H$ matrix */
/* $m$: Total number of constraints / number of rows in $A$ */
/* $max\_iter$: maximum iterations */

// Calculate $L_c$ and $B_t$ matrices
$L_c = choleksy(H)$
$B_t = tri\_solve(L_c, A)$
**while** *iters < max_iter* **do**
  // Select active constraints from $B_t$
  $B_{as} = B_t[idxs]$
  // Calculate $L_2$ matrix
  $L_2 = B_{as} \cdot B_{as}^T$
  // Construct L and U matrices
  $$L = \begin{bmatrix} L_c & 0 \\ B_{as} & L_2 \end{bmatrix}$$
  $$U = \begin{bmatrix} L_c^T & B_{as}^T \\ 0 & -L_2 \end{bmatrix}$$
  // Forward and backward substitution
  $s_f = tri\_solve(L, f)$
  $X = tri\_solve(U^T, s_f)$
  // Additional calculations for representation
  $x :=$ first n elements of $X$
  $adx = A \cdot x$
  copy $adx$ to another vector on CPU
  iters++
**end**

For a given problem with $H$ of size $n \times n$ and the number of constraints $m$, the algorithm of the solver is presented in Algorithm 1.

The functions in the algorithm, $cholesky()$ and $tri\_solve()$ represent cholesky factorization (`dpotrf`) and triangular matrix solver (`dtrsm`) routines. Two additional calculations were added at the end of each iteration, in order to represent the steps that follow the matrix equation solving in an actual application. The matrix-vector multiplication $A \cdot x$ is one of the steps in determining blocking constraints at a given point of a QP solver. We include it in our application model due to its considerable computation cost. The last copy of the resulting vector to the CPU is added to represent any control logic that might happen on the CPU at the end of each iteration. This way, the result is forced to be transferred back to the CPU regardless of where $A \cdot x$ is calculated.

## 4.3 API Overview

The ASML Numerical Library API already consisted of a number of functions that wrap BLAS/LAPACK routines. They were updated and more functions were added during the implementation of this project. In addition, for all the functions involved, an optional *deployment flag* argument was introduced. With this flag the user is able to specify the expected deployment platform (`DEP_CPU`, `DEP_GPU`) for the function (Figure 4.1). Since the implementation including data transfers is handled by the API, the application code does not need to be changed any further. When the value is not given, the routine will execute on the CPU by default. Hence, the interface is backward compatible.

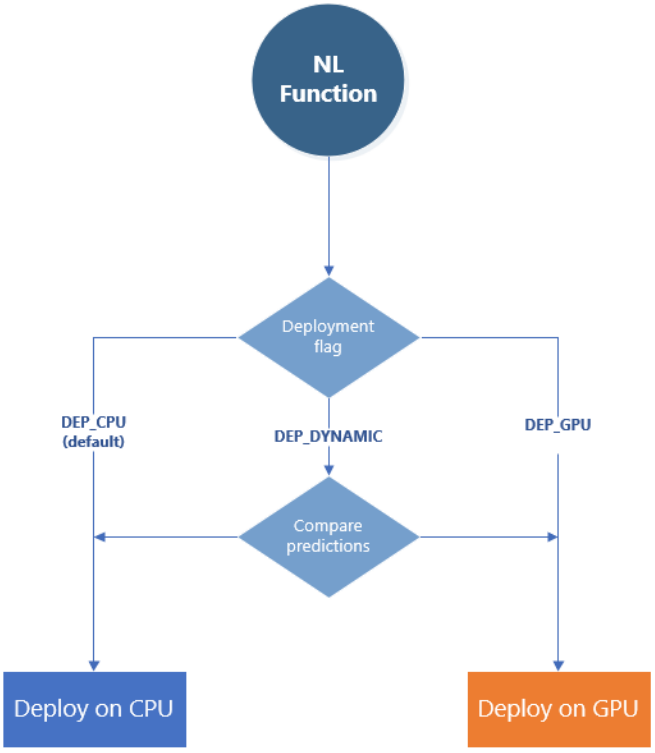



Figure 4.1: Deployment flag functionality

In addition to direct deployment as above, the `DEP_DYNAMIC` value allows dynamic deployment of the routine. With this flag value, the API predicts the input data transfer times, execution time and the output data transfer times for the two platforms and chooses the faster option. These predictions are made using the performance models that are explained in detail in Chapter 5. The

dynamic deployment feature itself is discussed in Section 4.4

The data flow analysis feature can be activated for a given part of an application code using the `dfa_start()` and `dfa_end()` functions. The API will collect profiling information on parameter sizes and the routines being called within the activated region at run time and write to a file. A data dependency graph is generated from this collected data, from which an optimized schedule can be derived. The dependency graphs and the scheduling algorithm are described in detail in Chapter 6.

### 4.3.1 GPU Deployment Approaches

When adding the GPU counterparts to the API functions, multiple implementation methods were considered. The main difference among these was how data transfers are handled through the PCI Express link. After comparing the performance of each method, one was chosen to move forward with in the study.

#### 4.3.1.1 Naive transfers

The API uses a `matrix` data structure to handle the data for the functions. The data structure contains dimension details of the matrix and a memory pointer to the data. When introducing the GPU implementation, an additional data pointer was introduced to point to the GPU copy of the data.

In this approach, all data transfers are *naively* performed for each routine. That is, first all input data are transferred to the GPU and after the execution the results are transferred back to the CPU immediately.

With this method, as all data transfers are known, a realistic prediction and comparison can be done against the CPU execution time. But the drawback of this approach is that there can be many avoidable data transfers happening for each routine. It will not try re-using data that is already available on the GPU.

#### 4.3.1.2 API managed conditional transfers

In this approach, two additional flag values are added to the `matrix` data structure (`cpu_synced` and `gpu_synced`) to keep track of synchronization status in each device. The flags are updated inside the API when the data objects get updated. For example, when a matrix is passed in for the output parameter, if the routine gets executed on the GPU, the flag values are set as `gpu_synced = 1` and `cpu_synced = 0`. So if a subsequent routine makes use of the same data object on the CPU, it needs to be transferred. However, if the that routine is executed on the GPU, no transfers are needed. Therefore, the transfers happen *conditionally*.

This method will minimize data transfers to an *on-demand* basis. That is, the outputs of routines executed on the GPU will not be transferred to the host memory until they are needed for another calculation. A disadvantage of this approach is that all data accesses need to be done through NL functions. This is because even simple reads and writes to the data arrays need to be monitored in order to update the sync flags.

There is also an impact on the predictions in the dynamic deployment mode. Transfer times for input data can be successfully predicted by considering the sync flag of each data structure. However, taking account of the output transfers in comparisons becomes difficult. For example consider Figure 4.2, where the `dgemm` routine is to be scheduled dynamically. It shows a case where the output of the routine is expected to be used in the CPU. An assumption has to be made because there is no knowledge of where the following routine will be deployed. In our study, we make the assumption

that the output of any routine will always be required subsequently on the CPU. Therefore, it will be an over-estimation for chained GPU routines, where the output of one is consumed by another.

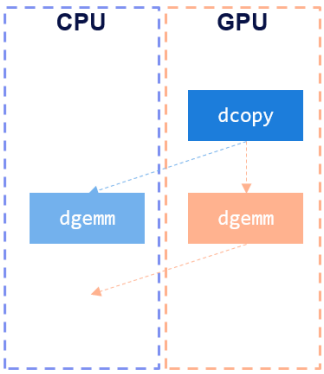



Figure 4.2: Deployment decisions with data transfers

#### 4.3.1.3 CUDA managed conditional transfers

Similar to *API managed conditional transfers*, this approach also makes data transfers only when needed, but at a more fine-grained level. The data synchronization is handled by the CUDA Unified Memory feature introduced from CUDA version 6 [59]. It is supported by Nvidia GPUs with Kepler and newer architectures. CUDA will manage a *Unified Memory* that is shared between the CPU and the GPU and can be accessed without any explicit data transfers in the code by the user (Figure 4.3). To use the Unified Memory, the memory allocations need to be done using CUDA's `cudaMallocManaged()` interface instead of the standard system `malloc()` function. Therefore, there is no need of maintaining separate data pointers for the CPU and the GPU as in the *Naive transfers* and *API managed conditional transfers* approaches. The unified memory uses a complex page fault handling mechanism to transfer data through the PCIe link on demand [60]. Thus, if only a part of a data pointer is accessed, it will transfer only the corresponding memory pages instead of the full data array as was done with *API managed conditional transfers*.

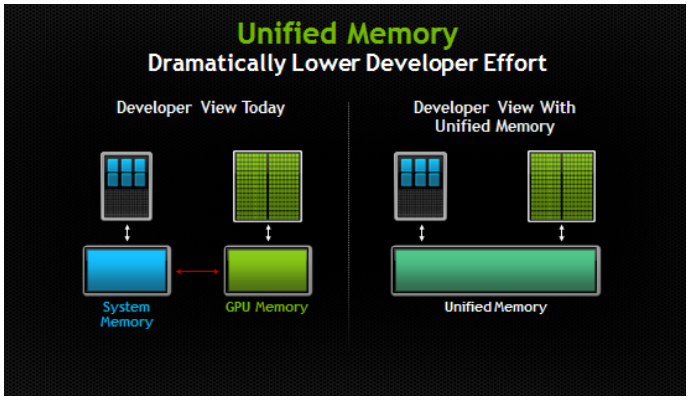



Figure 4.3: CUDA Unified Memory [59]

The main advantage of this approach is that the data synchronization does not need to be performed manually. Even if the data pointers are accessed directly, the CUDA memory management will transfer the required chunks of data automatically. According to Nvidia, in terms of bandwidth for large amounts of data, it does not perform as well as explicit memory transfers [60]. However for our application, it was observed that for frequent usage of the GPU, this method performs better than *API managed conditional transfers*.

In terms of disadvantages, in this method also we face the output transfer prediction issue. As in the previous case, the output data transfers cannot be predicted without knowledge of future calculations. In addition, even the accuracy of input transfer predictions could degrade as they are automatically handled by CUDA, whereas, explicit transfers would be used when generating the prediction models. However, in order to have some control over the predictions, the synchronization flags can be used like before to keep track of required data transfers.

#### 4.3.1.4 Comparison

In order to compare the three methods, the QP solver was executed for a range of problem sizes. The size of `H` matrix was varied from `50` to `2500` with a step size of `20`. For each size of `H`, the number of constraints was iterated from half the size of `H` to (`size_H - 1`), with the same step size. In Figure 4.4, the loop execution times of the solver is plotted for the extreme case of deploying all functions to the GPU. For reference, an all CPU instance is also included.

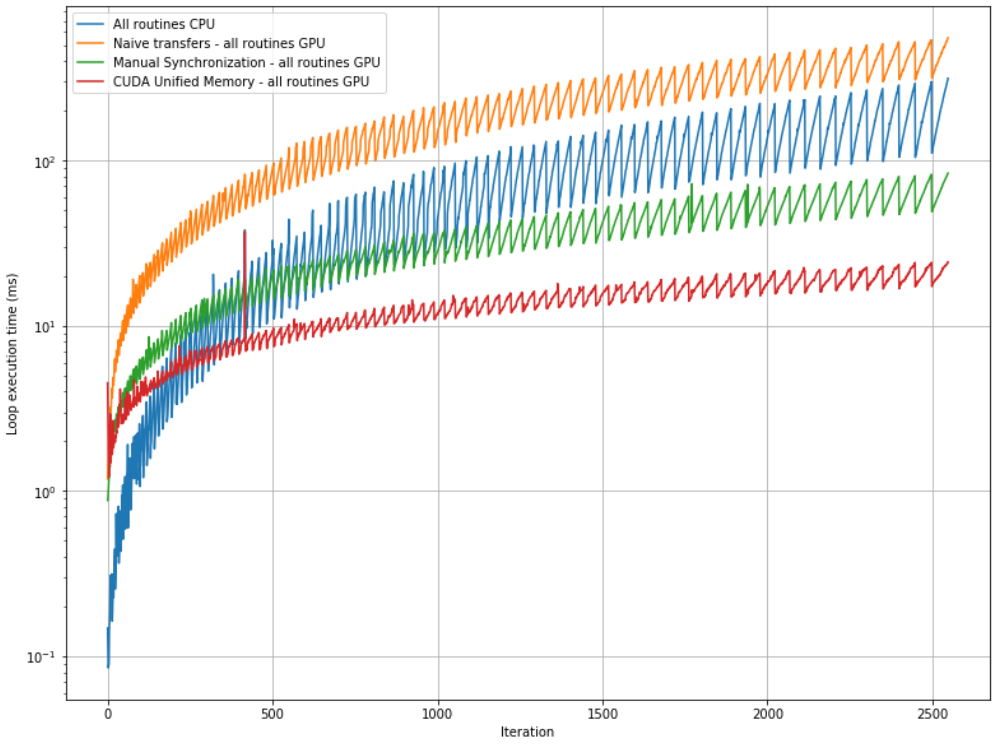



Figure 4.4: Comparison of different GPU implementations

It is clear from the figure that Unified Memory performs much better than the other two options for the application. We can also observe that the CPU performs better during around the first 250 iterations with lower sizes. This is because the time taken for the transfers to the GPU dominates the total time taken, even though there are not many transfers. Due to the better overall performance and scalability of the CUDA Unified Memory approach, it was chosen for GPU deployments in the rest of the project.

## 4.4 Dynamic Deployment

The dynamic deployment/offloading feature makes use of the performance models to predict execution and transfer times and make deployment decisions based on them. In ASML context, this may be used at pre-design time for design space exploration as a convenient method to quickly

check performance gains that can be achieved by deploying suitable functions to accelerators. In a more general use case, it may be used in a dynamic application where the data sizes change substantially at run time. For such an application, the API would try to minimize the time taken for each routine instance.

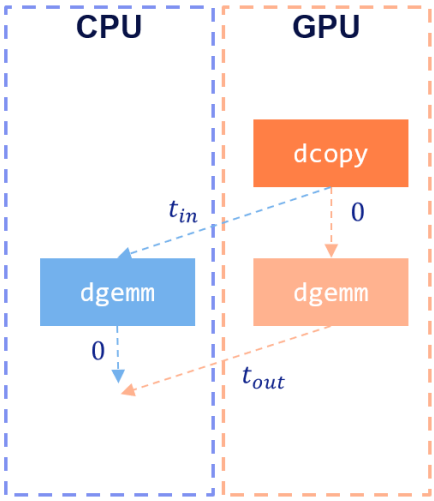



Figure 4.5: Comparison of different GPU implementations

As mentioned in Section 4.3.1, accurately predicting the transfer time becomes more complex with the CUDA Unified Memory approach. On one hand there is the issue of estimating the output data transfers due to having no knowledge of future deployments. For this, considering the CPU as the baseline, the output of routines will be assumed to be needed subsequently on the CPU, which is an overestimation for the GPU in some cases. Thus, in a scenario like in Figure 4.5 where the deployment of `dgemm` is considered, the two expressions that will be compared by the API are:

$$\begin{aligned} pred_{CPU} &= \text{(input transfer)} + \text{(CPU execution time)} + \text{(output transfer)} \\ &= t_{in} + exec_{CPU} + 0 \end{aligned}$$

and

$$\begin{aligned} pred_{GPU} &= \text{(input transfer)} + \text{(GPU execution time)} + \text{(output transfer)} \\ &= 0 + exec_{GPU} + t_{out} \end{aligned}$$

On the other hand, there is a further complication when only parts of data are accessed by routines or direct read/writes, as unified memory only transfers the relevant pages of memory between the devices. Because the API keeps track of the data using `cpu_synced` and `gpu_synced` flags, this only gives a binary view for the full data array. Thus, using the flags to determine data transfer times of the full array may result in another type of overestimation, specially when the data pointers are accessed directly. However, for routines that operate only on a part of a data array (e.g. copying a sub-matrix), the API will only use the size of the accessed elements for data transfer predictions.

Chapter 5 discusses the performance models used for predictions, while the data flow analysis mode of the API is described in Chapter 6. Finally in Chapter 7, results are presented with performance of all modes of the API.

# Chapter 5

# Performance Modeling

In this section, the performance modeling method is discussed in detail, which extends and makes use of the previous study [22] done at ASML regarding performance modeling (Section 2.3.1). That project was focused on using empirical modeling for GPU implementations of `dgemm`, `dpotrf` and `dpotrs` routines for square matrix inputs. In this project the performance models are extended in several ways:

1. Extend to more BLAS and LAPACK routines
2. Extend to modeling for the CPU
3. Extend to matrices of arbitrary sizes

Based on the usage in the example application, `dgemm`, `dgemv`, `dpotrf`, `dtrsm`, `dlacpy` and `excopy` (`mkl_domatcopy` and `cublasDgeam`) routines are selected for modeling (introduced in Section 2.1.1). For each routine, separate models are derived for the CPU and the GPU.

Extending the modeling to arbitrary sizes required re-visiting the modeling methods. With square matrices in the size range $32 \times 32$ to $10000 \times 10000$, the previous study was able to achieve relative error rates less than 20% for `dgemm`, `dpotrf` and `dpotrs`.

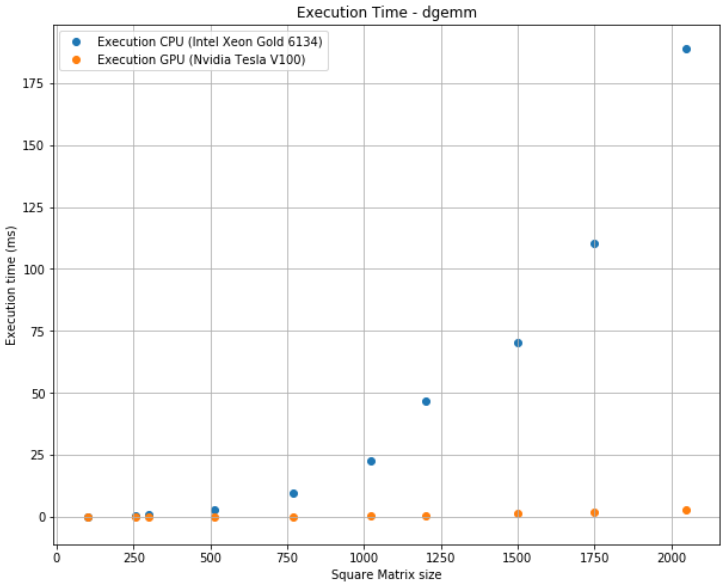



Figure 5.1: Execution time of `dgemm` for square matrices

Figure 5.1 shows the execution time measurements with square matrices for `dgemm`. The execution time is plotted against the row/column size of the square matrices. As can be observed, it shows a fairly consistent trend and it was this relationship between the matrix size and the execution time that was modeled in the previous study.

However, this method cannot be used for arbitrary sized matrices. For example, the `dgemm` routine takes in three dimension inputs `m`, `n` and `k`. These values define the sizes of the input (`m` $\times$ `k` and `k`$\times$`n`) and output matrices (`m`$\times$`n`) (Figure 5.2). Thus, they determine the amount of total operations required to calculate the output. This fact is also reflected in the computational complexity of `dgemm`, which is $O(mnk)$. For square matrices (`m = n = k`), the complexity then would be $O(m^3)$. Therefore, a single dimension value, as in the previous study, is enough to explain the execution time. But for arbitrary sized matrices, we need to take all three dimensions into account.

Thus, the same measurements of `dgemm` for square matrices are plotted against the product of dimensions (`m` $\times$ `n` $\times$ `k`) in Figure 5.3. Since it is also the computation complexity of the routine, a linear relationship can be observed now with the execution time. A similar relationship was observed for the other two routines (`dpotrf` and `dpotrs`) as well.

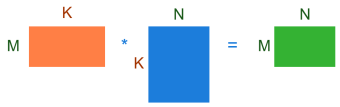



Figure 5.2: Relationship of dimension inputs to `dgemm`

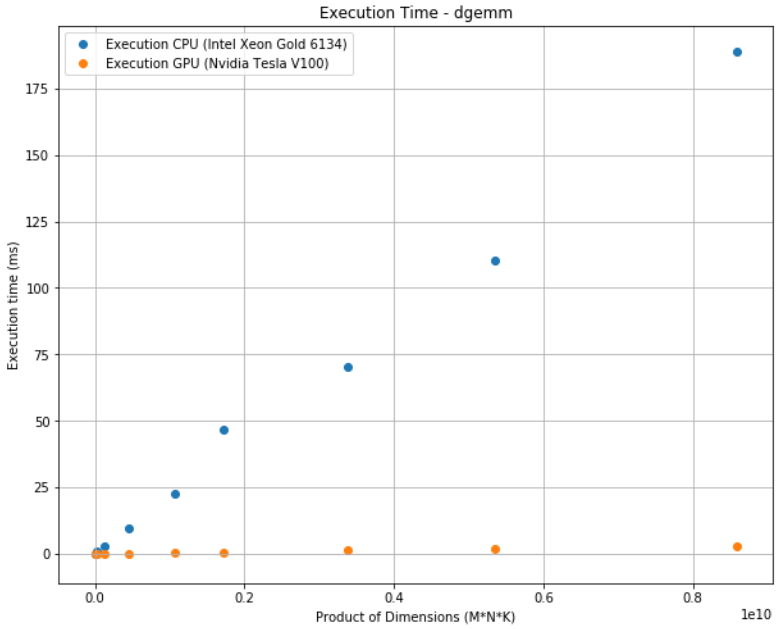



Figure 5.3: Execution time of `dgemm` for square matrices (against computation complexity)

However, for this project, the models are required to be able to predict execution times for non-square matrices as well. Therefore, for routines that accept more than one dimension argument (`dgemm`, `dgemv`, `dtrsm` and `excopy`) additional measurements were made. Initially, the aim was to observe the behavior in each routine.

First, a range of sizes was selected: powers of 2 in the range $[32, ..., 4096]$ and multiples of 500 in the range $[500, 1000, ..., 4000]$. Then for each value in the list, the computational complexity was calculated, assuming all inputs are square matrices (e.g. $O(m^3)$ for `dgemm`). Based on these com-

plexity values, all combinations of dimensions that result in the same complexity were calculated. As an example, for a size of $4$, the complexity value of `dgemm` would be $4^3 = 64$. Then the resulting combinations that yield the same complexity are:

$$(2, 2, 16), (2, 4, 8), (2, 8, 4), (2, 16, 2), (4, 2, 8), (4, 4, 4), (4, 8, 2), (8, 2, 4), (8, 4, 2), (16, 2, 2)$$

Dimensions of $1$ were ignored here so that vectors were not considered, and the matrix row/column sizes were kept below $10000$. This way, $2527$ measurements were made for `dgemm`. For the other routines which only take 2 dimension arguments, the number of combinations was as follows: For `dgemv` and `excopy` with the complexity $O(mn)$, it was $438$ measurements. The complexity of `dtrsm` is $O(m^2n)$. Consequently, it had only $135$ data points.

The main observation from the results is that for all routines, execution time changes substantially for different dimension combinations that result in the same complexity (Figure 5.4). This is the case for both CPU (MKL) and GPU (CuBLAS). As a consequence, accurate modeling of these routines becomes more complex.

It is not possible to pinpoint the exact reasons for these variations as both libraries are closed source. However, in order to gain some insight on optimized implementations of BLAS, the open source implementation of `dgemm` in the BLIS framework was referred [61]. As the performance values of BLIS are comparable with that of MKL, it is reasonable to inspect this algorithm. Generally, how such a method improves the performance over a naive triple-nested-loop implementation is by efficient use of the memory hierarchy of a CPU. The calculations are performed on small blocks of matrices that fit in the L1, L2 and L3 caches, and the re-use of fetched data is maximized. Therefore, on the whole, the number of main memory accesses is minimized, yielding better performance.

Consequently, a hypothesis for the execution time variations in routines is that the shapes of matrices affect the number of cache misses for the data. Particularly high run times are recorded on the CPU when `n < 256`, which results in "thin" matrices for the second input and the output arguments of `dgemm`. These reasons are not explored any further, as the aim of the project was to generate models using an empirical approach. Thus, some additional techniques are used to try and include the variations in the performance models.

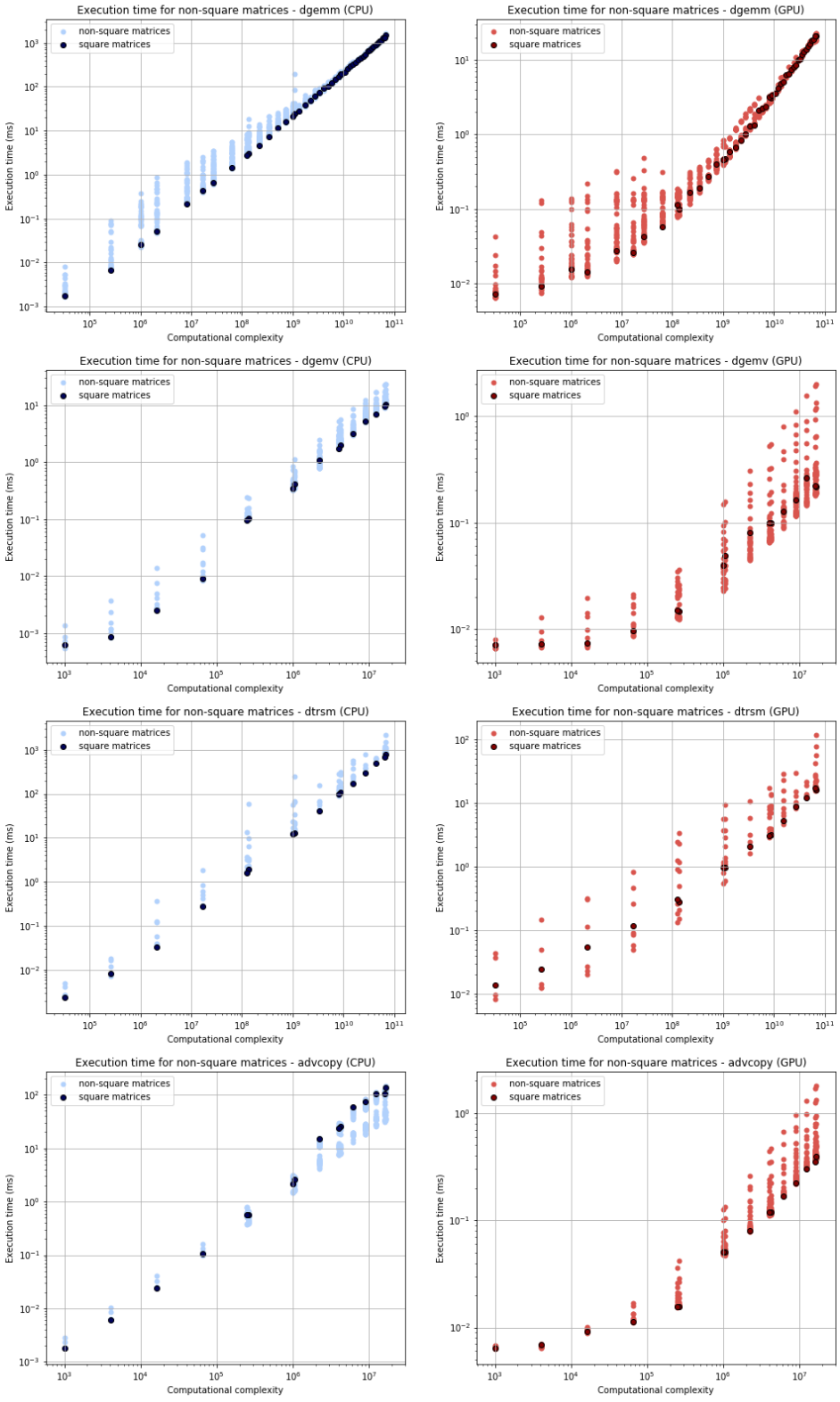



Figure 5.4: Execution time of routines with square and non-square matrices

## 5.1 Generating the Models

Primarily, we use a multiple regression method for modeling of routines with more than one dimension arguments, instead of simple regression with only computational complexity as an independent variable. This way, it is possible to include individual dimension values and their combinations in the models. A linear model derived from multiple regression can be generalized as below:

$$y = b_1 x_1 + b_2 x_2 + ... + b_n x_n + c$$

Where, $x_i$ are independent variables and $b_i$ are their corresponding coefficients. $c$ is the intercept of the polynomial.

In addition, a piecewise polynomial method similar to that in [46] and [48] is used to improve the fitting. Consequently, multiple polynomials are generated for each routine to predict for different input ranges, through which varying behaviors could be captured in the models.

A "*benchmarking*" application is developed to conveniently measure the execution time of BLAS and LAPACK routines. In the application, a high level Python (v2.7) interface acquires the routine and test details from a configuration file, which is then used to invoke the C++ program that handles executions and measurements.

The measured data is then used to generate the models using the scikit-learn [62] library for Python. The `LinearRegression` method of scikit-learn supports both simple and multiple regression. It uses a sum of squared error minimization approach to fit the linear model to the provided dataset.

$$minimize\ E = \sum_j |y_j - p(x_j)|^2$$

However, this often results in higher relative errors for smaller output sizes, which is not desirable for this project. Therefore, an additional weight is introduced in the cost (error) function to take this fact into account. The weight is defined to be the inverse of the sample values. This way, the minimized cost function becomes the squared relative error:

$$minimize\ E = \sum_j \frac{1}{y_j^2} \times |y_j - p(x_j)|^2$$

For demonstration, the modeling process of CPU `dgemv` routine is discussed in this section, while referring to other routines for comparisons. The metrics *relative error* and *percentage error* are used to measure the accuracy of the models in the rest of the report, where the percentage error is defined as $(\text{relative error} \times 100)\%$. Mostly the mean and maximum absolute percentage error values are used when assessing the models.

Firstly, we present the model derived using simple regression for non-square inputs as in the previous study [22]. The obtained polynomial can be written as:

$$y_{pred} = mn \cdot 6.530410 \times 10^{-7} - 2.406923 \times 10^{-2} \tag{5.1}$$

where, $mn$ is the complexity of the routine calculated by multiplying the two dimension arguments $m$ and $n$.

In order to check how the model performs with the full dataset, it is visualized using three types of plots as in Figure 5.10. The top-left graph shows the estimates plotted on top of the measured execution times. The predictions for the first few complexity values are not present in the plot because the values are negative and thus cannot be shown on a log scales. The model produces negative values for smaller inputs due to the negative intercept in equation 5.1. The predicted values are plotted against the expected ones in the top-right graph. Ideally, all predictions should fall on the red reference line. Finally, the relative error for each prediction is presented in the bottom graph, along with 20% references. From this, it can be seen that there are still a number of outliers in the estimates. These are quantified by two metrics: *maximum absolute percentage error*

and the *percentage of outliers* that fall outside the 20% reference. For this particular model, the two values are $55.54\%$ and $2580\%$. The mean absolute percentage error is $51.03\%$.

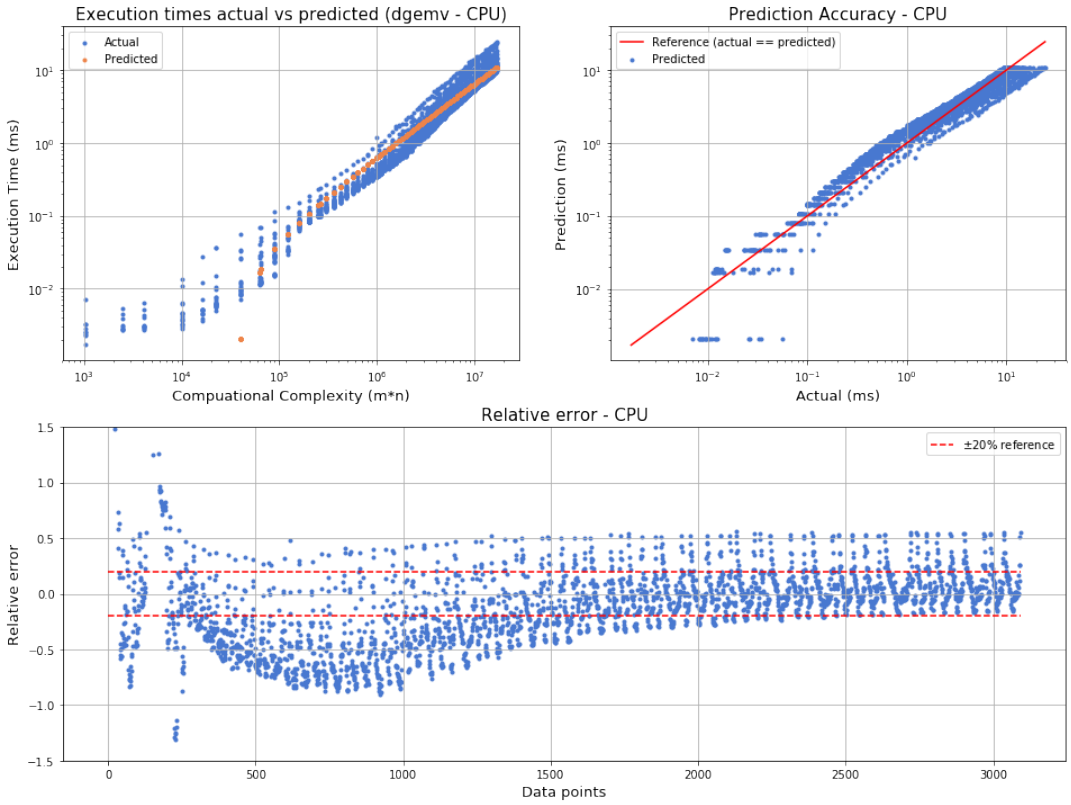



Figure 5.5: Model accuracy with simple regression for `dgemv` on GPU

In contrast to simple regression as above, the modeling approach we propose goes through multiple steps to determine suitable model parameters. As a first step, the measured execution times are plotted against the computation complexity of each routine. This helps to identify any irregularities in the execution behavior. For example, in Figure 5.6 a clear change in the pattern of measured execution times can be observed when the complexity of `dgemv` on CPU exceeds $10^5$. Based on this visualization, several input ranges are determined to test the model accuracies with.


Figure 5.6: Execution time of `dgemv` on CPU

At the same time, visualizing the execution time for a given complexity with different input combinations is useful in gaining insight on the influence of individual dimension arguments. As can be seen in Figure 5.7, the execution time of `dgemv` on CPU has increased more than 2 times compared to the square matrix case, when the matrix has dimensions $2 \times 39200$.

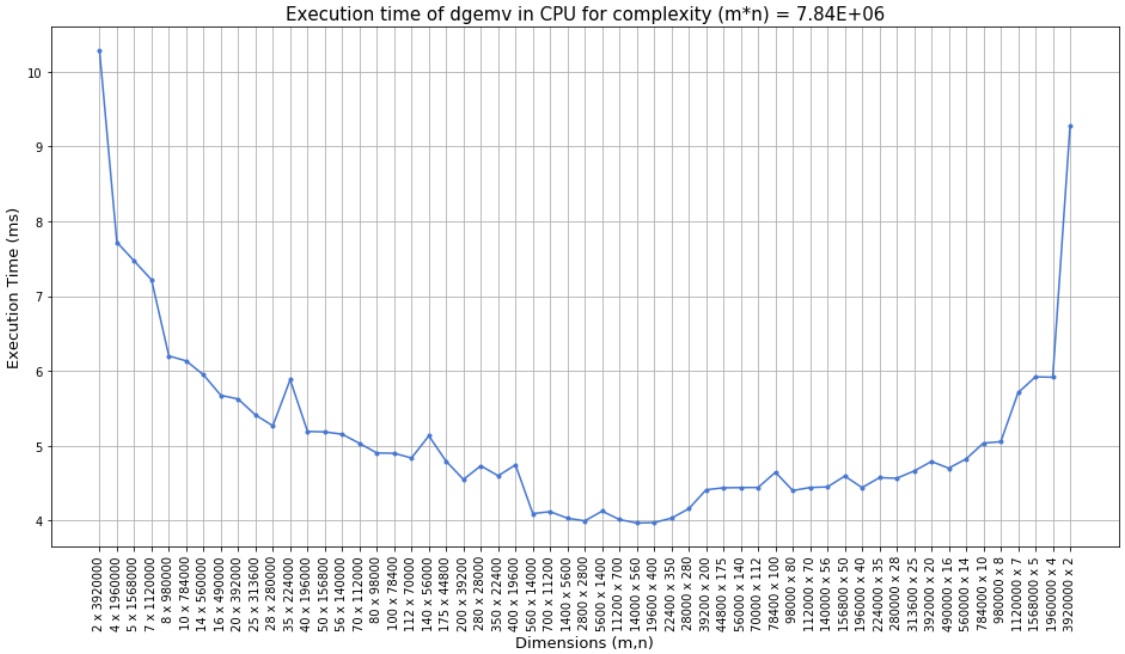



Figure 5.7: Execution time of `dgemv` on CPU for varying dimensions with same complexity

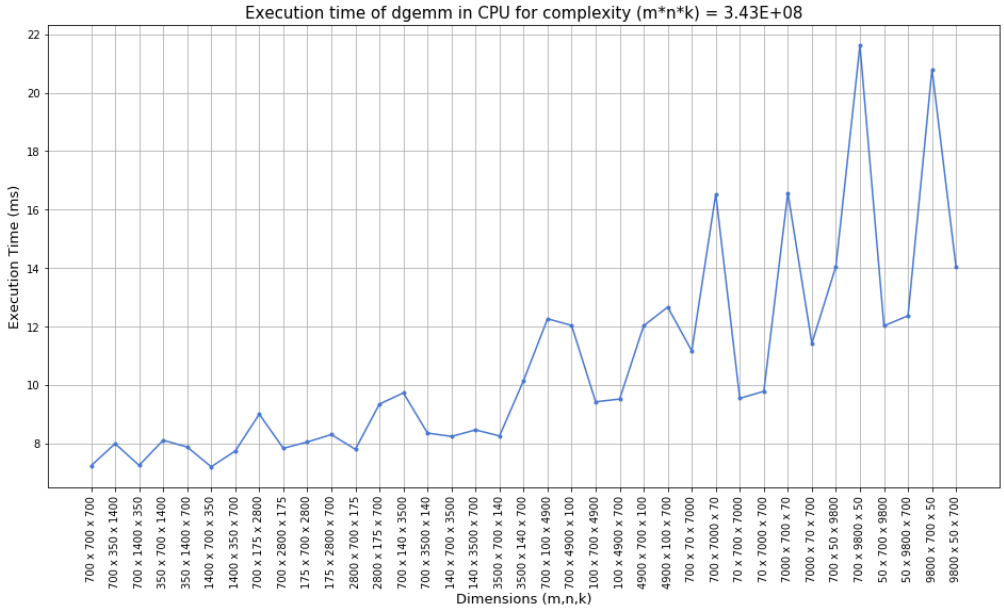



Figure 5.8: Execution time of `dgemm` on CPU for varying dimensions with same complexity

Naturally, these behaviors are different for each routine. For example, Figure 5.8 shows a similar graph for `dgemm`. There, the execution time increases substantially as the input matrices become relatively thin in shape. Furthermore, these effects were also observed to change for different input ranges even of the same routine. Thus, another (sub)division is considered within each complexity

range selected above. Dimension values that result in rather thin shapes are separated using a "*thinness ratio*" value ($r$). For `dgemv`, the selection condition is defined as:

$$(\mathtt{m} * \mathtt{r} < \mathtt{n}) || (\mathtt{n} * \mathtt{r} < \mathtt{m})$$

Subsequently, a range of values for $r$ is also selected for testing model accuracies, including value `0`, so that there is no division.

Another aspect to consider is choosing the independent variables (features) for multiple regression. First, following the approach in [46], all combinations of dimension values up to the order of complexity of the routine are selected as variables. For `dgemv`, these are $m, n$ for the first order and $m^2, n^2, mn$ for the second order. However, the number of variables increase substantially for 3 dimension arguments. For `dgemm`, it results in 19 terms as follows:

$$\begin{aligned}
\text{1st order} &: m, n, k \\
\text{2nd order} &: m^2, n^2, k^2, mn, mk, nk \\
\text{3rd order} &: m^3, n^3, k^3, m^2n, m^2k, n^2m, n^2k, k^2m, k^2n, mnk
\end{aligned}$$

If all the terms are to be used in a model, it would require around 16 multiplications and 20 additions at the least just to calculate the prediction values for `dgemm`. For fairly small input sizes with $m, n, k \leq 5$, this will be substantial in comparison to the number of operations in the calculation. In addition, some dimension combinations may contribute to the prediction in only a negligible amount. Therefore, we include an effort to identify terms that most influence the model. If we are to test the model accuracy with all possible combinations of $x$ number of variables we would have to consider $2^x$ number of combinations. In order to minimize this to a manageable amount, the variables are grouped together, so that for each routine a maximum of 4 groups are considered. This way only up to $2^4 = 16$ combinations need to be checked. In addition, with each group the complexity of the routine is always used.

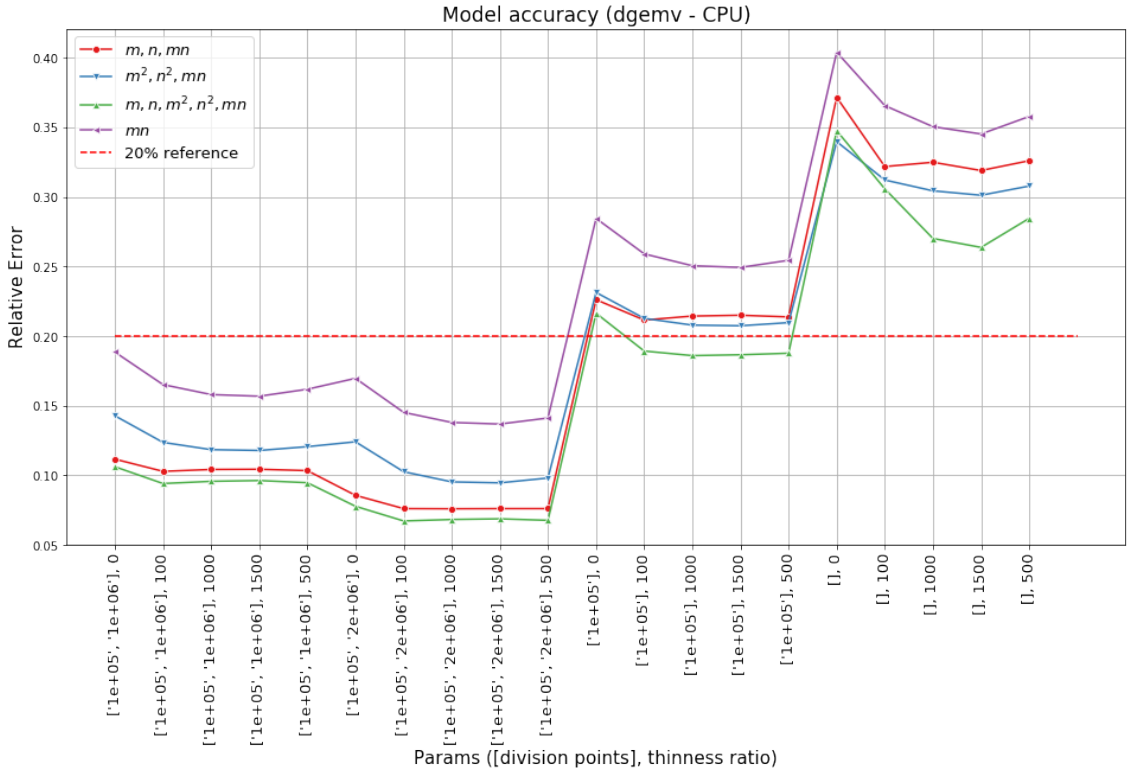



Figure 5.9: Model accuracy with different parameters for `dgemv`

Subsequently, separate models are made with the combinations of above selected feature groups

and input range divisions. Figure 5.9 demonstrates the mean absolute percentage errors of the generated models for `dgemv`. The four lines are the relative errors of models with different parameters, using the feature sets given in the legend.

The x-axis of the figure contains different parameter combinations for the model. The first parameter is a list of division points for polynomials given in the form: $[d_0, d_1, ...]$. The next parameter is the thinness ratio $r$. The blue and green curves provide the lowest relative errors in the graph, with feature sets $(m, n, mn)$ and $(m, n, m^2, n^2, mn)$. In order to reduce the number of calculations at prediction time, we choose the blue curve.

Now we focus on the left half of the graph (first 10 x-axis values), with the two division options $[1 \times 10^5, 1 \times 10^6]$ and $[1 \times 10^5, 2 \times 10^6]$. From these, the second one is chosen as it gives the lower relative errors. The effect of the thinness ratio in this region is negligible, with only a very slight drop from 0 to 100 and almost none after that. Therefore, the parameters chosen for the model are divisions $1 \times 10^5$ and $2 \times 10^6$ with no additional division for thinness ($r = 0$). The predictor variables are $m, n, mn$. The model can then be formally written as follows with the calculated coefficients:

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \cdot n + b_3 \cdot mn + b_4 & \text{if } mn < 10^5 \\ c_1 \cdot m + c_2 \cdot n + c_3 \cdot mn + c_4 & \text{if } 10^5 \leq mn < 2 \times 10^6 \\ d_1 \cdot m + d_2 \cdot n + d_3 \cdot mn + d_4 & \text{if } mn \geq 2 \times 10^6 \end{cases}$$

$$\begin{aligned} b_1 &= 3.200676 \times 10^{-7} & c_1 &= 6.711642 \times 10^{-7} & d_1 &= 1.239206 \times 10^{-6} \\ b_2 &= 2.094603 \times 10^{-6} & c_2 &= 1.085442 \times 10^{-6} & d_2 &= 1.791089 \times 10^{-6} \\ b_3 &= 1.801089 \times 10^{-7} & c_3 &= 3.652427 \times 10^{-7} & d_3 &= 6.905138 \times 10^{-7} \\ b_4 &= 1.662675 \times 10^{-3} & c_4 &= 4.548914 \times 10^{-4} & d_4 &= -8.179780 \times 10^{-1} \end{aligned}$$

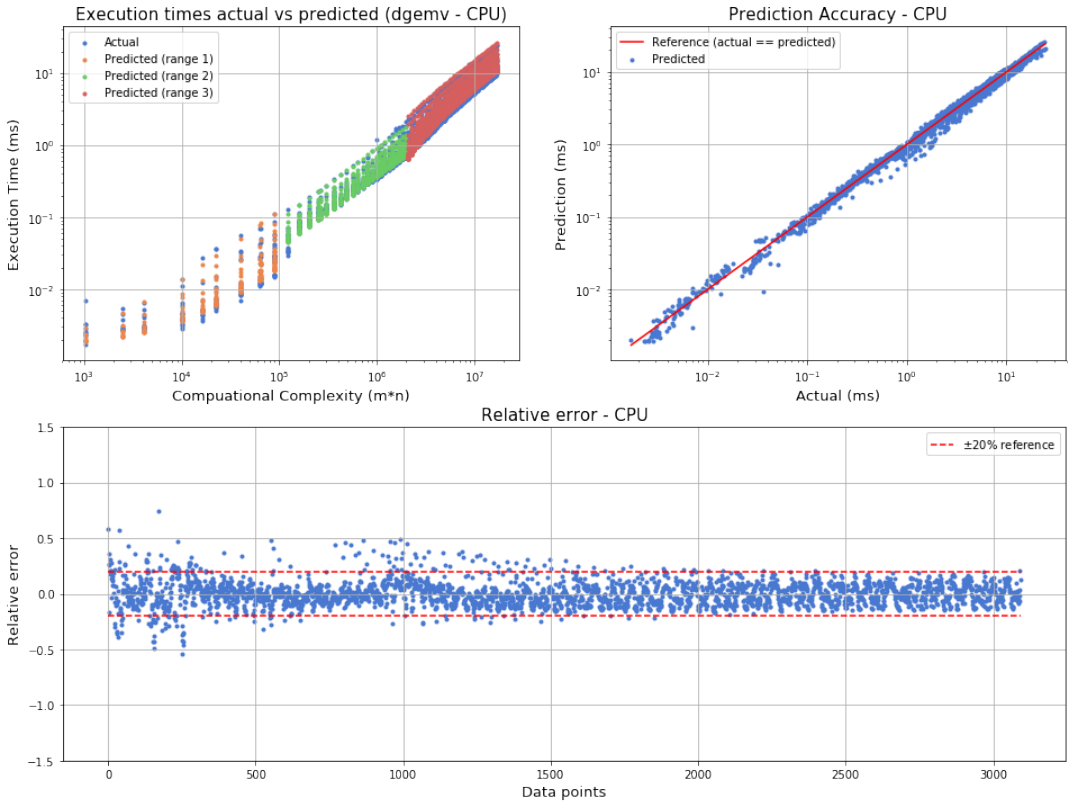



Figure 5.10: Model accuracy with the selected parameters for `dgemv` on GPU

Figure 5.10 visualizes the accuracy of this model, as was done for the simple regression model in Figure 5.5. The ranges 1, 2 and 3 in the top-left graph are predictions from the three polynomials for regions divided at $1 \times 10^5$ and $2 \times 10^6$. It is evident from this plot that this model manages to capture most of the variation introduced by non-square matrix inputs. The *maximum absolute percentage error* and the *percentage of outliers* for this model are calculated to be $74.09\%$ and $6.43\%$ respectively. In comparison, when the model that yields the minimum relative error in Figure 5.9 (green curve with divisions $1 \times 10^5, 1 \times 10^6$ and $r = 100$) is used, the maximum absolute percentage error and percentage of outliers go down to $69.95\%$ and $3.52\%$.

This proposed modeling approach was taken to generate the models of the other routines as well. Their corresponding models and related plots are provided in Appendix B. The relative errors of the models are summarized in the box plots in Figure 5.11 and Table 5.1.

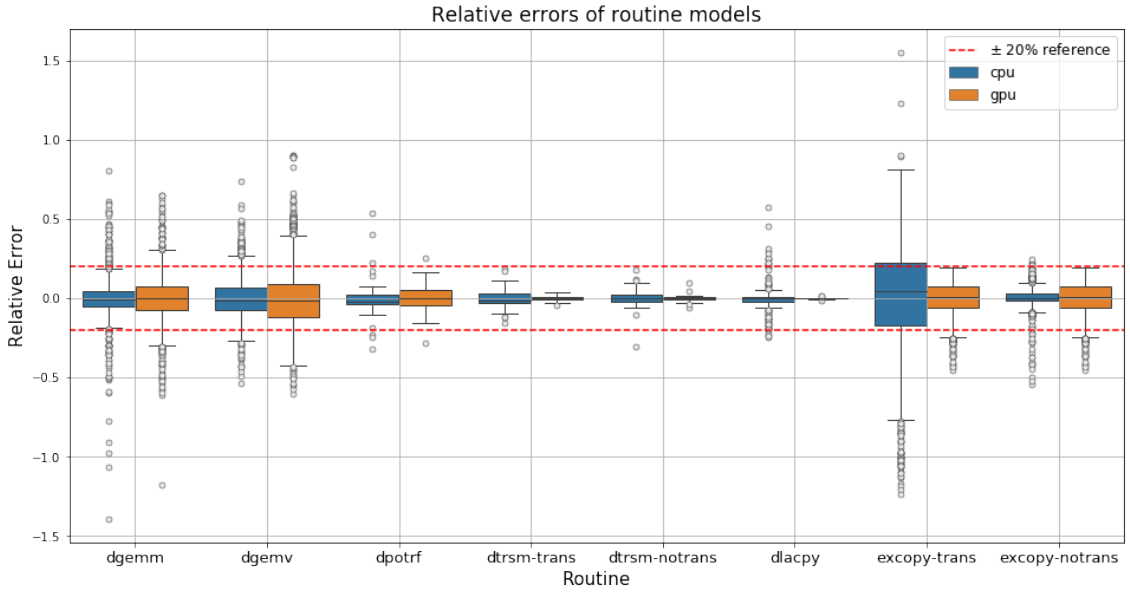



Figure 5.11: Relative error distribution of routine models

| Routine | `trans` flag | Platform | Mean absolute percentage error (%) | Maximum absolute percentage error (%) | % Outliers |
|---|---|---|---|---|---|
| `dgemm` | both `notrans` | CPU | 9.11 | 139.28 | 11.96 |
| | | GPU | 11.72 | 117.88 | 17.42 |
| `dgemv` | `trans` | CPU | 8.57 | 74.09 | 6.43 |
| | | GPU | 13.61 | 90.15 | 23.08 |
| `dpotrf` | NA | CPU | 7.90 | 53.41 | 12.50 |
| | | GPU | 6.43 | 28.29 | 4.17 |
| `dtrsm` | `notrans` | CPU | 4.28 | 30.82 | 2.08 |
| | | GPU | 1.35 | 9.69 | 0.0 |
| | `trans` | CPU | 4.76 | 18.76 | 0.00 |
| | | GPU | 1.38 | 4.56 | 0.0 |
| `dlacpy` | NA | CPU | 5.12 | 57.48 | 6.60 |
| | | GPU | 0.28 | 1.34 | 0.0 |
| `excopy` | `notrans` | CPU | 3.24 | 54.29 | 0.94 |
| | | GPU | 7.19 | 45.03 | 2.62 |
| | `trans` | CPU | 24.28 | 155.22 | 51.15 |
| | | GPU | 7.19 | 45.03 | 2.62 |

Table 5.1: Summary of model relative errors

As can be seen, there are two models each for `dtrsm` and `excopy` routines. These are based on the value of the `trans` flag, which specifies whether the input matrix should be transposed before execution. Similar to [46], this approach is used because the execution times demonstrate substantial differences for the two values. This change in behavior can be observed in the model accuracies specially for the CPU version of `excopy` (`mkl_domatcopy`). As can be seen in the Figure 16 in Appendix B, the execution time shows much variation for non square matrices when the `trans` flag is set (`trans = T`). For the GPU version of `excopy` (`cublasDgeam`) however, the same model is used as the variation is not substantial.

It is evident from the figure and the table that even though the mean percentage errors are within limits, the routines behave substantially different with arbitrary input sizes. Specially `mkl_domatcopy` still contains variations that are not captured well by the derived model. There are outliers of almost all routines falling beyond the 20% reference. Therefore, as we suggest in Section 8.1, more research can be done in terms of performance modeling to employ more sophisticated techniques to improve the accuracies. A good starting point would be to explore other machine learning and statistical techniques than multiple regression.

# Chapter 6

# Data Dependency Graphs

A method based on data dependency graphs is presented in this chapter to achieving better performance in applications by minimizing data transfer overhead. The Numerical Library API was updated to include a profiling functionality, which can be activated for a given segment of the application code. Each function call in the segment writes the parameter sizes and input dependencies to a file at runtime. The file contains the information in the form of a Julia script describing the data dependency graph. Julia language was chosen for this due to the availability of convenient off-the-shelf graph implementation libraries. In addition to the convenience, the language claims to provide performance that is comparable with C [63]. The version of Julia used in this project is 1.0.5.

The nodes of the graph represent BLAS/LAPACK routines, annotated with the predicted execution times for the CPU (`cpu_exec`) and the GPU (`gpu_exec`). The edges of the graph represent data dependencies between routines, and they are annotated with the transfer times for the corresponding data. Figure 6.1 shows a simplified dependency graph of the routines within the loop of the QP solver. Such a graph can be used to come up with a deployment schedule that minimizes the total execution time of the application. This is a Directed Acyclic Graph (DAG) scheduling problem, which is an actively researched area in heterogenous computing. Since the problem is NP complete, there are no algorithms to the best of our knowledge that achieve the optimal solution within polynomial time.

In this study, an intuitive scheduling algorithm is suggested, which is shown to yield good results for our system with only a CPU and a GPU. However, as mentioned in Section 3.3, the scope of this project does not include comparing algorithms provided in literature and implementing the best one. With the available time, this algorithm was proposed as a proof-of-concept to indicate the benefits of a graph analysis feature.

In addition to acquiring a deployment schedule, the graph can also be useful at design time for multiple other purposes:

1. To identify/visualize data dependencies and bottlenecks in a given application
2. To recognize parallelism in the application
3. To explore the impact of different problem/data sizes

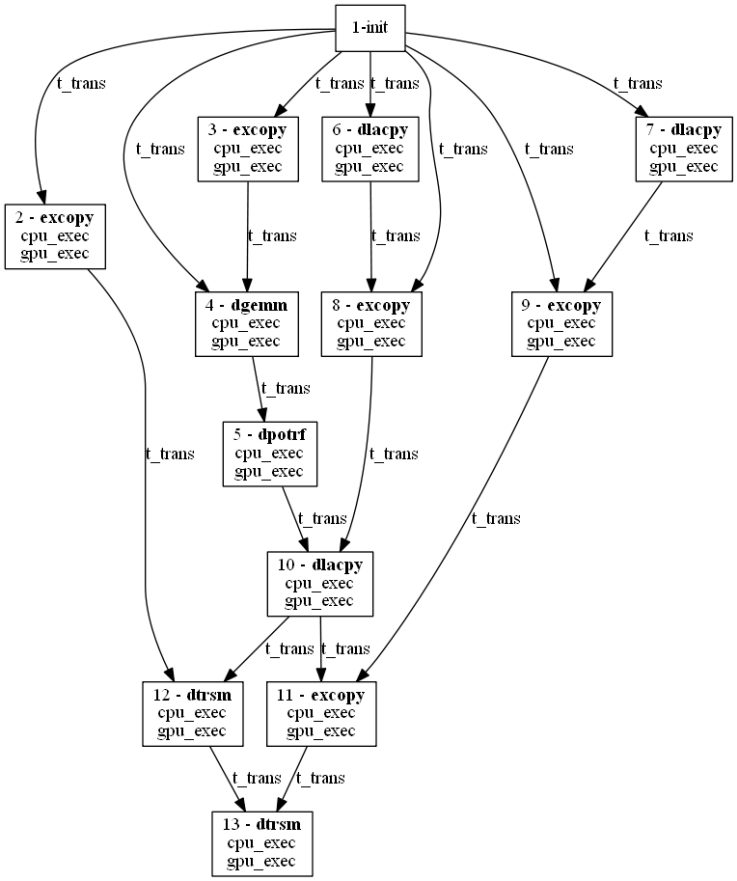



Figure 6.1: Data Dependency Graph with node and edge weights

## 6.1 Constructing the graph

A graph such as the one in Figure 6.1 has several types of parallelism. There is the computation parallelism that is relevant to the computation of each node in the graph. For example, there is parallelism in the matrix-matrix multiplication, which is exploited by the GPU implementation of `dgemm`. At the same time, there is a parallelism inherent in the graph. This is what is represented in the graph by *parallel* nodes that do not have any dependency among each other (e.g. nodes $2, 3, 5$ and $6$). In order to exploit this parallelism there needs to be multiple compute units such as combinations of multiple cores in the same CPU, multiple CPUs and/or multiple GPUs etc. Finally, there is a parallelism involved with executions and data transfers, where some data transfers may happen in the background while routines are being executed.

In this project, only computation parallelism is exploited for the sake of simplifying the scheduling algorithm. Thus, graph parallelism is eliminated from the graph by adding extra edges between subsequent nodes. These edges can be annotated with a transfer weight of $0$ as they do not represent data dependencies. After the transformation, the dependency graph also reflects the order of execution in the program as it would happen on a single processor thread. Figure 6.2 shows the same graph as before, but with the parallelism removed in this way. Additional edges have been added for example from nodes $2$ to $3$ and $5$ to $6$. The above process will be referred to as "*sequentializing*" in this report for convenience.

The execution-transfer parallelism is ignored by the algorithm as well, which is again for the sake of simplicity. However, despite these limitations, the algorithm is shown to yield considerable improvements over dynamic deployment for the QP solver application.

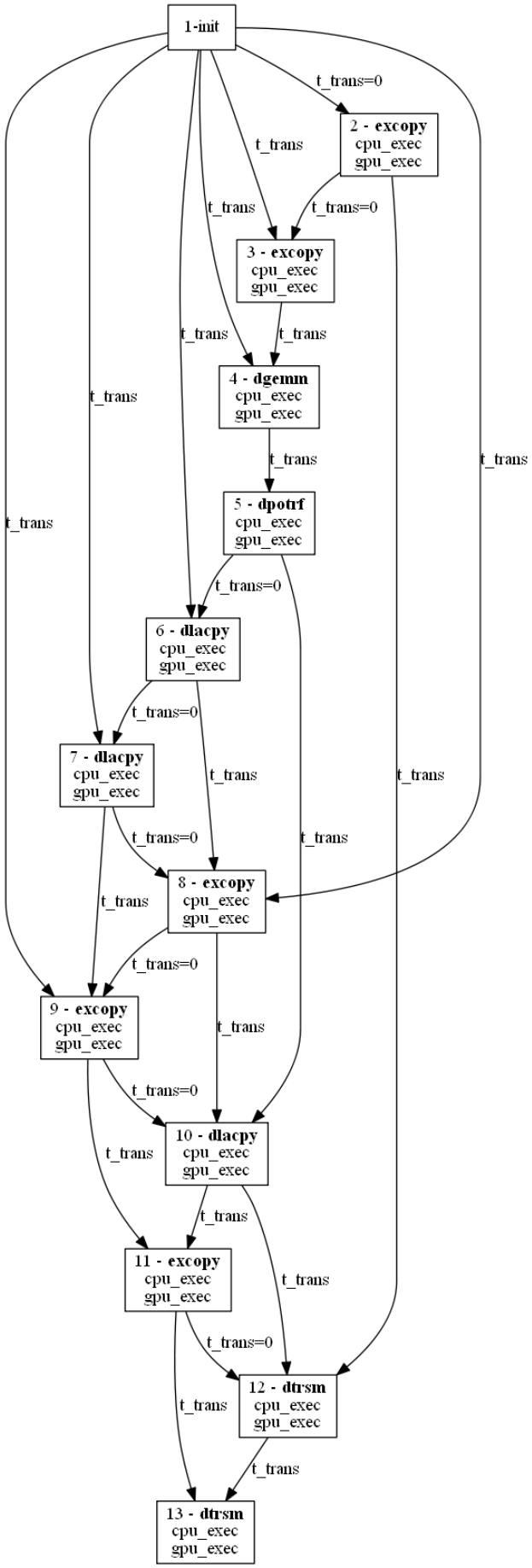



Figure 6.2: Dependency graph after removing graph parallelism

## 6.2 Finding the optimal schedule

Consider a hypothetical dependency graph as in Figure 6.3 which is then sequentialized. For demonstration, each node is annotated with a CPU execution time of 4 time units and GPU execution time of 2 time units. All data transfers between the CPU and GPU are assumed to take 1 time unit. The $init$ and $end$ nodes are additionally added when creating the graph. The $init$ node is simply there to enforce the beginning to be on the CPU and also as a source node for any data that are not generated by nodes within the graph. Similarly, the $end$ node forces the program segment to finish in the CPU and acts as a sink to the output data of the final calculation.

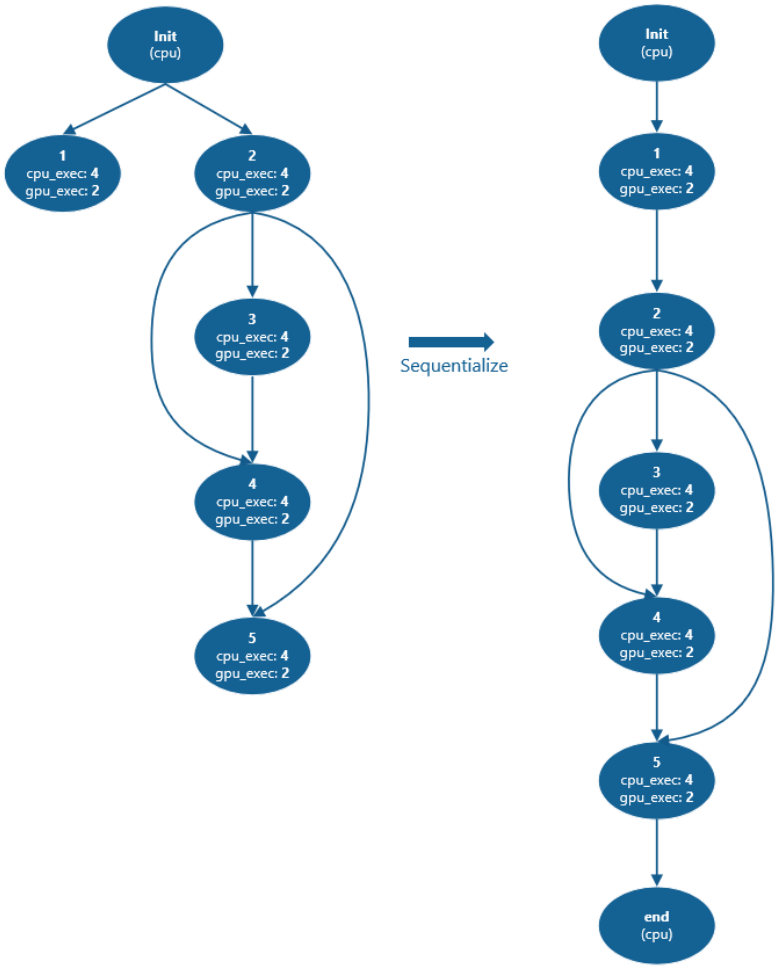



Figure 6.3: Graph sequentializing

The above graph can then be extended to represent the CPU and GPU deployment of each node separately. This is done by duplicating all nodes in the graph, except $init$ and $end$ (Figure 6.4). In addition, there may also be other nodes that are fixed to a specific platform by the user, which are also not duplicated. In the figure, the nodes are appended with a letter "$c$" or "$g$" to mean CPU or GPU deployment respectively. The effective data transfers are annotated on edges. For example, they become valid only when the data needs to be moved from one platform to the other, which is equivalent to diagonal edges in the graph. Also note that the diagonal edges with transfer value of 0 are dummy edges that were added for sequentializing, and thus no data is transferred between the nodes. The $init$ and $end$ nodes are simply "fixed" on the CPU and therefore transfer cost becomes effective for edges that are only to/from a GPU node. The lists of values beside each node is explained in the next section.

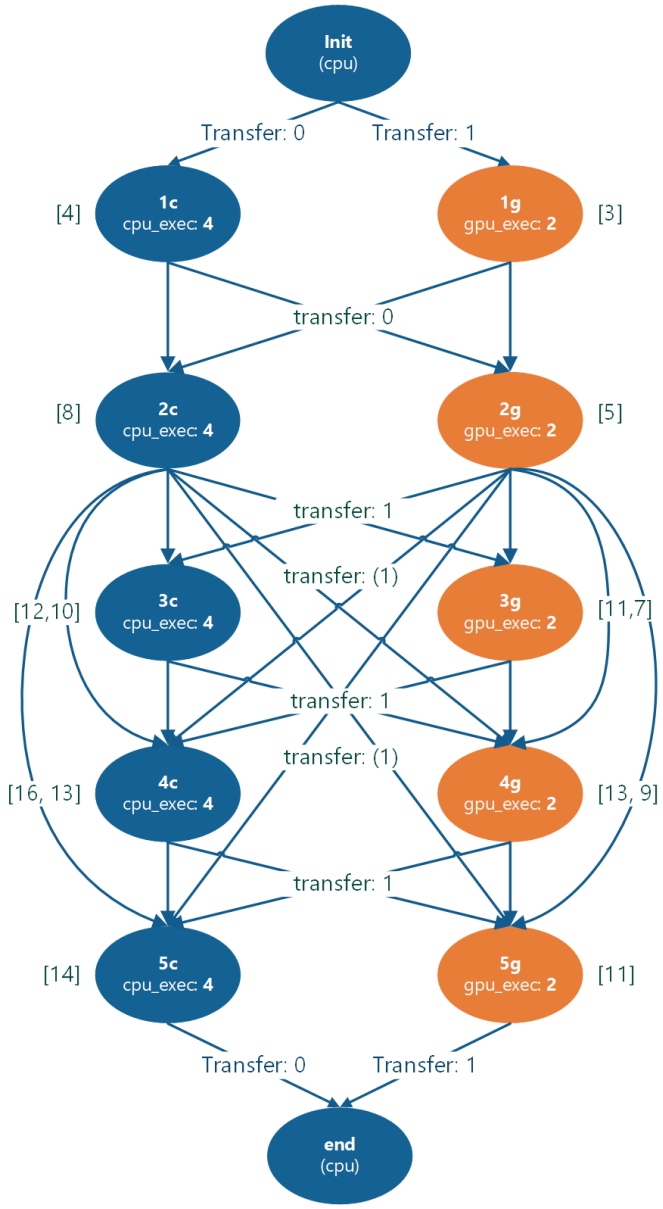



Figure 6.4: Extending the graph for separate representation of deployments

Finding the optimal deployment schedule for the routines now boils down to finding the least cost path from $init$ node to the $end$ node with the constraint of visiting either the CPU or the GPU instance of each routine. Due to this constraint, conventional shortest path algorithms cannot be utilized to create a deployment schedule. For instance, when there are additional edges in the graph such as the ones from 2 to 5 in Figure 6.4, a shortest path algorithm may pick these paths, ignoring the cost of going through 3 and 4.

An important point to note when finding the path is that some transfer values on the edges may become ineffective in certain schedules. For example, if the path is taken as $2c \rightarrow 3g \rightarrow 4g$, the data transfer of edge $2c \rightarrow 4g$ becomes redundant. The reason is that the same data would be transferred previously to the GPU when taking the $2c \rightarrow 3g$ edge. Such transfer values are

annotated in brackets in the figure.

### 6.2.1 Optimizing algorithm

The algorithm works incrementally through the graph from the $init$ node to the $end$ node. First consider node $1$. The time to obtain the output of the calculation for the CPU and the GPU cases are $1c_{o1} = 0 + 4 = 4$ and $1g_{o1} = 1 + 2 = 3$ respectively. The notation used here is to mean time to output ($o$) of the $x^{th}$ path of the corresponding node ($1c_{ox}$ or $1g_{ox}$). These values are annotated beside the $1c$ and $1g$ nodes within square brackets in Figure 6.4.

Once the two paths are calculated for $1$, all output time possibilities for node $2$ are calculated based on them:

$$
\begin{aligned}
2c_{o1}(1c \rightarrow 2c) &: 1c_{o1} + e_{2c} &&= 4 + 4 &&= 8 \\
2g_{o1}(1c \rightarrow 2g) &: 1c_{o1} + t_{1c \rightarrow 2g} + e_{2g} &&= 4 + 0 + 2 &&= 6 \\
2c_{o2}(1g \rightarrow 2c) &: 1g_{o1} + t_{1g \rightarrow 2c} + e_{2c} &&= 3 + 0 + 4 &&= 7 \\
2g_{o2}(1g \rightarrow 2g) &: 1g_{o1} + e_{2g} &&= 3 + 2 &&= 5
\end{aligned}
$$

Where,
$e_x$: Execution time associated with a node $x$
$t_{x \rightarrow y}$: Transfer time associated with an edge $x \rightarrow y$

The deployment possibilities (paths) now can be visualized as a binary tree, as shown in Figure 6.5. The time to the end of execution of each node is annotated beside them as calculated above. The information of this tree can be represented using two lists. One that contains the time values at the leaf level (`[8, 6, 7, 5]`) and another which maintains the list of "*open*" nodes (`[1,2]`). The term *open* is used here to mean the nodes of which all deployment possibilities are kept open at a given point in the algorithm. In addition to those two lists, another data structure needs to keep track of the deployment assigned to each node in each path.

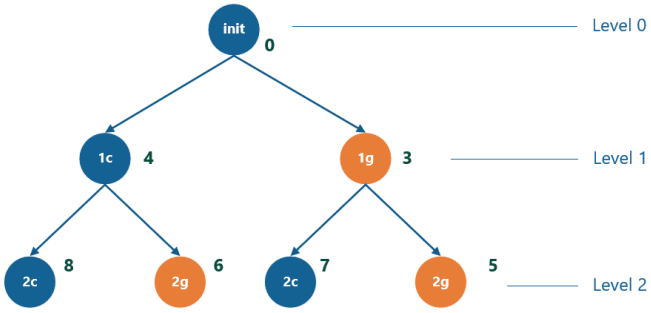



Figure 6.5: Deployment possibilities up to node 3 represented as a binary tree

After the values for $2$ are calculated, we can minimize the the number of possibilities, given the fact that node $1$ has no influence on any future nodes (no additional edges from node $1$ going past the current node $2$). This is done by choosing the paths with minimum time to reach $2c$ and $2g$ nodes, while considering deployment of $1$ as "do not care". That is, we choose one path from the pair $[(1c \rightarrow 2c), (1g \rightarrow 2c)]$ and another path from $[(1c \rightarrow 2g), (1g \rightarrow 2g)]$.

$$2c_{min} = min(2c_{o1}, 2c_{o2}) = 2c_{o1}(1c \rightarrow 2c) = 7$$
$$2g_{min} = min(2g_{o1}, 2g_{o2}) = 2g_{o2}(1g \rightarrow 2g) = 5$$

In terms of the binary tree, this is equivalent to eliminating the level 1 nodes and having only 2 nodes $2c$ and $2g$ (Figure 6.6). This elimination will be referred to as "*pruning*" from this point onwards in the report. Given the list of leaf time values `tlist`, pruning of the `n`-th level of the tree can be done using a recursive algorithm as shown in Algorithm 2.

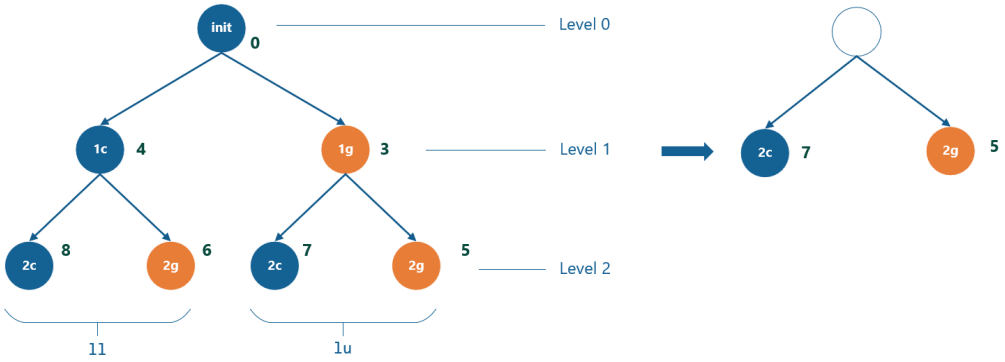



Figure 6.6: Pruning level 1 of the binary tree

**Algorithm 2:** Prune level in binary tree

```
1  Function prune_level(tlist, n, curr_level):
       /* tlist:  The list of time values at the leaf level          */
       /* n:  Level to be pruned                                     */
       /* curr_level:  Current level.  At start, value 1 is passed.  */
2
3      if length(tlist) == 1 then
4          return tlist
5      else
           // get lower and upper halves of tlist
6          ll, lu = divide_list(tlist)
7
8          if curr_level == n then
               // get the element-wise minimum list from the two halves
9              new_tlist = elem_min(ll, lu)
10             return new_tlist
11         end
12
           // recursively call for the next level
13         ll = prune_level(ll, n, curr_level+1)
14         lu = prune_level(lu, n, curr_level+1)
15         return concatenate(ll, lu);
16     end
```

Element-wise minimum in this context is defined as follows. Given two lists

$$l1 = [x_1, x_2, ..., x_n] \; and$$
$$l2 = [y_1, y_2, ..., y_n] \; ,$$

the element-wise minimum will be a new list $lmin$ of size $n$:

$$lmin = [min(x_1, y_1), min(x_2, y_2), ..., min(x_n, y_n)]$$

It is visualized in Figure 6.6, where the element-wise minimum values of two lists `ll=[8,6]` and `lu=[7,5]` have been chosen as the new leaf values (`[min(8,7), min(6,5)] = [7,5]`). Note that it is these minimized `tlist` values that are annotated beside each node in Figure 6.4.

Once the pruning is done for node 1, only 2 will remain as an open node. Next, the `tlist` can be calculated for node 3. At this point, we do not minimize 2 as it affects future nodes 4 and 5.

$$
\begin{aligned}
3c_{o1}(2c \rightarrow 3c) &: 2c_{o1} + e_{3c} &&= 7 + 4 &&= 11 \\
3g_{o1}(2c \rightarrow 3g) &: 2c_{o1} + t_{2c\rightarrow 3g} + e_{3g} &&= 7 + 1 + 2 &&= 10 \\
3c_{o2}(2g \rightarrow 3c) &: 2g_{o1} + t_{2g\rightarrow 3c} + e_{3c} &&= 5 + 1 + 4 &&= 10 \\
3g_{o2}(2g \rightarrow 3g) &: 2g_{o1} + e_{3g} &&= 5 + 2 &&= 7
\end{aligned}
$$

When calculating the leaf node values for node 4, previously mentioned redundant edge costs come into effect. For example, when we consider the path $2c \rightarrow 3c \rightarrow 4g$, the transfer cost of edge $2c \rightarrow 4g$ needs to be considered, as the data would have to be transferred to the GPU. However, if the path is $2c \rightarrow 3g \rightarrow 4g$, the data would have already been transferred to the GPU before execution of node 3. Thus, the cost of edge $2c \rightarrow 4g$ can be neglected.

$$
\begin{aligned}
4c_{o1}(2c \rightarrow 3c \rightarrow 4c) &: 3c_{o1} + e_{4c} &&= 11 + 4 &&= 15 \\
4g_{o1}(2c \rightarrow 3c \rightarrow 4g) &: 3c_{o1} + t_{2c\rightarrow 4g} + t_{3c\rightarrow 4g} + e_{4g} &&= 11 + 1 + 1 + 2 &&= 15 \\
4c_{o2}(2c \rightarrow 3g \rightarrow 4c) &: 3g_{o1} + t_{3g\rightarrow 4c} + e_{4c} &&= 10 + 1 + 4 &&= 15 \\
4g_{o2}(2c \rightarrow 3g \rightarrow 4g) &: 3g_{o1} + e_{4g} &&= 10 + 2 &&= 12 \\
\\
4c_{o3}(2g \rightarrow 3c \rightarrow 4c) &: 3c_{o2} + e_{4c} &&= 10 + 4 &&= 14 \\
4g_{o3}(2g \rightarrow 3c \rightarrow 4g) &: 3c_{o2} + t_{3c\rightarrow 4g} + e_{4g} &&= 10 + 1 + 2 &&= 13 \\
4c_{o4}(2g \rightarrow 3g \rightarrow 4c) &: 3g_{o2} + t_{2g\rightarrow 4c} + t_{3g\rightarrow 4c} + e_{4c} &&= 7 + 1 + 1 + 4 &&= 13 \\
4g_{o4}(2g \rightarrow 3g \rightarrow 4g) &: 3g_{o2} + e_{4g} &&= 7 + 2 &&= 9
\end{aligned}
$$

At this point, we are able to prune level 2 of the tree, as node 3 does not affect any further nodes. However, node 2 is still kept open. When pruning the second level, sub trees of node 3 are used in comparison. That is, the element-wise minimum values of ($ll_1$, $lu_1$) and ($ll_2$, $lu_2$) pairs are taken for the leaf nodes.

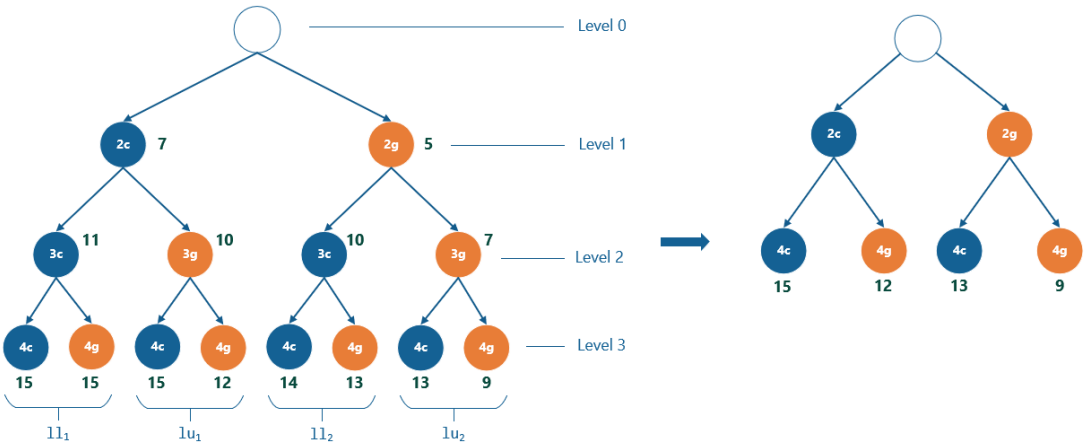



Figure 6.7: Purning level 3 of the binary tree

Once we calculate the leaf values for node 5, we are able to prune all previous levels in the tree.

With the above algorithm, the pruning will be done in two stages. First level 1 with node 2 options is pruned. In the resulting tree, the new level 1 contains options for node 4. This level is pruned in the next step (Figure 6.8).

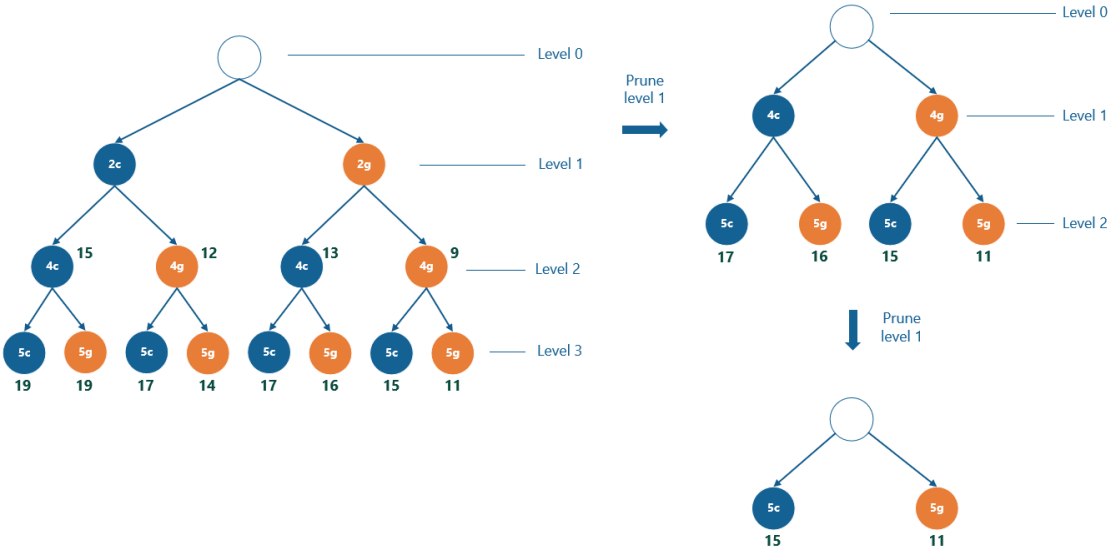



Figure 6.8: Pruning in two stages at node 5

Finally, at the $end$ node, one of the two resulting paths will be chosen. As the $end$ node is fixed to the CPU, comparisons are made between the path through $5c$ with 15 time units and the path through $5g$ with $11 + 1 = 12$ time units with the data transfer from GPU. So the chosen path that takes 12 time units is:

$$1g \rightarrow 2g \rightarrow 3g \rightarrow 4g \rightarrow 5g$$

It is to be noted that this also represents a special case where a node in the middle of the graph might have the deployment fixed to the CPU or the GPU. The comparisons would be similar to the above case when we are calculating the options for that particular node. Another special case is when doing the calculations for the node after a such a fixed node. Then it is similar to that of node 1 above, where the $init$ node was fixed on the CPU.

Limitations of this algorithm are discussed in detail in Section 7.2.1.

# Chapter 7

# Results and Analysis

In order to test the performance of the API, the QP solver was executed with two problem sizes (Table 7.1). It can be seen in the solver algorithm (Algorithm 1) that the data is mostly generated and used within the loop itself. This can be exploited when most of the routines are executed on a single platform to minimize data transfers. Test case 1 was chosen to observe how the API manages to capture that, in a large problem size where it is beneficial to deploy all routines on the GPU. Test case 2 is a smaller one where for an initial number of iterations the CPU performs better than the GPU and vice versa for later iterations. Thus, ideally the API should deploy the routines in the initial iterations on the CPU and the rest on the GPU.

| | $n$ | $m$ |
|---|---|---|
| Test case 1 | 1500 | 2000 |
| Test case 2 | 600 | 800 |

Table 7.1: QP solver test sizes

For each problem size, the loop was executed by continuously adding constraints, so that the execution time grows over time.

## 7.1 Test Case 1

Initially, the test case was executed in two runs, where all the routines were either deployed on the CPU or the GPU. The solver was executed for multiple iterations with the number of constraints in the system (`nidxs`) increasing at a step size of 50. The run time of each loop iteration was measured and was plotted as in Figure 7.1.

It can be seen that the CPU only and GPU only cases yield substantially different execution times for this problem size. The reason is as stated above, the data is used mostly within the loop itself, in which case hardly any data transfers are needed when all routines are on the same platform. Here, it is possible to see that the first iteration has taken longer than the second for the all GPU case. This is due to the initial transfer of the large $H$, $Bt$ and $A$ matrices in Algorithm 1, which are then re-used internally in the subsequent iterations.

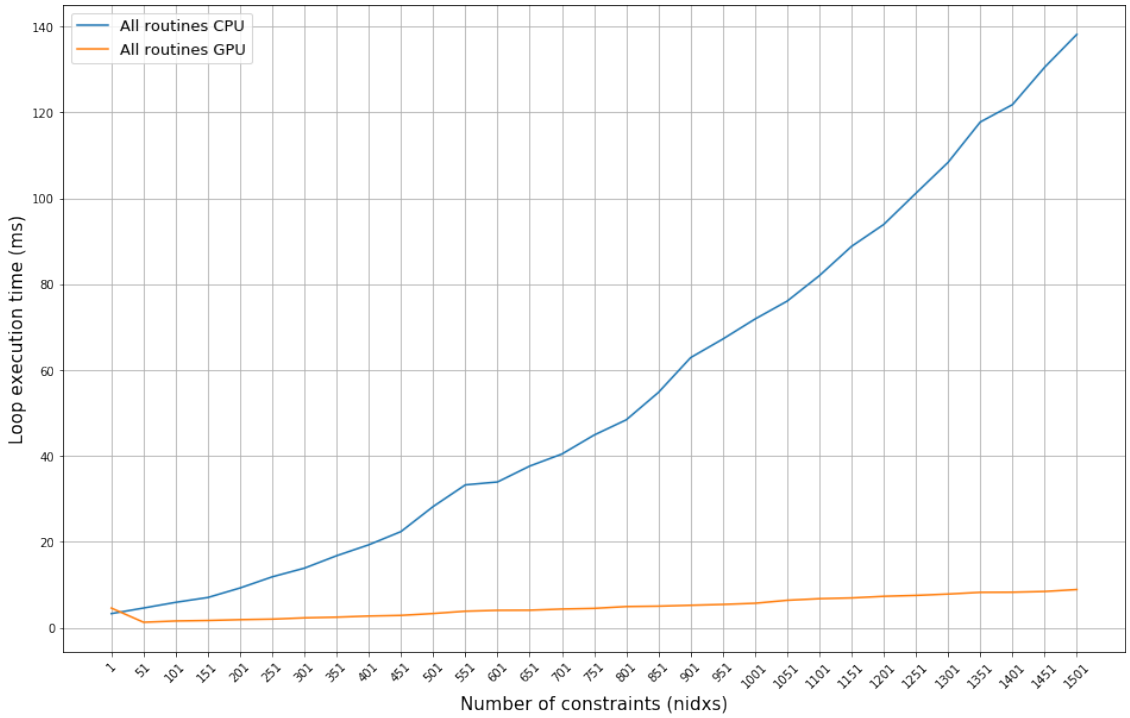



Figure 7.1: Loop execution time of QP solver with all routines in CPU and GPU

### 7.1.1 Performance Model Accuracies

The predicted and actual execution times of each routine were logged during above runs. Figures 7.2 to 7.5 demonstrate the predicted and measured execution times for `dgemm`, `dgemv`, `dpotrf` and `dtrsm` routines. The measured points were sorted according to execution time value and plotted along with the $\pm 20\%$ margins. On each point, the corresponding prediction value is presented as an error bar, which ideally should fall within the $\pm 20\%$ region.

An initial observation with `dtrsm` is that even for the same dimensions, the execution time differed substantially when the coefficient matrix is transposed. The transposed version is used in each iteration for backward substitution, where the $U$ matrix is constructed as lower triangular and is passed to `dtrsm` with the `trans` flag set. Due to this, two models were generated for `dtrsm` with additional measurements for transposed and non transposed cases. The predictions seen in Figure 7.5 are from these two models, which are chosen at run time based on the `trans` argument value. Similarly, two separate models based on transposition were made for `mkl_domatcopy` as well. The accuracies of these models are presented in Section 5.1, with detailed measurements and models provided in Appendix B.

In the measurements for `dgemv`, even though the input sizes are the same for all iterations, there are a number of variations observed in the GPU execution time. After sorting, these appear toward the right side of the GPU graph in Figure 7.3 (test point 13 onwards). It is non trivial to explore the exact reasons for such variations as the library is closed source and the variations are not regularly present in each run. Hence, this is not attempted in the scope of this project. Our assumption is that it is due to the influence of other GPU activity.

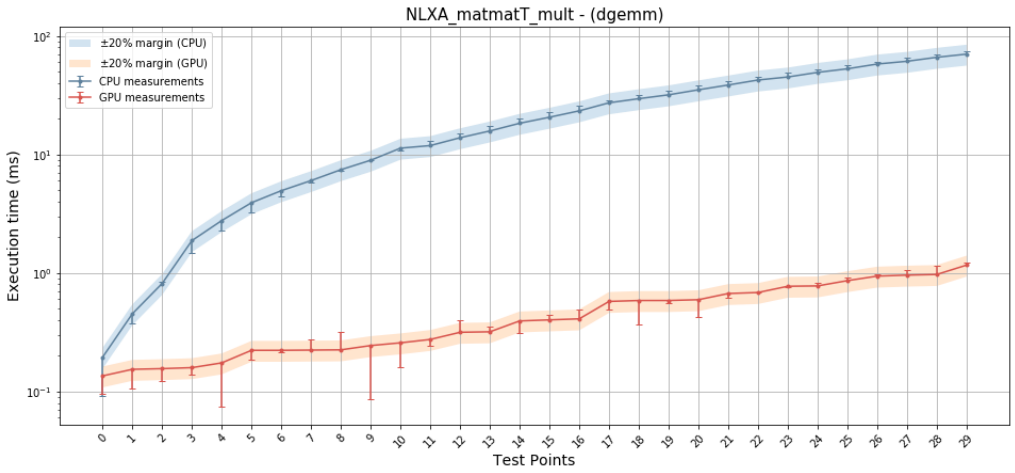



Figure 7.2: Actual and predicted execution times `dgemm`

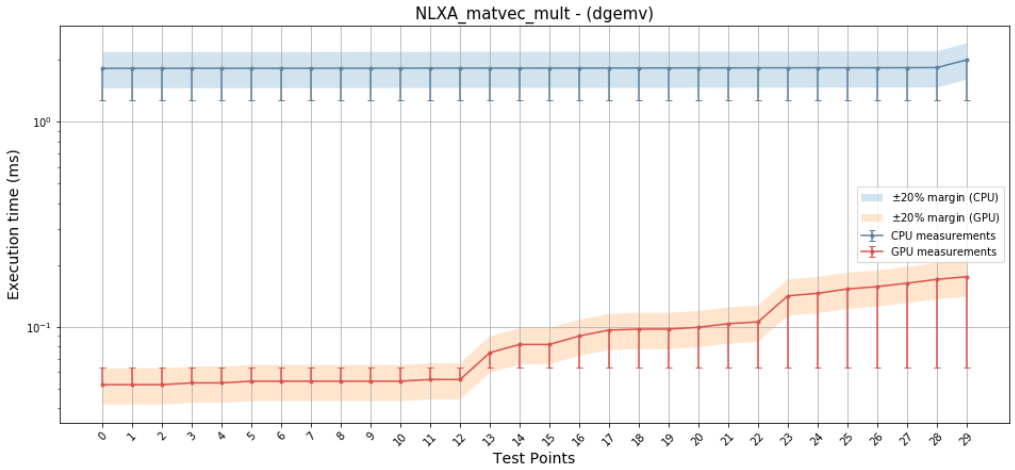



Figure 7.3: Actual and predicted execution times of `dgemv`

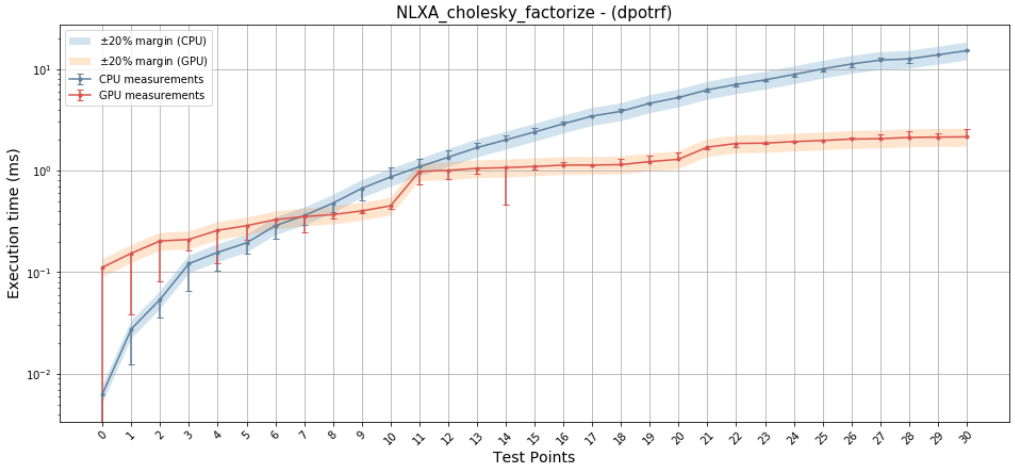



Figure 7.4: Actual and predicted execution times of `dpotrf`

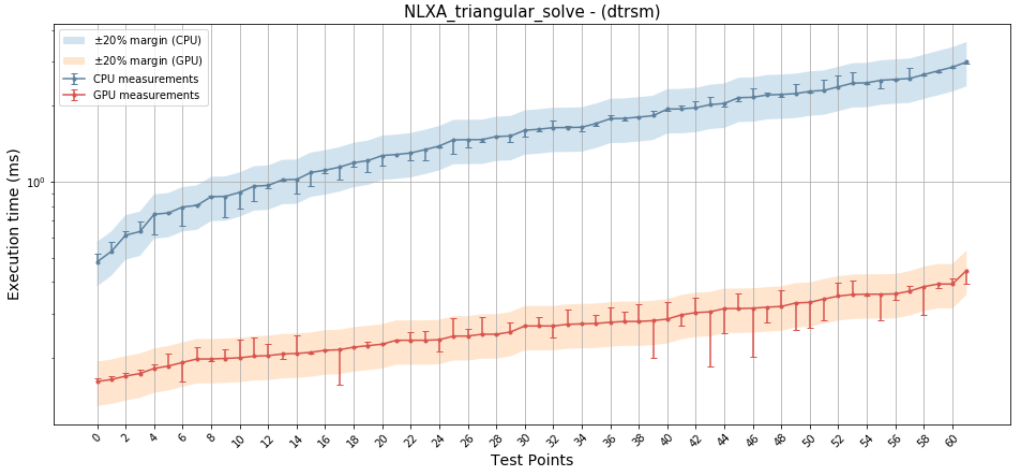



Figure 7.5: Actual and predicted execution times of `dtrsm`

Figures 7.6 and 7.7 show the actual and measured times for the matrix copy routines. Compared to the previous 4 routines, the CPU versions of both these copy functions demonstrate larger variations throughout the measured ranges. The reason is the effect of caching. This is most evident with first usage of matrices that get created inside the loop in the QP solver loop (Algorithm 1). For example, the $L$ and $U$ matrices are created with a new size and the lower triangle of $L_c$ gets copied to them using the `dlacpy` routine at the beginning of each iteration. When the two matrices are accessed before the routine, the mismatches with the predicted values decrease substantially (Figure 7.8). However, this additional access is not done during the actual tests because it influences the data flow within the application.

For `dgemm`, `dgemv`, `dpotrf` and `dtrsm` routines which involve calculations, the caching effect does not seem to have much impact. On one hand, some of the memory fetching time in those routines may get shadowed by the calculations, whereas in memory copies, the entire measured time is for the memory operations. On the other hand, the nature of the application determines the order of data accesses, where the routines may use data that is already in the cache. In either case, an effort can be made in future work to include the effect of caching in the performance models as was done in the works of [48] and [46]. It is not attempted in this project due to time constraints.

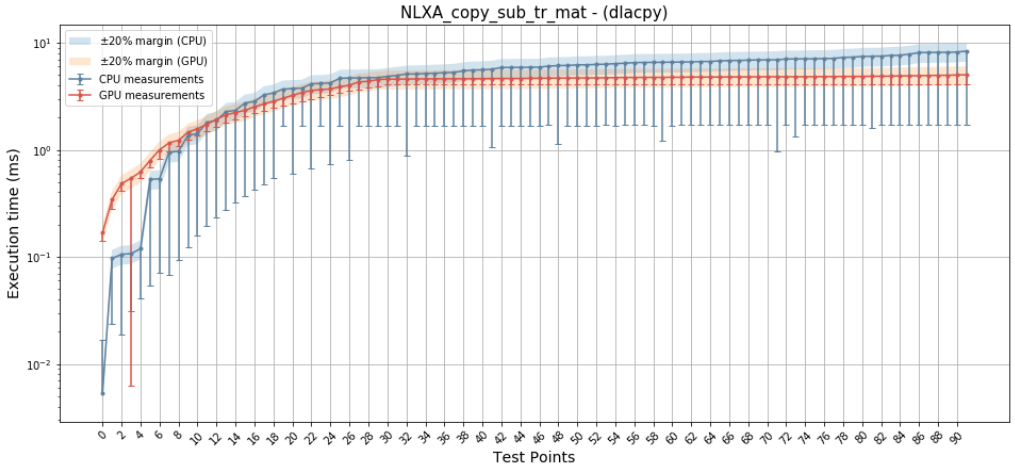



Figure 7.6: Actual and predicted execution times of `dlcapy`

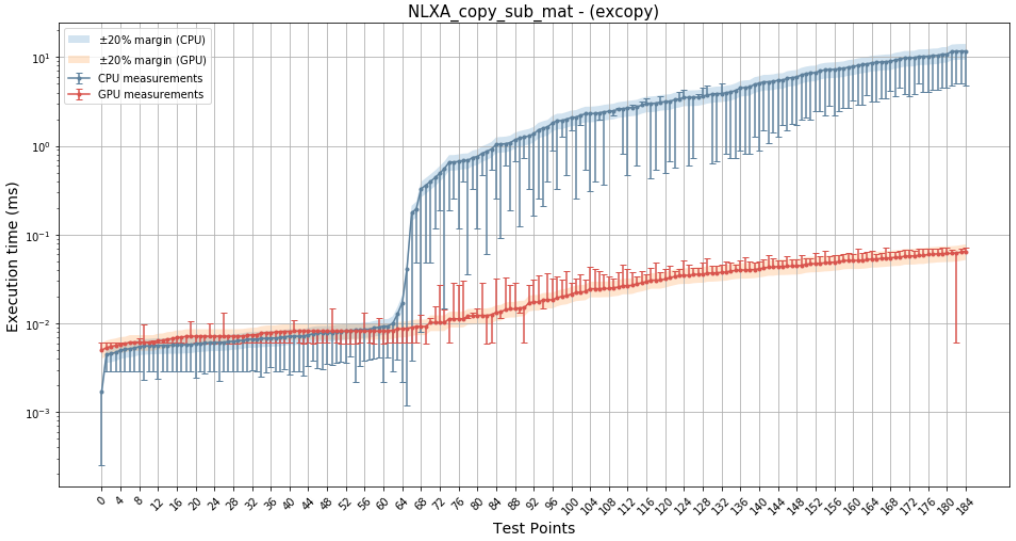



Figure 7.7: Actual and predicted execution times of `excopy`

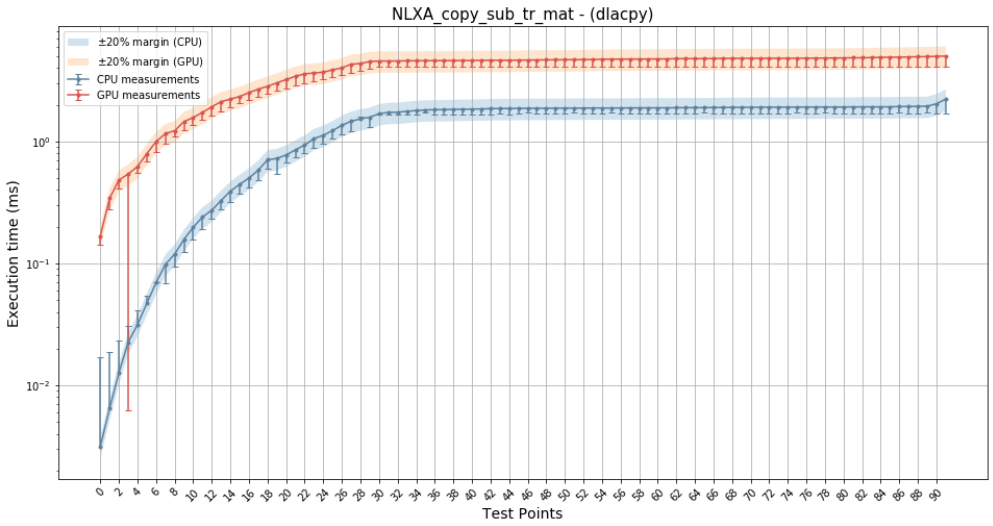



Figure 7.8: Actual and predicted execution times of `dlcapy`, with caching effect removed on CPU

### 7.1.2 Dynamic Deployment

Figure 7.9 shows the execution times of the solver with dynamic deployment, along with the CPU and GPU only measurements from before. In the general case (light green), the API manages to achieve a speed up of around 2x by later iterations when compared to the CPU only case. There are multiple reasons that it does not achieve the execution time of the all GPU curve.

Firstly, the first two `dlacpy` routines of each iteration get deployed on CPU as the predicted performance is better there[1]. These two routines are used while constructing the $L$ and $U$ matrices in Algorithm 1. However, with the caching effect shown earlier, the CPU execution time is in fact slightly worse than that of the GPU. But the decision to deploy the two copies on the CPU results in subsequent routines that use those two matrices to get deployed on CPU as well, because the cost of transferring them dominates the execution time gains. Furthermore, the fact that the API

[1]The `dlacpy` routine, uses a custom naive implementation for the GPU as explained in Section 2.1.1. As seen in Figure 13 in Appendix B, it performs worse than the CPU for the most part.

makes an overestimation for the output transfers as mentioned in Section 4.4 adds further bias on choosing the CPU. Finally, the accuracy of the prediction models plays a part as well, when the execution times for the CPU and GPU are not substantially different. A large over-prediction of the faster option and an under-prediction of the slower option could cause the API to choose wrongly.

To observe the effect of the `dlacpy` routine deployment and the overestimation of the output transfer time, another run was made. Here, the `dlacpy` routines were deployed on the GPU manually, so the large input data transfers for $L$ and $U$ do not introduce the bias explained in the previous paragraph. In addition, the output transfers were not considered when making the deployment decisions. In Figure 7.9, execution time of this custom deployment run is plotted in dark green.

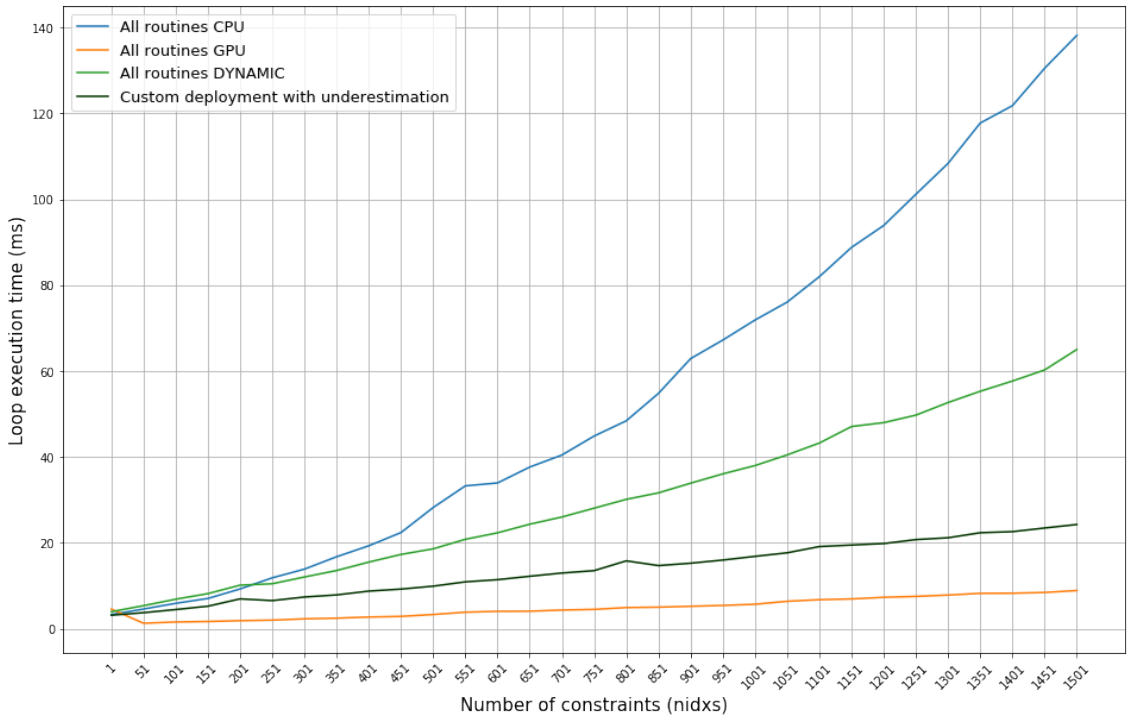



Figure 7.9: Loop execution time of QP solver (Test case 1) with dynamic deployment

The remaining gap between the custom deployment line and the all GPU line can be attributed to the model accuracies and to the fact that when individual routines are considered, the input data transfer time still may dominate the execution time. For example, the `dgemv` routine always gets scheduled to the CPU as the input transfer time of the $A$ matrix is much larger than the execution time on the CPU.

The number of routines deployed to the GPU in each iteration is plotted in Figure 7.10 for the two dynamic deployment tests (the total number of routines in a loop of the algorithm is 14). As can be seen, manually deploying the `dlacpy` instances in the loop to the GPU and neglecting the output transfers have caused more routines to be deployed to the GPU.

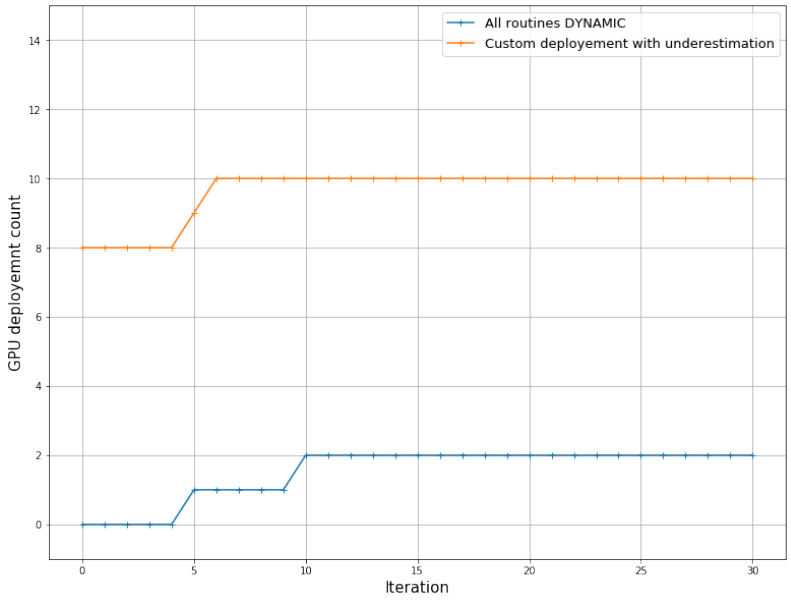



Figure 7.10: Count of GPU deployments in each iteration (Test case 1)

### 7.1.3 DAG Schedule

For the same problem size, a DAG was generated for the full program as explained in Chapter 6. Since the graph includes all iterations of the loop, it represents a loop unrolled version of the solver. Figure 7.11 shows the execution times of the CPU/GPU only and the general dynamic deployment cases along with that of two graph schedules.

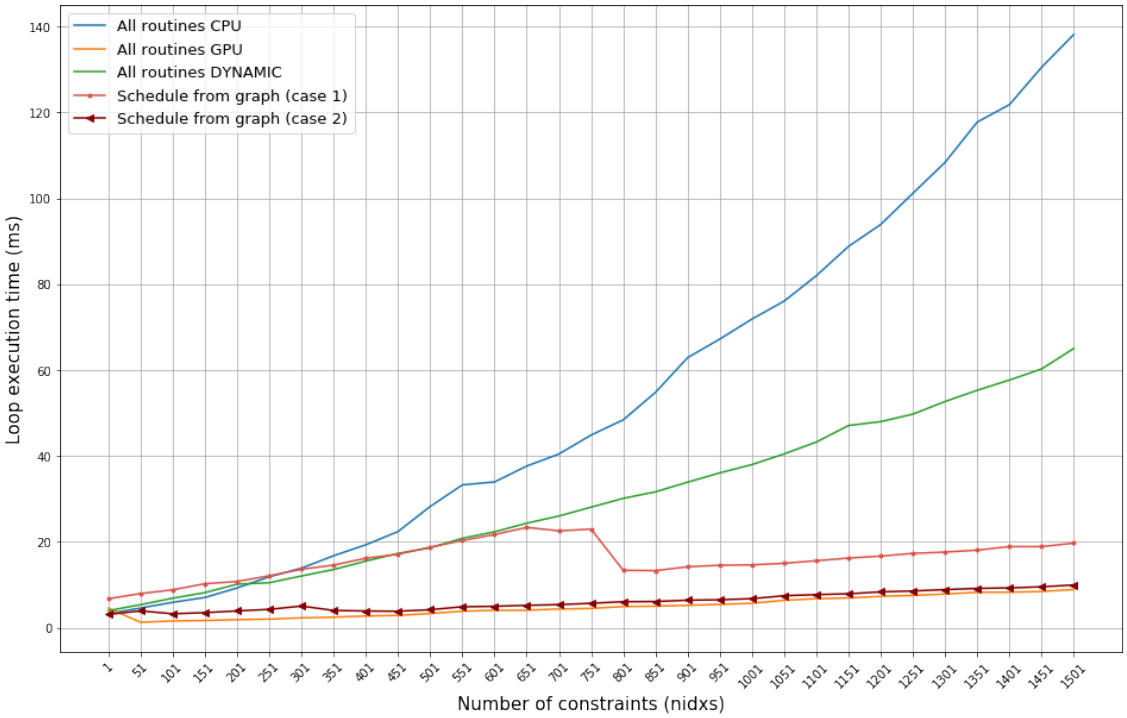



Figure 7.11: Loop execution time of QP solver (Test case 1) with DAG schedules

The schedule generated without any modifications is shown as case 1. As can be seen, the execution time is slightly worse than even that of the dynamic deployment curve at a low number of constraints. Up to around $nidxs = 201$, the loop execution time is higher than the all CPU case as well. This happens when the schedule chooses deployment combinations where intermediate GPU instances add extra data transfer times. These decisions are caused by the variations in the prediction accuracies. As mentioned in the dynamic deployment case, mis-predictions when the CPU and GPU times are not substantially different can cause adverse decisions.

To examine whether the prediction issues of `dlacpy` has an effect, case 2 was introduced. The only difference here was that the CPU predictions for the `dlacpy` was manually scaled to reduce the difference with the actual measurements, so that the predictions roughly include the caching effect (Figure 7.12). With the correction to predictions, it can be seen that the schedule performs better than the dynamic deployment from the beginning. Thus, mis-predicting `dlacpy` to be faster on the CPU, when in fact it wasn't for the most part, has caused the schedule to deploy that and other routines to the CPU, causing much worse results than expected.

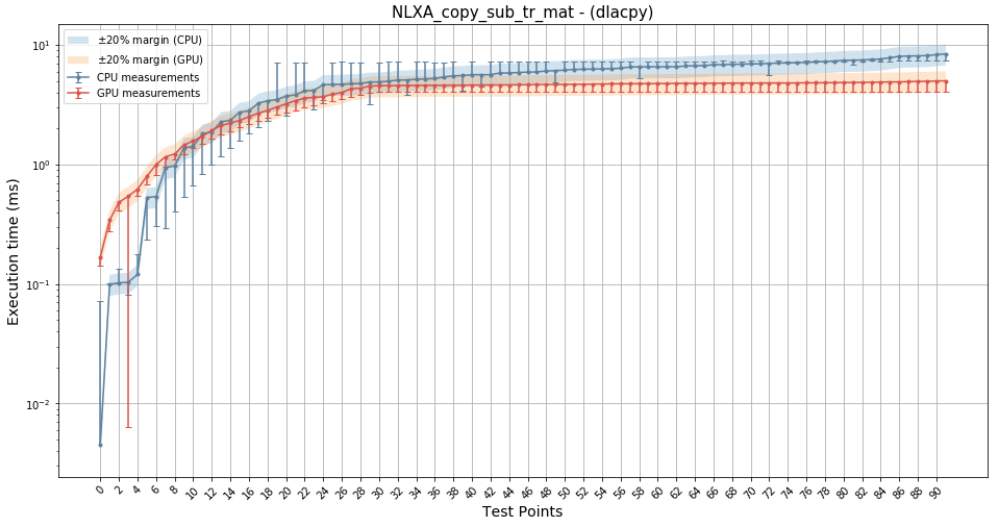



Figure 7.12: Actual and predicted execution times of `dlcapy`, with manual scaling ($\times 4.30$) on CPU

However, even the case 2 does not deploy all routines to the GPU even when the calculations become much bigger. The only substantial difference when $nidxs \geq 351$ was found to be the matrix vector multiplication (`dgemv`) at the end of the loop in Algorithm 1. Similar to dynamic deployment, this is caused by the large input data transfer time involved for the $A$ matrix which holds all the constraints.

The transfer time of $A$ to the GPU is predicted at roughly $5.10\ ms$ for this test case. The execution time predictions for the CPU and GPU are $1.26\ ms$ and $0.06\ ms$ respectively. In each iteration, as the matrix copy routine that follows `dgemv` is fixed to the CPU, the algorithm minimizes the options up to that point by choosing the CPU path which only takes $1.26\ ms$ compared to that of GPU taking $5.10+0.06 = 5.16\ ms$. Now consider 5 iterations of the loop with only these two routines present in it. With our algorithm, it will minimize options at the last matrix copy routine each time, choosing the CPU option, which results in an overall run time of $1.26 \times 5 = 6.30$. However, if the $A$ matrix was transferred in the first iteration to the GPU, suffering the penalty, the matrix can be re-used in the subsequent turns which leads to a full run time of only $5.10 + 0.06 \times 5 = 5.40$. Clearly, this gain in execution time will only increase for more iterations and it is not exploited by the proposed algorithm.

Finally, the total execution times of each test is presented in Table 7.2 for reference. It can be observed that the total time of the graph schedule from case 2 only differs by around $38ms$ from the all GPU deployment.

| Deployment | Total execution time (ms) |
|---|---|
| All routines CPU | 1695.76 |
| All routines GPU | 149.63 |
| All routines Dynamic | 932.08 |
| Schedule from graph (case 1) | 491.04 |
| Schedule from graph (case 2) | 187.0 |

Table 7.2: Total execution time with different deployment options (Test Case 1)

## 7.2 Test Case 2

Similar to the Test Case 1, a number of tests are run with different schedules for the smaller problem size of Test Case 2. The number of constraints in this test was increased by a step sizes of 20. At this size, random variations in iteration time are more visible. These variations may be caused by various reasons such as background processes running on the CPU.

In Figure 7.13 it can be seen that the loop execution time with all routines on the GPU is slightly higher in the lower iterations than for the CPU. The dynamic deployment instance (green) stays closer to the CPU curve in the lower iterations and then deviates to lower execution times by the end.

In terms of the DAG schedule, the case 1 without any modifications perform worse than both CPU only and GPU only instances the majority of the time. Since the execution times on the two platforms are closer together in this region, it is easier for the API to make adverse decisions due to prediction errors. This, together with more data transfers between the CPU and GPU, results in worse execution times than single platform cases.

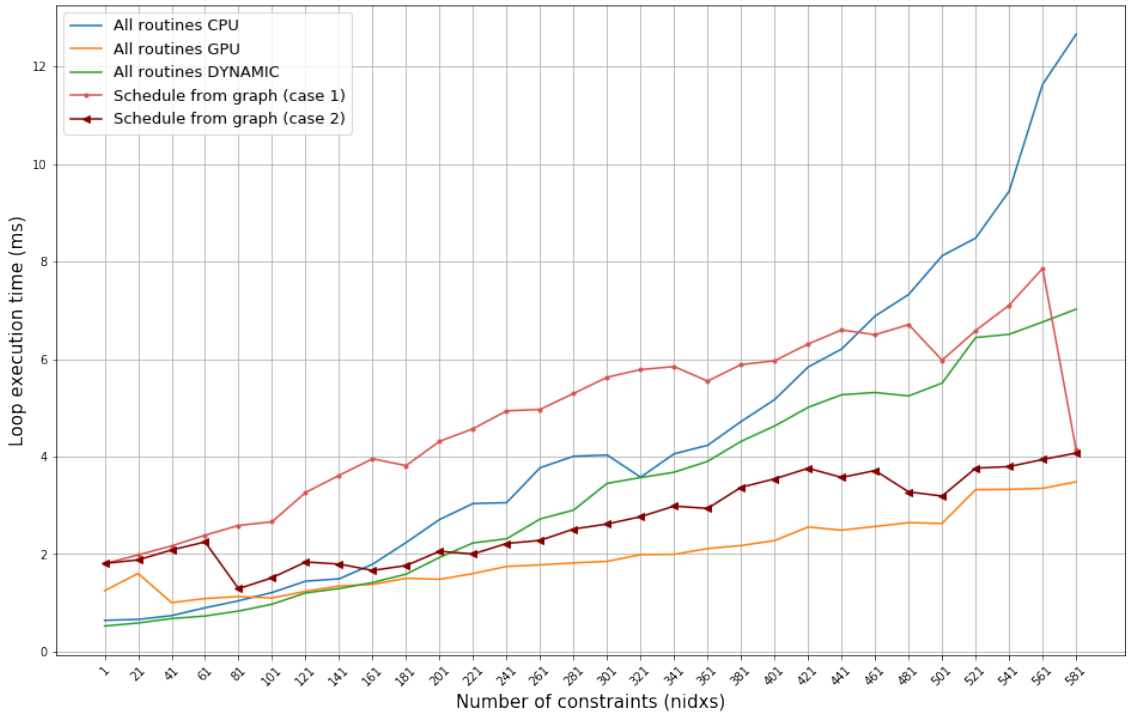



Figure 7.13: Loop execution time of QP solver (Test case 2)

For this problem size, the trigger was identified to be the accuracy of the `mkl_domatcopy` routine.

Similar to `dlacpy` in the previous case, another schedule was generated with the predictions of `mkl_domatcopy` routine scaled by a factor of $2.0$, which reduced the error margins. The case 2 curve in the figure is for this test, which performs better than dynamic deployment when $nidxs \geq 161$ and gets closer to the GPU only curve by later stages.

The total execution times of the solver with each mode are summarized in Table 7.3. A key takeaway from the Test Case 2 is that the schedules are very sensitive to the accuracy of each routine model, although the sensitivity may vary depending on the application. Since in general larger relative errors are observed for lower input sizes in most routines, schedules generated for such ranges may sometimes yield worse results than expected.

| Deployment | Total execution time (ms) |
|---|---|
| All routines CPU | 131.11 |
| All routines GPU | 59.87 |
| All routines Dynamic | 98.55 |
| Schedule from graph (case 1) | 144.74 |
| Schedule from graph (case 2) | 80.29 |

Table 7.3: Total execution time with different deployment options (Test Case 2)

### 7.2.1 Limitations of the graph scheduling algorithm

The graph scheduling algorithm manages to find an optimized schedule for a *sequential* graph. However, by sequentializing, we lose the opportunity to exploit the graph parallelism, as even with a setup of a single CPU and a single GPU it may be possible to achieve faster execution times by scheduling calculations simultaneously on both of them. Similarly, overlapping data transfers with executions may also improve the total execution time. However, the given algorithm cannot scale to address these issues without substantial modifications.

Although an algorithm such as SPAGHETtI [53] can be used to obtain a schedule that exploits graph parallelism, as mentioned in Section 3.3, implementing the parallel executions in an application will require modifications outside the API. Furthermore, such modifications will be required each time a new schedule is generated. This goes against our goal of abstracting hardware details from the user. However, further studies can be conducted on the feasibility of implementing such functionality within the API itself as suggested in Section 8.1.

Another limitation of the algorithm is not being able to tolerate short term penalties in order to achieve longer term performance gains. This is observed when a set of non-adjacent nodes have a common parent node, with a large data transfer cost associated with the edge from that parent. The issue explained previously in Section 7.1.2, with the `dgemv` routine getting scheduled to the CPU in all iterations is an example of this limitation.

In addition, as explained in Section 6.2.1, the algorithm maintains all possible deployment options for nodes that affect future nodes (having edges going past the current one). For an application with many dependencies, the maintained list of time values will grow exponentially ($2^N$, where $N$ is the number of previous nodes with edges going past a given stage). This will increase the memory usage as well as the execution time of the algorithm.

# Chapter 8

# Conclusion

In this thesis, a framework for a hardware independent API is presented for BLAS and LAPACK libraries. The API implements the CPU and GPU versions of a selected set of routines and provides a convenient interface to the user for deployment in a selected platform. In addition, it contains a dynamic deployment mode where the execution platform of corresponding routines are decided at run time based on execution and data transfer models. The same performance models are used in the data flow analysis mode to generate a data dependency graph of a marked segment of an application, with nodes and edges annotated with predictions of CPU/GPU execution times and data transfer times respectively. An intuitive algorithm is presented in the project as a proof-of-concept to derive an optimized deployment schedule from the dependency graph, which minimizes the execution time of the application segment.

In the course of developing the API the following research questions were addressed in the thesis:

1. How accurately can we predict the run time of BLAS/LAPACK routines for arbitrary input sizes?
   When the performance modeling method was extended to non-square matrices from the square only case, additional approaches were required to keep the accuracy in the required range. Multiple regression and piecewise polynomial techniques were employed to keep the mean absolute percentage errors of the models below the 20 % margin. However, manual intervention was required to determine the predictor variables and division points for polynomials, as the behaviors differ per routine. Even with those approaches, models for `dgemm`, `dgemv` and `mkl_domatcopy` routines showed more variance than the others. In addition, the effect of caching was substantial for matrix copy routines. Therefore, it can be concluded that generating accurate models to predict run time of routines for arbitrary input sizes is non trivial, but possible with sufficient tests and analysis.

2. To what extent can we use model based predictions for runtime deployment decisions?
   The answer to this question is linked with that of the previous one. With models predicting with less than 20 % error, it is feasible to make run time deployment decisions when the execution times on the platforms are sufficiently different from each other. Otherwise, over-prediction on the faster platform and under-prediction on the slower platform could result in adverse behavior. However, these margins can be taken into account in the decision making, so that the faster option would be selected only if the performance gain is above a certain threshold.

3. How can we minimize the overhead of data transfers to improve total execution time of an application?
   A method based on data dependency graphs was investigated to address limitations of dynamic deployment. The API was extended to collect routine information during run time, which is used offline to generate a DAG. The algorithm presented in the project shows that such a DAG can be used to make an optimized deployment schedule which takes into account all data dependencies and corresponding transfer times in the application segment. This way,

the overhead of data transfers is minimized, yielding better overall execution times. However, as discussed in Section 7.2.1 the presented algorithm has its limitations. It does not exploit parallelisms in the graph and does not try to transfer data in the background during executions. It also does not tolerate short term penalties for long term gains in certain cases. Furthermore, the algorithm does not scale well for large graphs when there are nodes with many children. In the next section possible gains in adopting an improved algorithm are pointed out.

## 8.1 Future Work

For further research and to improve and extend the concept of this API, we recommend the following areas:

1. Improvements and extensions to performance models
   In this project only simple and multiple regression were considered for the prediction models. While it is sufficient for modeling only for square matrices, more advanced statistical and machine learning approaches may produce more accurate models for arbitrary input sizes. In addition, when selecting predictor variables for multiple regression, the same set of variables were used in the piecewise polynomial set for a given routine. This was done for the sake of simplicity. But the models may yield better predictions if the variables were determined for each region separately, so that different behaviors for input ranges are captured more precisely. This applies for the thinness ratio defined in Section 5.1 as well.

2. Modeling the effect of caching
   As was observed in the test results, caching has a considerable influence on the execution time of routines. Therefore, more research can be done in this direction to include the effects of caching in the CPU models. Work in [48] and [46] on modeling the cache behavior and memory stalls can be referred for this.

3. Extension to hardware implementations
   In our project, a system with only a single CPU and a GPU were considered, with Intel MKL and Nvidia Cublas libraries. However, as discussed in Section 3.1, hardware acceleration of BLAS/LAPACK is a highly researched and developing area. Our API can easily be extended to include more libraries while keeping the interface to the user unchanged. For example, better performance may be achieved on a compute node with multiple GPUs using libraries like MAGMA [17] and PLASMA [36]. Similarly, batched implementations of routines can be included to be used in applications where there is a large number of calculations with small data sizes. In addition, the API can also be updated with support to platforms such as FPGAs, with suitable implementations on hardware.

4. Implementing a scheduler in the API
   A suggested improvement on run time deployment is for a "scheduler" to maintain a queue of routines as they are called at run time. The routines can then be deployed when the data is required by a non-API function. If such a method can be implemented, it would benefit applications like the QP solver where multiple routines execute in a chain. The scheduler would have the dependency information of the adjacent routines, which can be used to deploy them in an optimal way at runtime using a scheduling algorithm. In terms of the graph scheduling mode of the API, having such a scheduler in place will help exploiting graph parallelism of the DAG as well. It may be possible to deploy multiple non-dependent routines simultaneously in available platforms, without further modifications by the user in the application. This way, deployment schedules from algorithms in literature such as SPAGHETtI [53] can be executed by the API itself.

5. Better graph scheduling algorithms
   Section 3.3 discussed a number of DAG scheduling algorithms presented in literature. A more comprehensive study can be done on suitable approaches that can exploit the inherent graph parallelism, where utilization of the available compute resources is maximized.

The SPAGHETtI [53] is one such algorithm which can exploit parallelism in the graph. In addition, the performance can be improved by overlapping executions and data transfers between the platforms. Frameworks such as StarPU [64] and DAGuE [65] may also provide good insight on implementing such algorithms.

# Bibliography


[1] C. Lawson, Richard Hanson, David Kincaid, and Fred Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, 09 1979.

[2] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Supercomputing '90:Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 2–11, Nov 1990.

[3] MathWorks. Lapack in matlab. https://nl.mathworks.com/help/matlab/math/lapack-in-matlab.html. Accessed: November 2019.

[4] Numpy.org. Numpy documentation - accelerated blas/lapack libraries. https://numpy.org/devdocs/user/building.html. Accessed: November 2019.

[5] Oak Ridge National Laboratory AT&T Bell Laboratories, University of Tennessee. Netlib repository. https://netlib.org. Accessed: November 2019.

[6] Intel. Intel math kernal library (mkl). https://software.intel.com/mkl. Accessed: November 2019.

[7] AMD. Amd optimizing cpu libraries (aocl). https://developer.amd.com/amd-aocl/. Accessed: January 2020.

[8] Nvidia. Nvidia cublas. https://developer.nvidia.com/cublas. Accessed: November 2019.

[9] Nvidia. Nvidia cusolver. https://developer.nvidia.com/cusolver. Accessed: November 2019.

[10] Oracle. Sun performance library for programs with intensive compuation. https://docs.oracle.com/cd/E18659_01/html/821-2763/gjgis.html/. Accessed: November 2019.

[11] ASML. Asml official website. https://www.asml.com/en. Accessed: November 2019.

[12] Nvidia. Cuda 7.0 performance report. http://on-demand.gputechconf.com/gtc/2015/webinar/gtc-express-cuda7-performance-overview.pdf. Accessed: January 2020.

[13] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance. In *Supercomputing '96:Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pages 5–5, Jan 1996.

[14] Openblas. https://www.openblas.net/. Accessed: January 2020.

[15] Automatically tuned linear algebra software (atlas). http://math-atlas.sourceforge.net/. Accessed: January 2020.

[16] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[17] ICL University of Tennessee. Matrix algebra on gpu and multicore architectures (magma). https://icl.cs.utk.edu/magma/. Accessed: November 2019.

[18] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 20:1–20:11, New York, NY, USA, 2016. ACM.

[19] Encyclopedia.com. Performance model. https://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/performance-model, Jan 2020. Accessed: March 2020.

[20] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[21] Mohamed A Bamakhrama, Alejandro Arrizabalaga, Frank Overman, Jean-Paul Smeets, Kornel van der Sommen, Remko van der Vossen, and John Wagensveld. Gpu acceleration of real-time control loops. *arXiv preprint arXiv:1902.08018*, 2019.

[22] R.K. Gupta. Investigation of hardware acceleration of mathematical calculations, 2019.

[23] Michael Boyer, Jiayuan Meng, and Kalyan Kumaran. Improving gpu performance prediction with data transfer modeling. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, page 1097–1106, USA, 2013. IEEE Computer Society.

[24] John von Neumann. Introduction to "the first draft report on the edvac". *Annals of the History of Computing*, 15(1):11–21, 1993.

[25] Intel. Intel xeon processors. https://www.intel.com/content/www/us/en/products/processors/xeon.html. Accessed: February 2020.

[26] Niels Hagoort. Exploring the gpu architecture. https://nielshagoort.com/2019/03/12/exploring-the-gpu-architecture, March 2019. Accessed: February 2020.

[27] Nvidia. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: February 2020.

[28] Wikipedia. Pci express. https://en.wikipedia.org/wiki/PCI-Express, February 2020. Accessed: February 2020.

[29] Emmanuel Agullo et al. Dynamic scheduling within magma. http://www.einfrastructureforum.ac.uk/sites/default/files/uploads/ProjectFiles/ASEArch/magma.pdf, April 2012. Accessed: February 2020.

[30] ICL University of Tennessee. High-performance batched computations. http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/Batched-BLAS-2016/Day1/04_Azzam_BBLAS_Approaches_App.pdf, May 2016. Accessed: March 2020.

[31] Jack Dongarra, Iain Duff, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J Higham, Jonathan Hogg, Pedro Valero Lara, Piotr Luszczek, Mawussi Zounon, et al. Batched blas (basic linear algebra subprograms) 2018 specification. 2018.

[32] Siham Tabik, G Ortega, and Ester M Garzón. Performance evaluation of kernel fusion blas routines on the gpu: iterative solvers as case study. *The Journal of Supercomputing*, 70(2):577–587, 2014.

[33] Cedric Nugteren. Clblast: A tuned opencl blas library. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18, pages 5:1–5:10, New York, NY, USA, 2018. ACM.

[34] Intel. Introducing batch gemm operations. https://software.intel.com/en-us/articles/introducing-batch-gemm-operations. Accessed: February 2020.

[35] Cedric et al. Nugteren. Clblast: The tuned opencl blas library. https://github.com/CNugteren/CLBlast. Accessed: January 2020.

[36] ICL University of Tennessee et al. Parallel linear algebra software for multicore architectures (plasma). https://icl.bitbucket.io/plasma/, February 2020. Accessed: February 2020.

[37] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, jul 2009.

[38] Nvidia. Nvidia nvblas. https://docs.nvidia.com/cuda/nvblas/index.html, November 2019. Accessed: February 2020.

[39] ICL University of Tennessee. Parallel runtime scheduling and execution controller (parsec). http://icl.utk.edu/parsec/. Accessed: February 2020.

[40] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 156–165, May 2015.

[41] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. Fblas: Streaming linear algebra on fpga. *arXiv preprint arXiv:1907.07929*, 2019.

[42] Sébastien Rousseaux, Damien Hubaux, Pierre Guisset, and Jean-Didier Legat. A high performance fpga-based accelerator for blas library implementation. *Proc. of the Third Annual Reconfigurable Systems Summer Institute (RSSI'07)*, 2007.

[43] Vipin Kizheppatt and Suhaib Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys*, 51:1–39, 07 2018.

[44] G. C. Cardarilli, L. Di Carlo, A. Nannarelli, F. M. Pandolfi, and M. Re. A framework for dynamically-loaded hardware library (hll) in fpga acceleration. In *2015 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 291–296, Dec 2015.

[45] S. Kestur, J. D. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pages 288–293, July 2010.

[46] Elmar Peise. Performance modeling and prediction for dense linear algebra. *arXiv preprint arXiv:1706.01341*, 2017.

[47] Elmar Peise. Experimental linear algebra performance studies (elaps. https://github.com/HPAC/ELAPS. Accessed: November 2019.

[48] Roman Iakymchuk. *Performance modeling and prediction for linear algebra algorithms*. PhD thesis, Hochschulbibliothek der Rheinisch-Westfälischen Technischen Hochschule Aachen, 2012.

[49] I. Baldini, S. J. Fink, and E. Altman. Predicting gpu performance from cpu runs using machine learning. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 254–261, Oct 2014.

[50] M. Amarís, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram. A comparison of gpu execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 326–333, Oct 2016.

[51] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73–84. Springer, 2008.

[52] Jörg Keller and Rainer Gerhards. Peelsched: a simple and parallel scheduling algorithm for static taskgraphs. *PARS: Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware*, 28, 10 2014.

[53] Denis Barthou and Emmanuel Jeannot. Spaghetti: Scheduling/placement approach for task-graphs on heterogeneous architecture. In Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, pages 174–185, Cham, 2014. Springer International Publishing.

[54] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[55] Houssam-Eddine Zahaf, Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Giuseppe Lipari. A c-dag task model for scheduling complex real-time tasks on heterogeneous platforms: preemption matters. *arXiv preprint arXiv:1901.02450*, 2019.

[56] Intel. Intel xeon gold 6134 processor. https://ark.intel.com/content/www/us/en/ark/products/120493/intel-xeon-gold-6134-processor-24-75m-cache-3-20-ghz.html. Accessed: January 2020.

[57] Nvidia. Nvidia tesla v100. https://www.nvidia.com/en-us/data-center/v100/. Accessed: January 2020.

[58] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[59] Nvidia. Unified memory in cuda 6. https://devblogs.nvidia.com/unified-memory-in-cuda-6/. Accessed: January 2020.

[60] Nvidia. Maximizing unified memory performance in cuda. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/. Accessed: January 2020.

[61] T. M. Smith, R. v. d. Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee. Anatomy of high-performance many-threaded matrix multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1049–1059, May 2014.

[62] Scikit-learn. https://scikit-learn.org/stable/. Accessed: January 2020.

[63] Julia. Julia micro-benchmarks. https://julialang.org/benchmarks/. Accessed: January 2020.

[64] A unified runtime system for heterogeneous multicore architectures. http://starpu.gforge.inria.fr/. Accessed: February 2020.

[65] ICL University of Tennessee. The direct acyclic graph environment. https://icl.utk.edu/dague/overview/index.html. Accessed: February 2020.

# Appendices

# A Abbreviations

| | |
|---|---|
| AOCL | AMD Optimizing CPU Libraries |
| API | Application Programming Interface |
| BLAS | Basic Linear Algebra Subprograms |
| CPU | Central Processing Unit |
| DAG | Directed Acyclic Graph |
| DDR | Double Data Rate |
| FLOPS | Floating point Operations Per Second |
| FPGA | Field-Programmable Gate Array |
| GPU | Graphical Processing Unit |
| HBM | High Bandwidth Memory |
| LAPACK | Linear Algebra PACKage |
| LLC | Last Level Cache |
| MKL | Math Kernel Library |
| PCI | Peripheral Component Interconnect |
| SIMD | Single Instruction, Multiple Data |
| SVM | Support Vector Machines |

# B Performance Models

This section provides the measured execution times of the selected routines and their finalized models. The plots were explained in detail in Section 5.1, using `dgemv` as an example.

## B.1 Matrix-matrix multiplication - `dgemm`

| Attribute | Values |
|---|---|
| Size range | Initial values selected as powers of 2 in range $[32, ..., 4096]$ and multiples of 500 in range $[500, 1000, ..., 4000]$. Then all dimension combinations that result in the same computation complexity was calculated keeping the sizes within $[1, 10000]$ range. |
| Number of measurements | The above size range resulted in 1062 measurements. |
| Dimension arguments | $m, n, k$ |
| Complexity | $O(mnk)$ |
| Thinness condition | `m*r < n || m*r < k || n*r < m || n*r < k || k*r < m || k*r < n` |

Table 1: `dgemm` test attributes

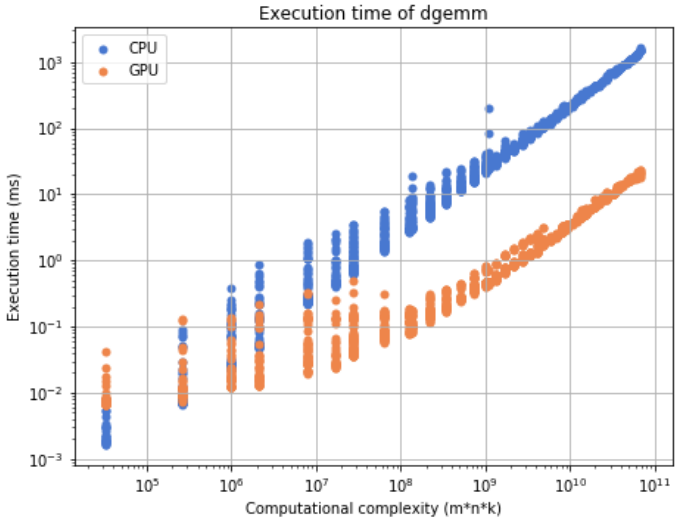



Figure 1: Execution time measurements for `dgemm`

### B.1.1 CPU Model

Features: $m^2n, m^2k, n^2m, n^2k, k^2m, k^2n, mnk$

$$y_{pred} = \begin{cases} b_1 \cdot m^2n + b_2 \cdot m^2k + b_3 \cdot n^2m + b_4 \cdot n^2k + b_5 \cdot k^2m + b_6 \cdot k^2n + b_7 \cdot mnk + b_8 & \text{if } mnk < 10^6 \\ c_1 \cdot m^2n + c_2 \cdot m^2k + c_3 \cdot n^2m + c_4 \cdot n^2k + c_5 \cdot k^2m + c_6 \cdot k^2n + c_7 \cdot mnk + c_8 & \text{if } 10^6 \leq mnk < 10^9 \\ d_1 \cdot m^2n + d_2 \cdot m^2k + d_3 \cdot n^2m + d_4 \cdot n^2k + d_5 \cdot k^2m + d_6 \cdot k^2n + d_7 \cdot mnk + d_8 & \text{if } mnk \geq 10^9 \end{cases}$$

Where,

$$
\begin{array}{lll}
b_1 = 3.188860E-10 & c_1 = 9.823811E-11 & d_1 = 3.202378E-11 \\
b_2 = 1.382362E-10 & c_2 = 1.080467E-10 & d_2 = 1.817564E-10 \\
b_3 = 2.265036E-10 & c_3 = 9.100953E-11 & d_3 = 1.481524E-11 \\
b_4 = 2.237570E-10 & c_4 = 6.052924E-11 & d_4 = 7.690169E-11 \\
b_5 = 1.287367E-10 & c_5 = 1.014187E-10 & d_5 = 1.021577E-10 \\
b_6 = 2.502405E-10 & c_6 = 4.170197E-11 & d_6 = 4.735532E-11 \\
b_7 = 1.729919E-08 & c_7 = 2.360127E-08 & d_7 = 2.030274E-08 \\
b_8 = 1.362984E-03 & c_8 = 1.007329E-02 & d_8 = 2.972709E+00
\end{array}
$$

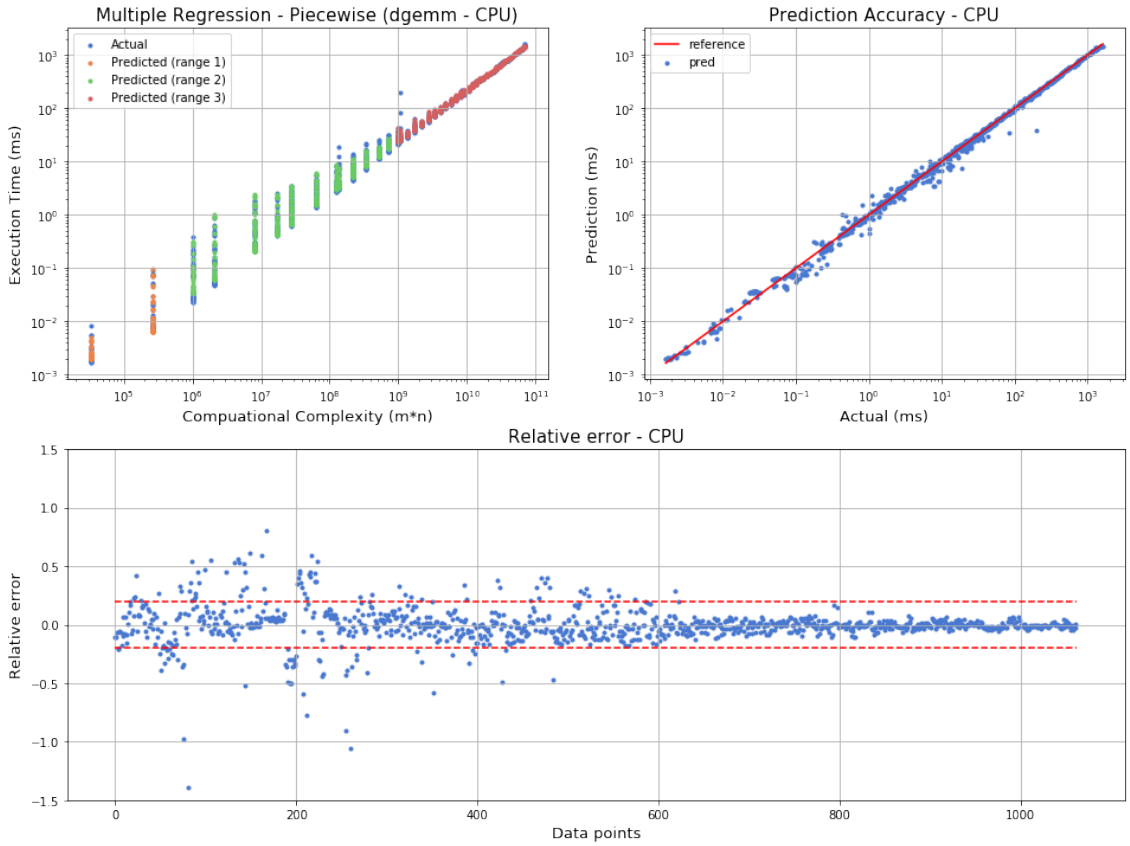



Figure 2: Accuracy of model `dgemm` - CPU

| Metric | Value |
|---|---|
| Mean absolute percentage error | 9.11 % |
| Max absolute percentage error | 139.28 % |
| Percentage of outliers | 11.96 % |

Table 2: `dgemm` CPU model accuracy

### B.1.2 GPU Model

Features: $m, n, k, m^2, n^2, k^2, mn, mk, nk, mnk$

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \cdot n + b_3 \cdot k + b_4 \cdot m^2 + b_5 \cdot n^2 + b_6 \cdot k^2 + b_7 \cdot mn \\ +b_8 \cdot mk + b_9 \cdot nk + b_{10} \cdot mnk + b_{11} & \text{if } mnk < 10^5 \\ \\ c_1 \cdot m + c_2 \cdot n + c_3 \cdot k + c_4 \cdot m^2 + c_5 \cdot n^2 + c_6 \cdot k^2 + c_7 \cdot mn \\ +c_8 \cdot mk + c_9 \cdot nk + c_{10} \cdot mnk + c_{11} & \text{if } 10^5 \leq mnk < 10^7 \\ \\ d_1 \cdot m + d_2 \cdot n + d_3 \cdot k + d_4 \cdot m^2 + d_5 \cdot n^2 + d_6 \cdot k^2 + d_7 \cdot mn \\ +d_8 \cdot mk + d_9 \cdot nk + d_{10} \cdot mnk + d_{11} & \text{if } 10^7 \leq mnk < 10^9 \\ \\ f_1 \cdot m + f_2 \cdot n + f_3 \cdot k + f_4 \cdot m^2 + f_5 \cdot n^2 + f_6 \cdot k^2 + f_7 \cdot mn \\ +f_8 \cdot mk + f_9 \cdot nk + f_{10} \cdot mnk + f_{11} & \text{if } mnk \geq 10^9 \end{cases}$$

Where,

$$\begin{array}{llll} b_1 = -1.021454e-05 & c_1 = -1.197942e-06 & d_1 = 5.845717e-07 & f_1 = 7.120777e-05 \\ b_2 = -2.244785e-05 & c_2 = 8.694128e-07 & d_2 = 1.740683e-06 & f_2 = 7.100771e-05 \\ b_3 = 3.272036e-05 & c_3 = 3.569870e-05 & d_3 = 3.441209e-05 & f_3 = 1.217734e-04 \\ b_4 = -2.568222e-09 & c_4 = 7.033282e-11 & d_4 = 9.561252e-11 & f_4 = -6.054775e-09 \\ b_5 = -3.624547e-10 & c_5 = 7.428646e-11 & d_5 = -2.687771e-11 & f_5 = -6.164011e-09 \\ b_6 = -1.996822e-08 & c_6 = -2.600823e-09 & d_6 = -2.534741e-09 & f_6 = -6.992795e-09 \\ b_7 = 6.642099e-07 & c_7 = 1.538421e-08 & d_7 = 3.186237e-09 & f_7 = 4.566090e-09 \\ b_8 = 2.712864e-07 & c_8 = 1.921575e-08 & d_8 = 8.909711e-09 & f_8 = 1.617112e-08 \\ b_9 = 9.195006e-07 & c_9 = 3.271383e-09 & d_9 = 7.064973e-09 & f_9 = 1.624427e-08 \\ b_10 = 0.000000e+00 & c_10 = 1.760548e-09 & d_10 = 4.111111e-10 & f_10 = 2.990509e-10 \\ b_11 = 5.445280e-03 & c_11 = 8.262003e-03 & d_11 = 2.315747e-02 & f_11 = -2.063431e-01 \end{array}$$

| Metric | Value |
|---|---|
| Mean absolute percentage error | 11.72 % |
| Max absolute percentage error | 117.88 % |
| Percentage of outliers | 17.42 % |

Table 3: `dgemm` GPU model accuracy

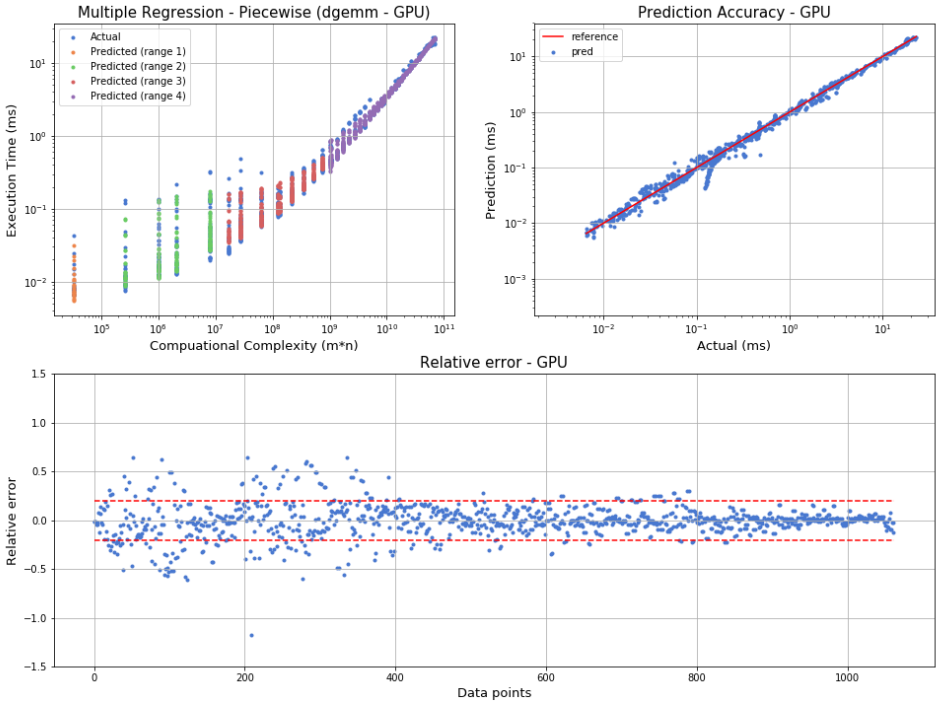



Figure 3: Accuracy of model `dgemm` - GPU

## B.2 Matrix-vector multiplication - `dgemv`

| Attribute | Values |
|---|---|
| Size range | Initial values selected as powers of 2 in range $[32, ..., 4096]$ and multiples of 50 in range $[50, 1000..., 400]$. Then all dimension combinations that result in the same computation complexity was calculated keeping the sizes within $[1, 10000]$ range. |
| Number of measurements | 3093 measurements. |
| Dimension arguments | $m, n$ |
| Complexity | $O(mn)$ |
| Thinness condition | `m*r < n || n*r < m` |

Table 4: `dgemv` test attributes

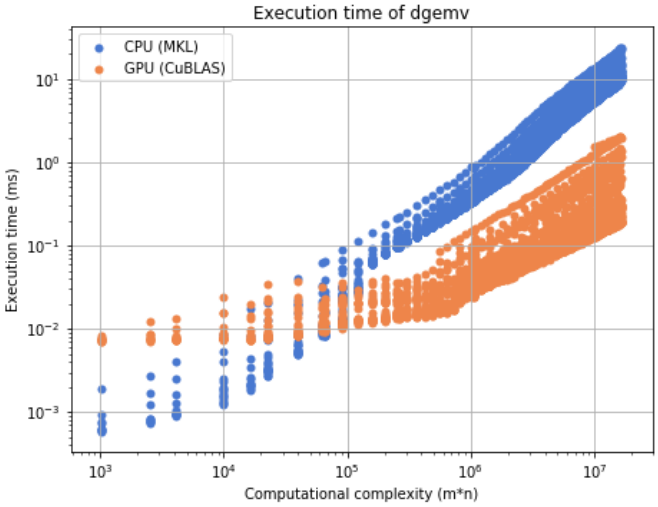



Figure 4: Execution time measurements for `dgemv`

### B.2.1 CPU Model

Features: $m, n, mn$

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \cdot n + b_3 \cdot mn + b_4 & \text{if } mnk < 10^5 \\ c_1 \cdot m + c_2 \cdot n + c_3 \cdot mn + c_4 & \text{if } 10^5 \leq mnk < 2 \times 10^6 \\ d_1 \cdot m + d_2 \cdot n + d_3 \cdot mn + d_4 & \text{if } mnk \geq 2 \times 10^6 \end{cases}$$

Where,

$$\begin{aligned} b_1 &= 3.200676E-07 & c_1 &= 6.711642E-07 & d_1 &= 1.239206E-06 \\ b_2 &= 2.094603E-06 & c_2 &= 1.085442E-06 & d_2 &= 1.791089E-06 \\ b_3 &= 1.801089E-07 & c_3 &= 3.652427E-07 & d_3 &= 6.905138E-07 \\ b_4 &= 1.662675E-03 & c_4 &= 4.548914E-04 & d_4 &= -8.179780E-01 \end{aligned}$$

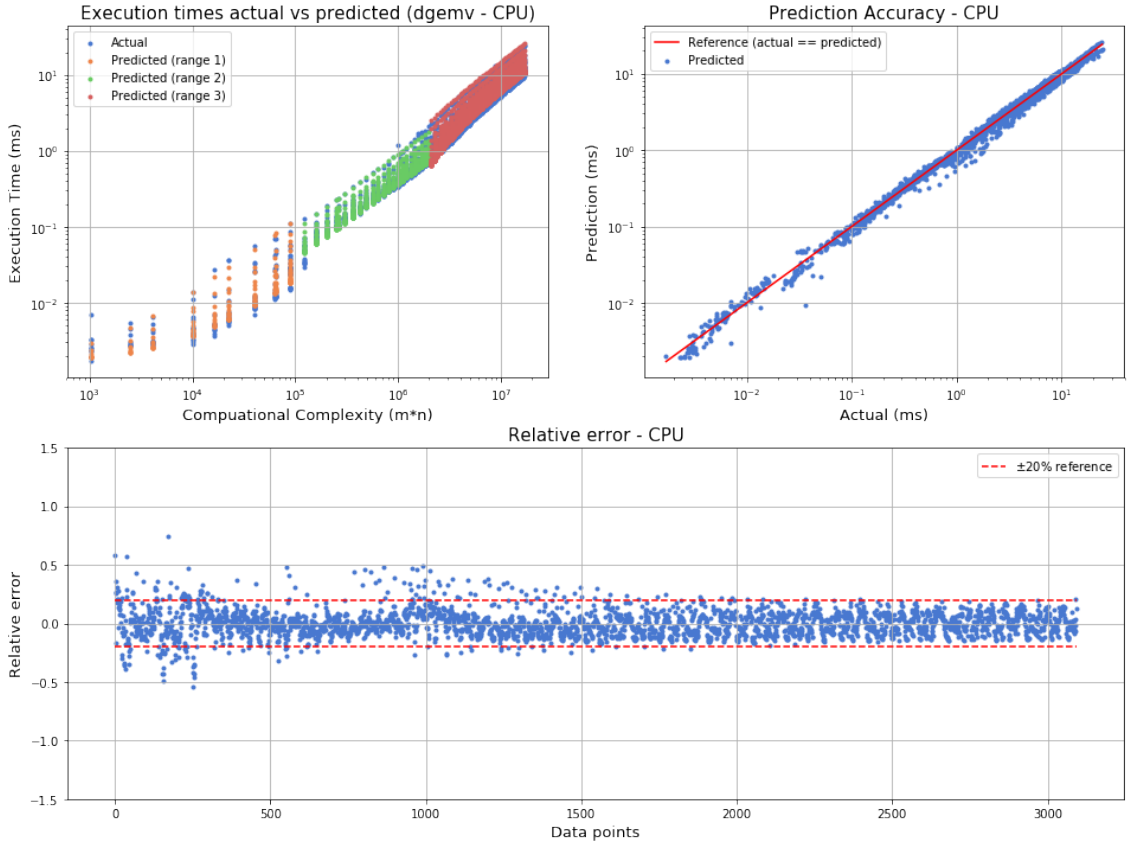



Figure 5: Accuracy of model `dgemv` - CPU

| Metric | Value |
|---|---|
| Mean absolute percentage error | 8.57 % |
| Max absolute percentage error | 74.09 % |
| Percentage of outliers | 6.43 % |

Table 5: `dgemv` CPU model accuracy

### B.2.2 GPU Model

Features: $m, n, mn$

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \cdot n + b_3 \cdot mn + b_4 & \text{if } mnk < 10^6 \\ c_1 \cdot m + c_2 \cdot n + c_3 \cdot mn + c_4 & \text{if } mnk \geq 10^6 \end{cases}$$

Where,

$$\begin{aligned} b_1 &= -1.719696E-09 & c_1 &= 9.268826E-09 \\ b_2 &= 2.256339E-07 & c_2 &= 2.151023E-07 \\ b_3 &= 1.931148E-08 & c_3 &= 1.254851E-08 \\ b_4 &= 9.730325E-03 & c_4 &= 2.506619E-02 \end{aligned}$$

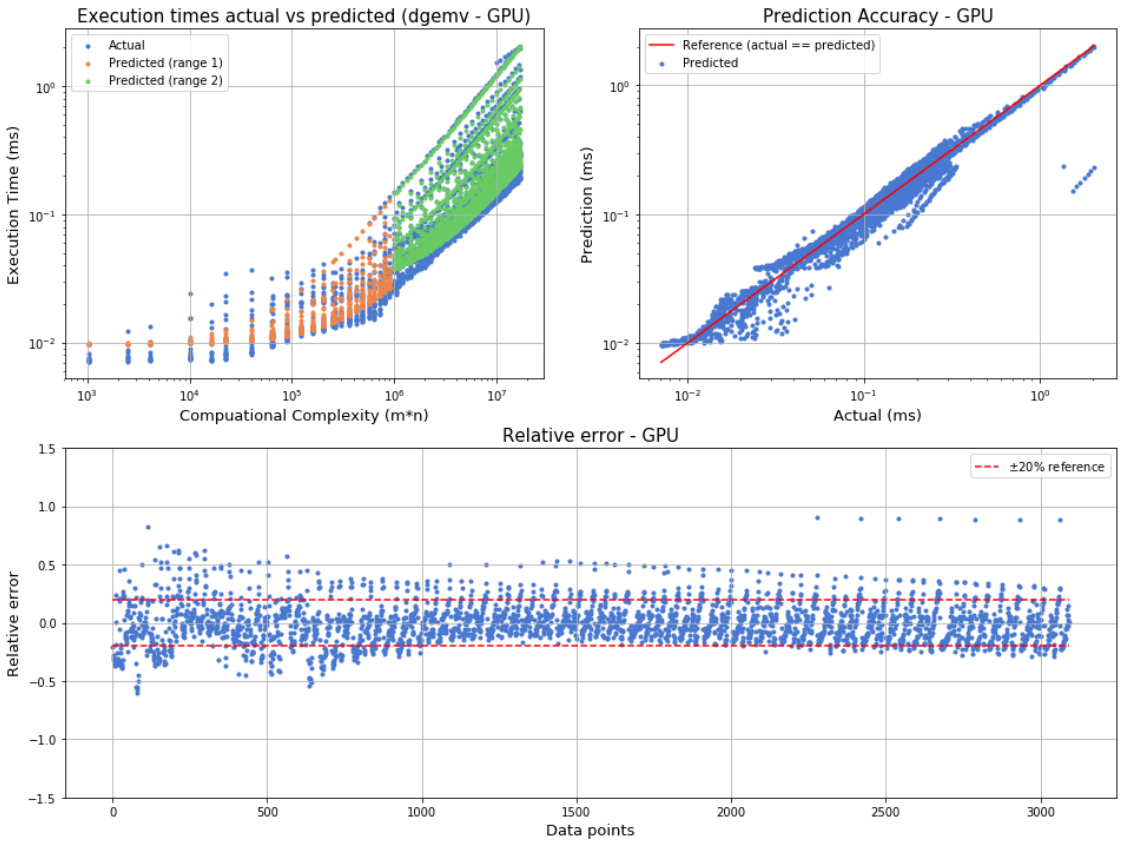



Figure 6: Accuracy of model `dgemv` - GPU

| Metric | Value |
|---|---|
| Mean absolute percentage error | 13.61 % |
| Max absolute percentage error | 90.15 % |
| Percentage of outliers | 23.08 % |

Table 6: `dgemv` GPU model accuracy

## B.3 Cholesky factorization - `dpotrf`

| Attribute | Values |
|---|---|
| Size range | Powers of 2 in range $[32, ..., 4096]$ and multiples of 100 in range $[100, 200..., 4000]$ |
| Number of measurements | 48 measurements. |
| Dimension arguments | $m$ |
| Complexity | $O(m^3)$ |

Table 7: `dpotrf` test attributes

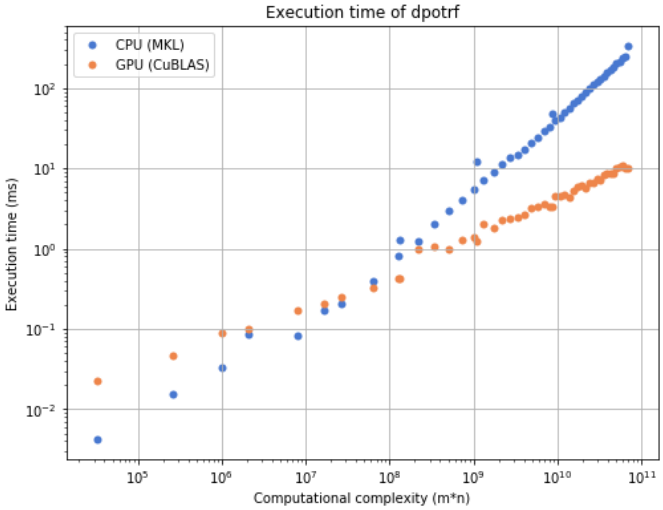



Figure 7: Execution time measurements for `dpotrf`

### B.3.1 CPU Model

Features: $m, m^3$

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \cdot m^3 + b_3 & \text{if } m^3 < 10^8 \\ c_1 \cdot m + c_2 \cdot m^3 + c_3 & \text{if } m^3 \geq 10^8 \end{cases}$$

Where,

$$\begin{aligned} bb_1 &= 4.116973E-04 & c_1 &= 1.493065E-03 \\ bb_2 &= 3.640432E-09 & c_2 &= 3.946053E-09 \\ bb_3 &= -9.154697E-03 & c_3 &= -1.782443E-01 \end{aligned}$$

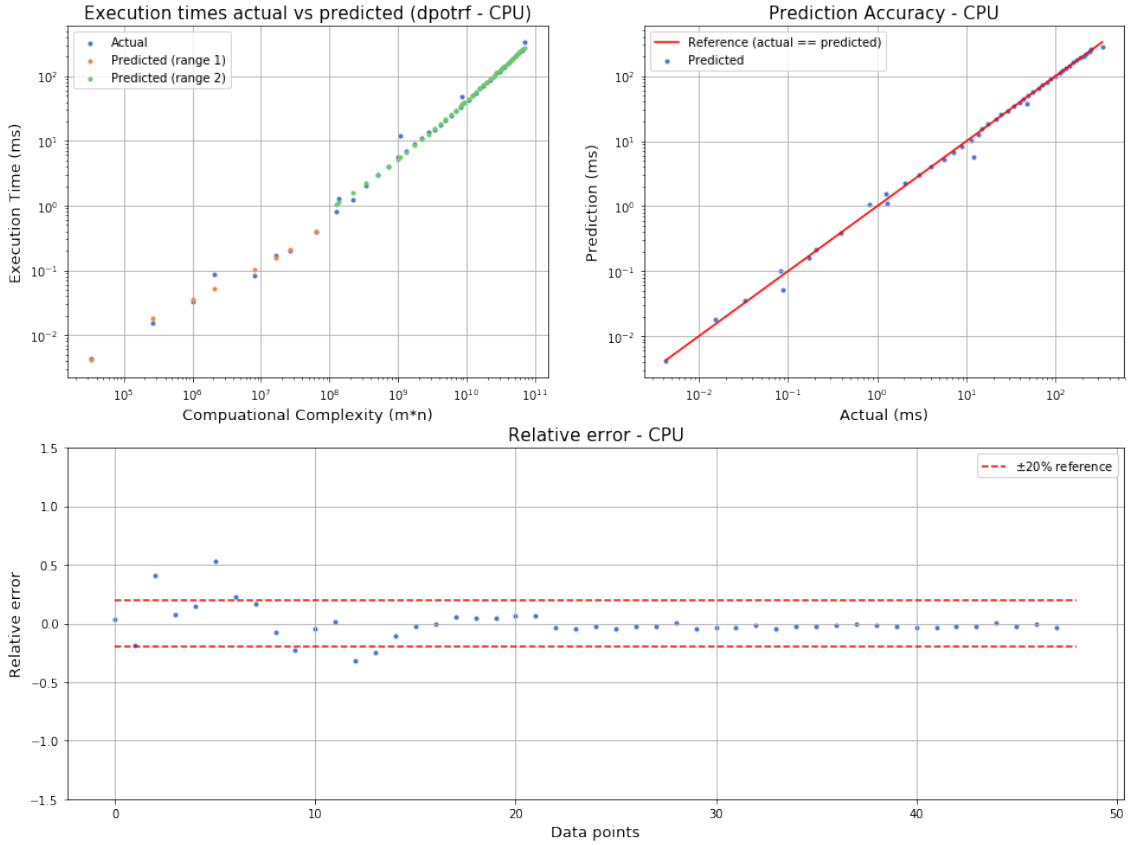



Figure 8: Accuracy of model `dpotrf` - CPU

| Metric | Value |
|---|---|
| Mean absolute percentage error | 7.90 % |
| Max absolute percentage error | 53.41 % |
| Percentage of outliers | 12.50 % |

Table 8: `dpotrf` CPU model accuracy

### B.3.2 GPU Model

Features: $m, m^3$

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \cdot m^3 + b_3 & \text{if } m^3 < 2 \times 10^8 \\ c_1 \cdot m + c_2 \cdot m^3 + c_3 & \text{if } m^3 \geq 2 \times 10^8 \end{cases}$$

Where,

$$\begin{aligned} b_1 &= 8.361554E-04 & c_1 &= 1.836161E-03 \\ b_2 &= 3.513062E-11 & c_2 &= 6.072266E-11 \\ b_3 &= -3.895336E-03 & c_3 &= -3.843064E-01 \end{aligned}$$

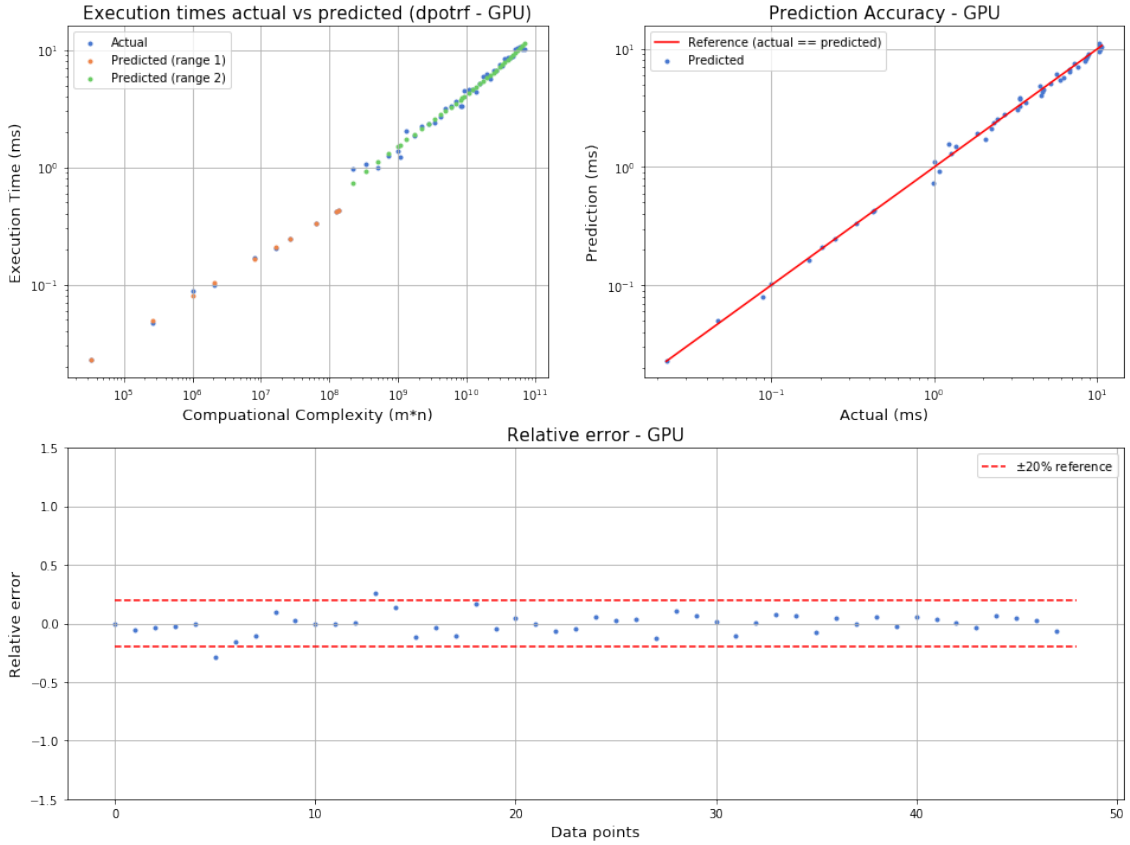



Figure 9: Accuracy of model `dpotrf` - GPU

| Metric | Value |
|---|---|
| Mean absolute percentage error | 6.43 % |
| Max absolute percentage error | 28.29 % |
| Percentage of outliers | 4.17 % |

Table 9: `dpotrf` GPU model accuracy

## B.4 Triangular matrix system solver - `dtrsm`

| Attribute | Values |
|---|---|
| Size range | First dimension $m$ given the values of powers of 2 in range $[2, ..., 4096]$ and multiples of 100 in range $[100, 200..., 4000]$. Only value 1 used for the second dimension $n$ as other values are not used in the example application. |
| Number of measurements | 48 measurements for both with and without input transposed cases. |
| Dimension arguments | $m, n$ |
| Complexity | $O(m^2n)$ |

Table 10: `dtrsm` test attributes

It should be noted that the measurements show much variance when $n$ takes other values as shown in Figure 5.4 in Chapter 5. Therefore, for a more general case, the modeling has to be done including a range of values for $n$ as well.

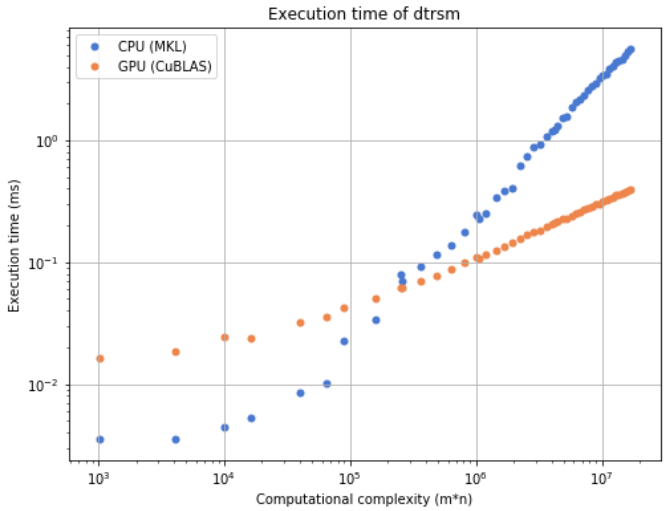



Figure 10: Execution time measurements for `dtrsm`

### B.4.1 CPU Model

Model when `trans` flag is `N` (No transposing).
Features: $m^2, n^2, mn, m^2n$

$$y_{pred} = \begin{cases} b_1 \cdot m^2 + b_2 \cdot n^2 + b_3 \cdot mn + b_4 \cdot m^2n + b_5 & \text{if } m^2n < 2 \times 10^5 \\ c_1 \cdot m^2 + c_2 \cdot n^2 + c_3 \cdot mn + c_4 \cdot m^2n + c_5 & \text{if } 2 \times 10^5 \leq m^2n < 2 \times 10^6 \\ d_1 \cdot m^2 + d_2 \cdot n^2 + d_3 \cdot mn + d_4 \cdot m^2n + d_5 & \text{if } m^2n \geq 2 \times 10^6 \end{cases}$$

Where,

$$\begin{array}{lll} b_1 = 1.344218E-07 & c_1 = 1.240022E-07 & d_1 = 1.661670E-07 \\ b_2 = 0.000000E+00 & c_2 = 1.323489E-23 & d_2 = 0.000000E+00 \\ b_3 = -3.479020E-05 & c_3 = -7.498253E-05 & d_3 = 4.856615E-05 \\ b_4 = 1.344218E-07 & c_4 = 1.240022E-07 & d_4 = 1.661670E-07 \\ b_5 = 4.704199E-03 & c_5 = 4.745292E-02 & d_5 = -2.080714E-01 \end{array}$$

Model when `trans` flag is `T` (With transposing).
Features: $m^2, n^2, mn, m^2n$

$$y_{pred} = \begin{cases} b_1 \cdot m^2 + b_2 \cdot n^2 + b_3 \cdot mn + b_4 \cdot m^2n + b_5 & \text{if } m^2n < 2 \times 10^5 \\ c_1 \cdot m^2 + c_2 \cdot n^2 + c_3 \cdot mn + c_4 \cdot m^2n + c_5 & \text{if } 2 \times 10^5 \leq m^2n < 2 \times 10^6 \\ d_1 \cdot m^2 + d_2 \cdot n^2 + d_3 \cdot mn + d_4 \cdot m^2n + d_5 & \text{if } m^2n \geq 2 \times 10^6 \end{cases}$$

Where,

$$\begin{array}{lll} b_1 = 1.295043E-07 & c_1 = 1.003128E-07 & d_1 = 1.431553E-07 \\ b_2 = -2.646978E-23 & c_2 = 0.000000E+00 & d_2 = 0.000000E+00 \\ b_3 = -1.913967E-05 & c_3 = 2.392070E-05 & d_3 = 2.498099E-04 \\ b_4 = 1.295043E-07 & c_4 = 1.003128E-07 & d_4 = 1.431553E-07 \\ b_5 = 3.688210E-03 & c_5 = 1.351287E-02 & d_5 = -5.024207E-01 \end{array}$$

Figure 11 shows the graphs only for the no transpose case as it is not very different from the one with transposing.

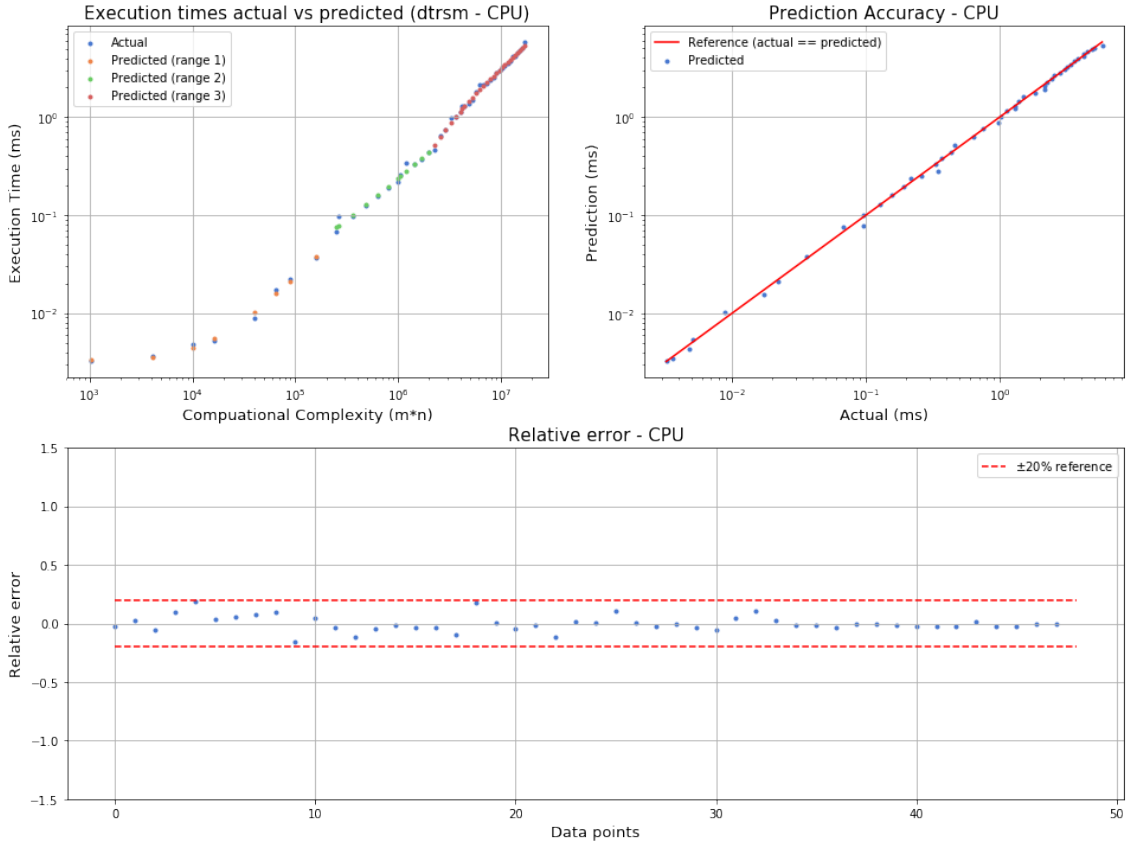



Figure 11: Accuracy of model `dtrsm` - CPU

| Metric | Value (`trans = N`) | Value (`tans = T`) |
|---|---|---|
| Mean absolute percentage error | 5.73 % | 4.76 |
| Max absolute percentage error | 47.66 % | 18.76 |
| Outliers | 4.51 % | 0.00 |

Table 11: `dtrsm` CPU model accuracy

### B.4.2 GPU Model

Model when `trans` flag is `N` (No transposing).
Features: $m, n, m^2n$

$$y_{pred} = \Big\{ b_1 \cdot m + b_2 \cdot n + b_3 \cdot m^2 n + b_4$$

Where,

$$b_1 = 9.605335E - 05$$
$$b_2 = 0.000000E + 00$$
$$b_3 = -6.042791E - 10$$
$$b_4 = 1.259568E - 02$$

Model when `trans` flag is `T` (With transposing).
Features: $m, n, m^2n$

$$y_{pred} = \Big\{ b_1 \cdot m + b_2 \cdot n + b_3 \cdot m^2 n + b_4$$

Where,

$$b_1 = 1.141587E - 04$$
$$b_2 = 0.000000E + 00$$
$$b_3 = 6.327967E - 09$$
$$b_4 = 1.425764E - 02$$

Figure 12 shows the graphs only for the no transpose case as it is not very different from the one with transposing.

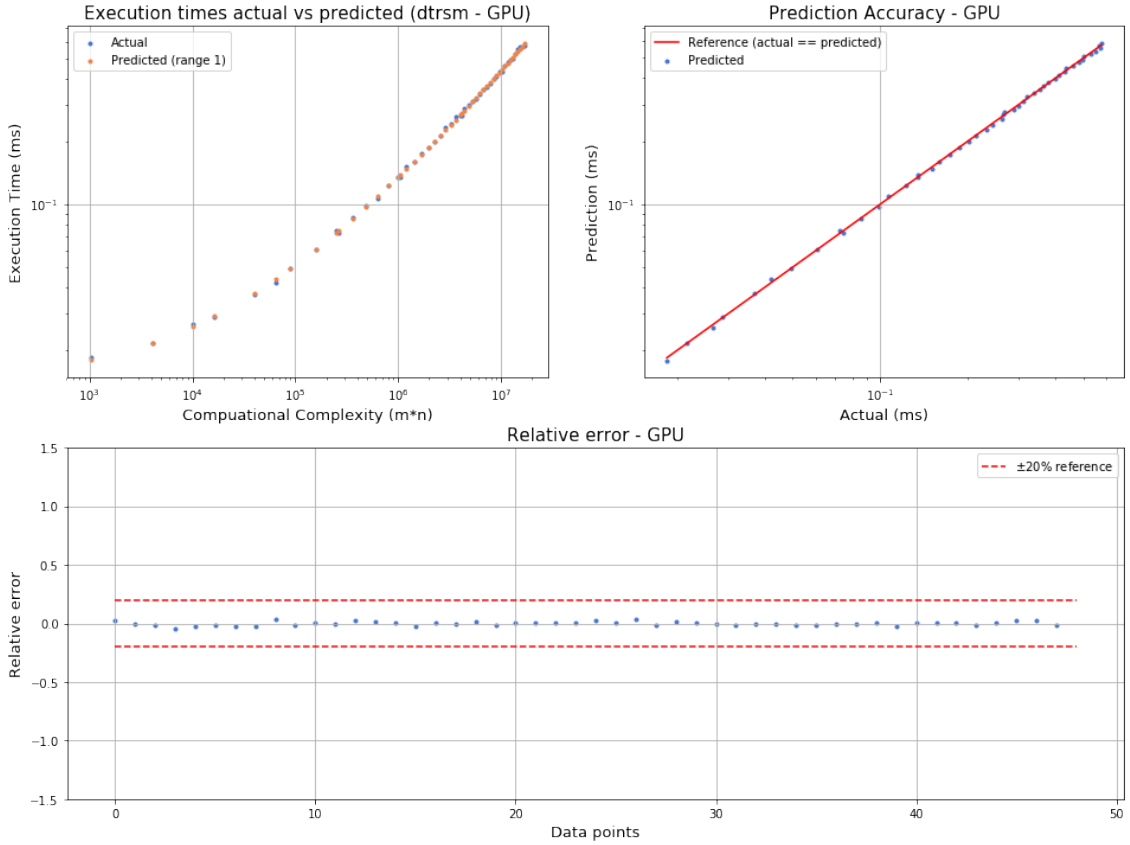



Figure 12: Accuracy of model `dtrsm` - GPU

| Metric | Value (`trans = N`) | Value (`tans = T`) |
|---|---|---|
| Mean absolute percentage error | 1.35 % | 1.38 |
| Max absolute percentage error | 9.69 % | 4.56 |
| Outliers | 0.00 % | 0.00 |

Table 12: `dtrsm` GPU model accuracy

## B.5 Triangular matrix copy - `dlacpy`

| Attribute | Values |
|---|---|
| Size range | Powers of 2 in range $[32, ..., 4096]$ and multiples of 100 in range $[100, 200..., 4000]$ |
| Number of measurements | 296* and 48 measurements for CPU and GPU respectively. |
| Dimension arguments | $m$ |
| Complexity | $O(m^2)$ |

Table 13: `dlacpy` test attributes

* In the measurements from the application, it was observed that even for the same size $m$, the execution time slightly increased when the leading dimension of the destination matrix (`ldb`) increased (`dlacpy` copies the $L_c$ matrix of size $n \times n$ to the $L$ and $U$ matrices which increase in size in each iteration, having the dimensions $(n + nidxs) \times (n + nidxs)$). In order to capture this effect when making measurements for modeling, for each size $m$, 6 `ldb` values were calculated to span the range $[m, 2m]$. Thus, for the CPU 296 measurements were made.

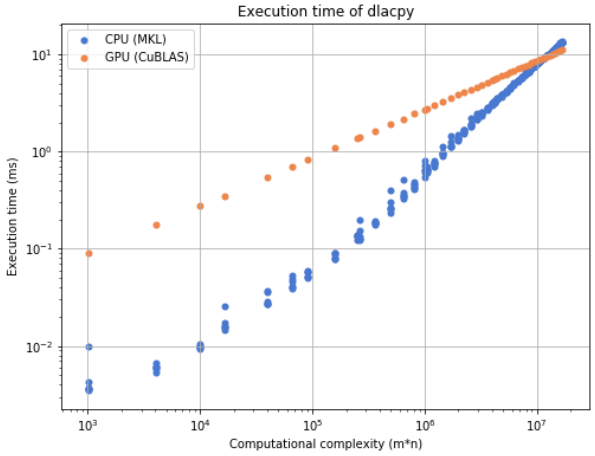



Figure 13: Execution time measurements for `dlacpy`

### B.5.1 CPU Model

Features: $ldb, m^2$

$$y_{pred} = \begin{cases} b_1 \cdot ldb + b_2 \cdot m^2 + b_3 & \text{if } m^2 < 10^5 \\ c_1 \cdot ldb + c_2 \cdot m^2 + c_3 & \text{if } 10^5 \leq m^2 < 2 \times 10^6 \\ d_1 \cdot ldb + d_2 \cdot m^2 + d_3 & \text{if } m^2 \geq 2 \times 10^6 \end{cases}$$

Where,

$$\begin{aligned} b_1 &= 9.014109E-06 & c_1 &= 2.266375E-05 & d_1 &= 3.348356E-05 \\ b_2 &= 5.603457E-07 & c_2 &= 6.661555E-07 & d_2 &= 7.979377E-07 \\ b_3 &= 3.319108E-03 & c_3 &= -5.591714E-02 & d_3 &= -1.809488E-01 \end{aligned}$$

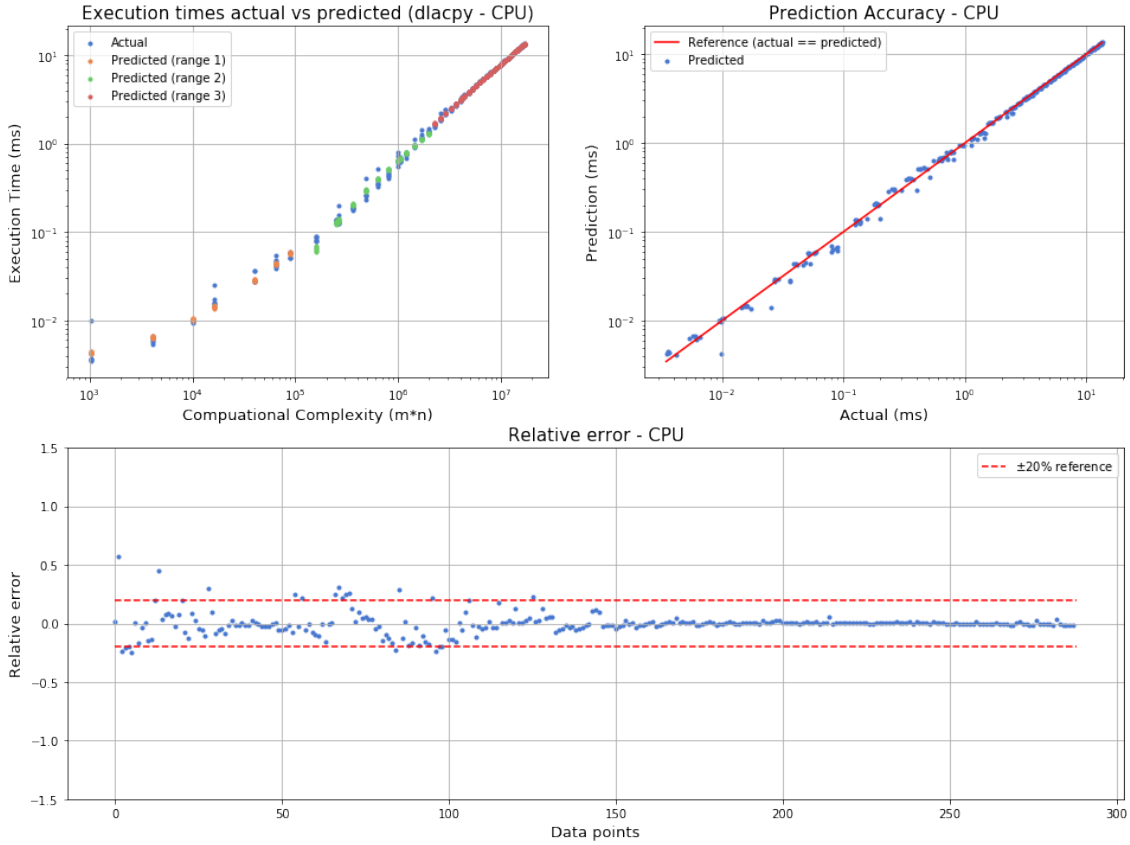



Figure 14: Accuracy of model `dlacpy` - CPU

| Metric | Value |
|---|---|
| Mean absolute percentage error | 5.12 % |
| Max absolute percentage error | 57.48 % |
| Outliers | 6.6 % |

Table 14: `dlacpy` CPU model accuracy

### B.5.2 GPU Model

Features: $m$

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \end{cases}$$

Where,

$$b_1 = 2.720431E - 03$$
$$b_2 = 3.559786E - 03$$

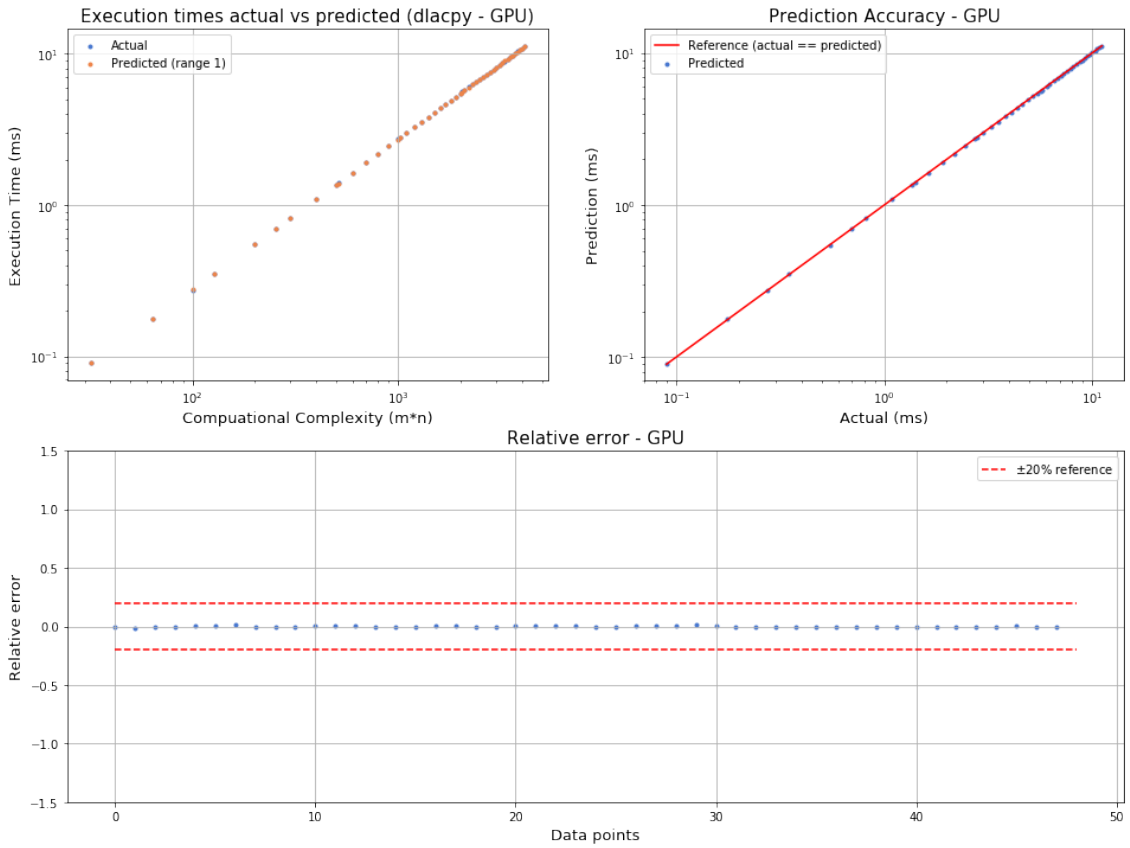



Figure 15: Accuracy of model `dlacpy` - GPU

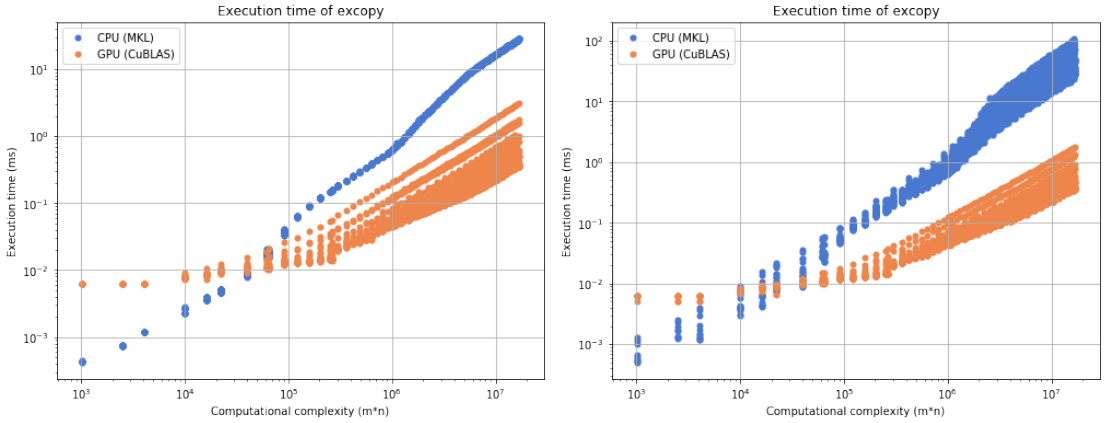
| Metric | Value |
|---|---|
| Mean absolute percentage error | 0.28 % |
| Max absolute percentage error | 1.34 % |
| Outliers | 0.0 % |

Table 15: `dlacpy` GPU model accuracy

## B.6 Extended copies - `excopy` (`mkl_domatcopy` and `cublasDgeam`)

| Attribute | Values |
|---|---|
| Size range | Initial values selected as powers of 2 in range $[32, ..., 4096]$ and multiples of 50 in range $[50, 1000..., 400]$. Then all dimension combinations that result in the same computation complexity was calculated keeping the sizes within $[1, 10000]$ range. |
| Number of measurements | 3093 measurements for both with and without input transposed cases. |
| Dimension arguments | $m, n$ |
| Complexity | $O(mn)$ |
| Thinness condition | `m*r < n || n*r < m` |

Table 16: `excopy` test attributes




(a) `trans = N` (b) `trans = T`

Figure 16: Execution time measurements for `excopy`

### B.6.1 CPU Model

Model when `trans` flag is `N` (No transposing).
Features: $m, n, m^2, n^2, mn$

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \cdot n + b_3 \cdot m^2 + b_4 \cdot n^2 + b_5 \cdot mn + b_6 & \text{if } mn < 3 \times 10^4 \\ c_1 \cdot m + c_2 \cdot n + c_3 \cdot m^2 + c_4 \cdot n^2 + c_5 \cdot mn + c_6 & \text{if } 3 \times 10^4 \leq mn < 10^6 \\ d_1 \cdot m + d_2 \cdot n + d_3 \cdot m^2 + d_4 \cdot n^2 + d_5 \cdot mn + d_6 & \text{if } 10^6 \leq mn < 3 \times 10^6 \\ f_1 \cdot m + f_2 \cdot n + f_3 \cdot m^2 + f_4 \cdot n^2 + f_5 \cdot mn + f_6 & \text{if } mn \geq 3 \times 10^6 \end{cases}$$

$b_1 = -2.729217E-08$ $c_1 = -5.551119E-09$ $d_1 = -3.591668E-08$ $f_1 = 2.805059E-08$
$b_2 = -2.860477E-08$ $c_2 = -1.158916E-08$ $d_2 = -3.701099E-08$ $f_2 = 2.023917E-08$
$b_3 = 5.939399E-13$ $c_3 = -4.811315E-15$ $d_3 = 1.659772E-14$ $f_3 = -2.934457E-15$
$b_4 = 6.880051E-13$ $c_4 = 3.735659E-15$ $d_4 = 1.631665E-14$ $f_4 = -2.014148E-15$
$b_5 = 2.122399E-07$ $c_5 = 6.406802E-07$ $d_5 = 1.407590E-06$ $f_5 = 1.803279E-06$
$b_6 = 2.571306E-04$ $c_6 = -1.858215E-02$ $d_6 = -8.570731E-01$ $f_6 = -1.566771E+00$

Model when `trans` flag is `T` (With transposing).
Features: $m, n, mn$

Let $thin = (m * r < n) || (n * r < m)$ be the condition for thin matrices, where $r = 10000$ is the thinness ratio as defined in Section 5.1.

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \cdot n + b_3 \cdot m^2 + b_4 \cdot n^2 + b_5 \cdot mn + b_0 & \text{if } mn < 3 \times 10^4 \text{ and } thin = False \\ \\ c_1 \cdot m + c_2 \cdot n + c_3 \cdot m^2 + c_4 \cdot n^2 + c_5 \cdot mn + c_0 & \text{if } 3 \times 10^4 \leq mn < 10^6 \text{ and } thin = True \\ d_1 \cdot m + d_2 \cdot n + d_3 \cdot m^2 + d_4 \cdot n^2 + d_5 \cdot mn + d_0 & \text{if } 3 \times 10^4 \leq mn < 10^6 \text{ and } thin = False \\ \\ f_1 \cdot m + f_2 \cdot n + f_3 \cdot m^2 + f_4 \cdot n^2 + f_5 \cdot mn + f_0 & \text{if } 10^6 \leq mn < 3 \times 10^6 \text{ and } thin = True \\ g_1 \cdot m + g_2 \cdot n + g_3 \cdot m^2 + g_4 \cdot n^2 + g_5 \cdot mn + g_0 & \text{if } 10^6 \leq mn < 3 \times 10^6 \text{ and } thin = False \\ \\ p_1 \cdot m + p_2 \cdot n + p_3 \cdot m^2 + p_4 \cdot n^2 + p_5 \cdot mn + p_0 & \text{if } mn \geq 3 \times 10^6 \text{ and } thin = True \\ q_1 \cdot m + q_2 \cdot n + q_3 \cdot m^2 + q_4 \cdot n^2 + q_5 \cdot mn + q_0 & \text{if } mn \geq 3 \times 10^6 \text{ and } thin = False \end{cases}$$

Where,

$b_1 = 9.408887E-07$ $c_1 = 9.976481E-07$ $d_1 = 7.506850E-07$ $f_1 = -9.977038E-08$
$b_2 = 6.543644E-07$ $c_2 = 7.363898E-07$ $d_2 = -6.604001E-07$ $f_2 = -4.407489E-07$
$b_3 = 1.923671E-11$ $c_3 = -3.631078E-13$ $d_3 = -8.533586E-12$ $f_3 = 9.185349E-14$
$b_4 = 5.492603E-11$ $c_4 = 2.305160E-13$ $d_4 = -1.442084E-11$ $f_4 = 5.170897E-13$
$b_5 = 3.197080E-07$ $c_5 = 5.061793E-07$ $d_5 = 8.794052E-07$ $f_5 = 1.691632E-06$
$b_6 = 1.698319E-04$ $c_6 = -4.869950E-03$ $d_6 = -2.437491E-02$ $f_6 = -9.950919E-01$

$g_1 = 7.098822E-07$ $p_1 = -2.113541E-06$ $q_1 = 5.793630E-05$
$g_2 = -1.682908E-05$ $p_2 = -2.872194E-06$ $q_2 = -5.801809E-05$
$g_3 = -5.820694E-11$ $p_3 = 1.353055E-13$ $q_3 = -2.332556E-10$
$g_4 = 3.384809E-11$ $p_4 = 2.713573E-13$ $q_4 = 7.456757E-11$
$g_5 = 3.028687E-06$ $p_5 = 2.288364E-06$ $q_5 = 3.796003E-06$
$g_6 = -2.053047E+00$ $p_6 = -8.427003E-01$ $q_6 = -3.766503E+00$

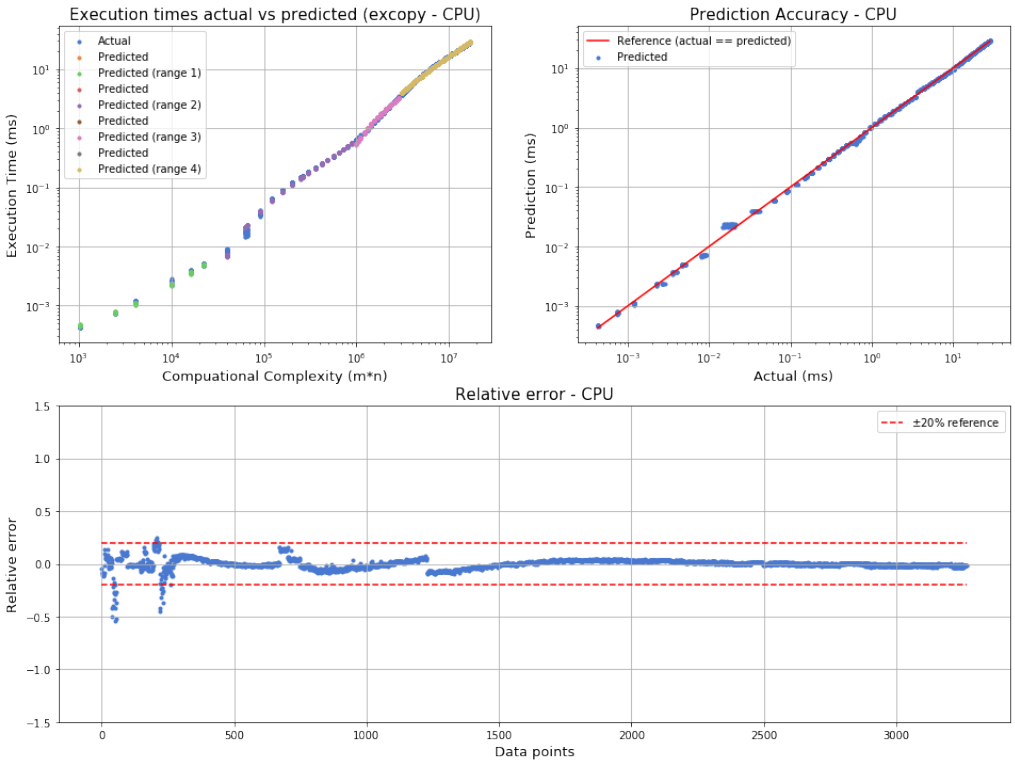



Figure 17: Accuracy of model without transpose `excopy` - CPU

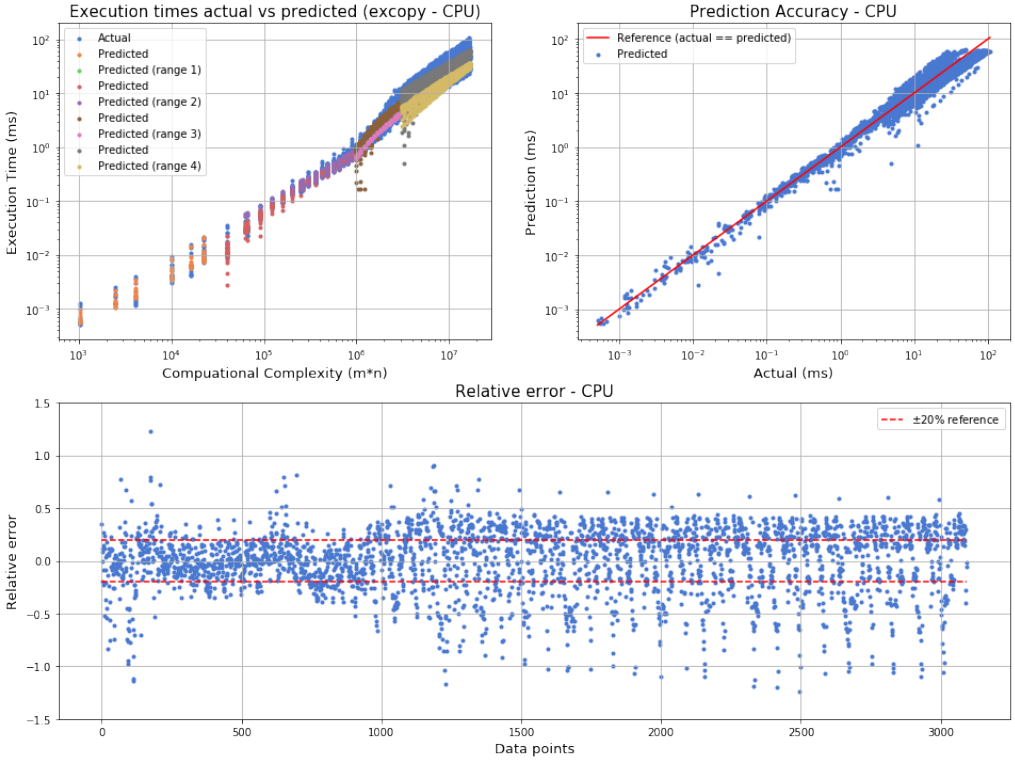



Figure 18: Accuracy of model with transpose `excopy` - CPU

| Metric | Value (`trans = N`) | Value (`tans = T`) |
|---|---|---|
| Mean absolute percentage error | 3.15 | 24.18 % |
| Max absolute percentage error | 54.29 | 151.29 % |
| Outliers | 0.88 | 50.76 % |

Table 17: `excopy` CPU model accuracy

### B.6.2 GPU Model

For the GPU, the same model was used for transposed and non transposed cases.

Features: $m, n, mn$

$$y_{pred} = \begin{cases} b_1 \cdot m + b_2 \cdot n + b_3 \cdot mn + b_4 & \text{if } mn < 10^4 \\ c_1 \cdot m + c_2 \cdot n + c_3 \cdot mn + c_4 & \text{if } 10^4 \leq mn < 3 \times 10^5 \\ d_1 \cdot m + d_2 \cdot n + d_3 \cdot mn + d_4 & \text{if } mn \geq 3 \times 10^5 \end{cases}$$

Where,

$$\begin{aligned} b_1 &= -1.208197E-07 & c_1 &= 8.095752E-08 \\ b_2 &= -4.358342E-07 & c_2 &= 1.559921E-07 \\ b_3 &= 2.616894E-08 & c_3 &= 2.342008E-08 \\ b_4 &= 6.016339E-03 & c_4 &= 8.330997E-03 \end{aligned}$$

$$\begin{aligned} d_1 &= 9.679107E-08 \\ d_2 &= 1.415087E-07 \\ d_3 &= 2.163352E-08 \\ d_4 &= 1.855496E-02 \end{aligned}$$

| Metric | Value (`trans = N`) | Value (`tans = T`) |
|---|---|---|
| Mean absolute percentage error | 7.74 % | 7.19 |
| Max absolute percentage error | 65.15 % | 45.03 |
| Outliers | 5.51 % | 2.62 |

Table 18: `excopy` GPU model accuracy

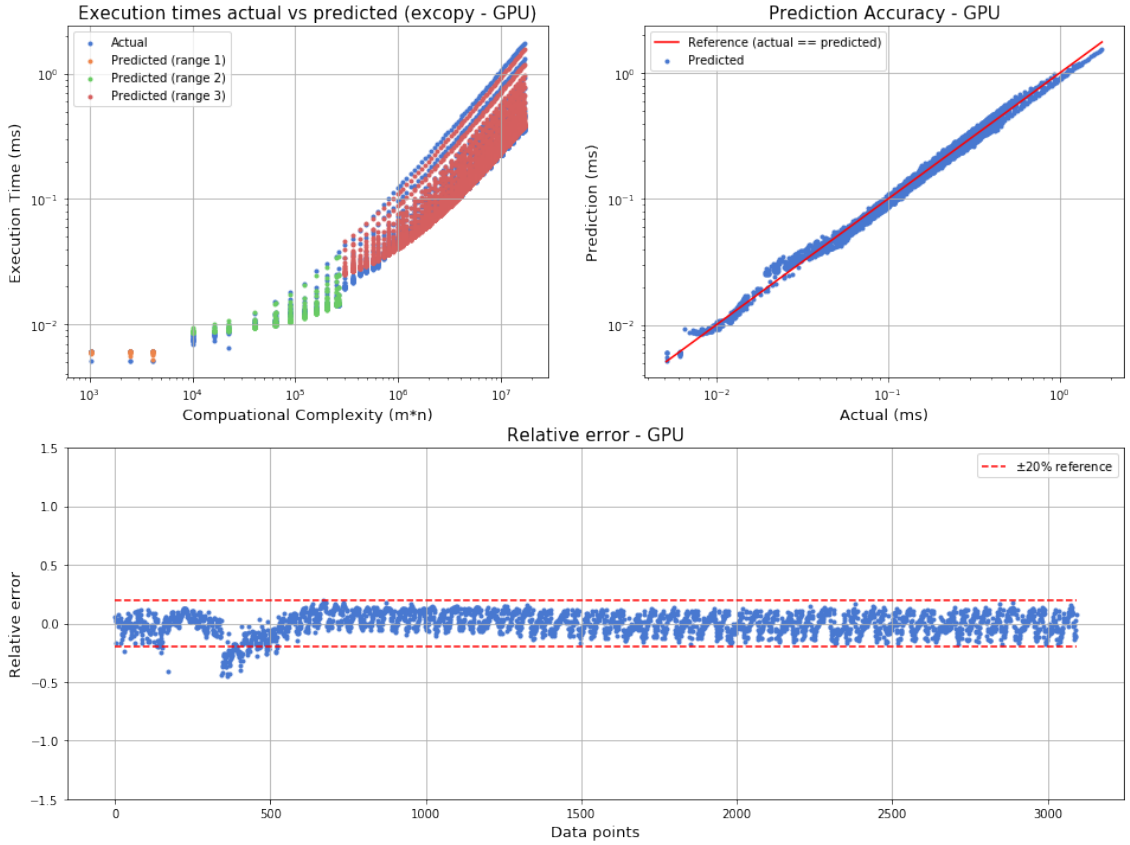



Figure 19: Accuracy of model `excopy` - GPU