Maximizing GPU Utilization in High-Performance Sensor Signal Processing

Master's Thesis in Embedded Electronic System Design

David Ernstig

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Maximizing GPU Utilization in High-Performance Sensor Signal Processing

David Ernstig

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Maximizing GPU Utilization in High-Performance Sensor Signal Processing

David Ernstig

© David Ernstig, 2024.

Supervisor: Lena Peterson, Department of Computer Science and Engineering
Advisor: Anton Karlsson, Jacob Lundberg, Anders Åhlander, SAAB
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2024

Maximizing GPU Utilization in High-Performance Sensor Signal Processing

David Ernstig
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The purpose of this thesis work was to investigate the necessary steps to increase the utilization of a GPU and to eventually establish if the GPU is suitable as a processing unit in a challenging sensor signal processing chain. This was done by implementing the 2D-ESPRIT algorithm on a Quadro RTX 6000 GPU and analyzing the time that the GPU required to finish the necessary calculations. Then, the code was improved to increase computation speed and lower the execution time of the program.

The first version of the implementation achieved a calculated GPU utilization of 0.82%. It was later found that the utilization of the GPU could be increased by dividing the data set into threads and CUDA streams which were computed concurrently. This increased the calculated utilization to 2.94%. Further, the main bottleneck of the 2D-ESPRIT algorithm was the singular value decomposition calculation, which if improved, would increase the utilization of the GPU further.

Keywords: Computer science, engineering, master thesis, ESPRIT, GPU, RADAR, sensors, signal processing.

Acknowledgements

I want to extend my thanks to my supervisors Anton Karlsson, Anders Åhlander, and Jacob Lundberg at SAAB for their input and assistance in progressing this project. I would also like to thank my supervisor Lena Peterson at Chalmers for her input and massive help with improving the writing of this master thesis report.

David Ernstig, Gothenburg, 2024-10-13

Contents

Acronyms **xiii**

1 Introduction **1**
1.1 Problem description . . . 1
1.2 Related works . . . 2
1.2.1 GPU usage . . . 2
1.2.2 GPU usage in other contexts . . . 3
1.2.3 ESPRIT . . . 3
1.2.4 Radar signal processing . . . 3
1.3 Goals . . . 4
1.4 Objectives . . . 4
1.5 Scope and limitations . . . 4
1.5.1 Thesis outline . . . 5

2 Theory **7**
2.1 Radar Background and Theory . . . 7
2.1.1 Radar fundamentals . . . 7
2.2 Data gathering . . . 8
2.2.1 Sampling . . . 8
2.3 ESPRIT . . . 10
2.4 GPUs . . . 14
2.5 CUDA C/C++ . . . 16
2.5.1 Programming in CUDA C/C++ . . . 16
2.5.2 CUDA streams . . . 17
2.5.3 Supported libraries . . . 19
2.6 Utilization . . . 19

3 Workflow **21**
3.1 Initial ESPRIT version . . . 21
3.2 Second ESPRIT version . . . 22
3.3 Third ESPRIT version . . . 23

4 Results **25**
4.1 First version . . . 26
4.1.1 FP32 . . . 26
4.1.2 FP64 . . . 30

4.1.3 Wall-clock times for the initial ESPRIT version . . . 34
4.2 Second version . . . 34
4.2.1 One thread with 500 CUDA streams . . . 34
4.2.2 Two threads with 250 CUDA streams each . . . 38
4.2.3 Wall-clock times for the second ESPRIT version . . . 42
4.3 Third version . . . 43
4.3.1 One thread with 500 CUDA streams . . . 43
4.3.2 Five threads with 100 CUDA streams per thread . . . 48
4.3.3 Wall-clock times for the third ESPRIT version . . . 52
4.4 Speed-up . . . 53
4.5 Utilization . . . 54

5 Discussion 57
5.1 Version results . . . 57
5.2 Utilization . . . 58
5.3 Issues . . . 59
5.4 Improvements and future work . . . 60
5.5 Societal, ethical and ecological aspects . . . 62

6 Conclusion 63

Bibliography 65

A Appendix 67
A.1 Version 2, 1 threads, 500 streams . . . 67
A.1.1 FP64, size 64 . . . 67
A.1.2 FP64, size 360 . . . 69
A.2 Version 2, 2 threads, 250 streams . . . 71
A.2.1 FP64, size 64 . . . 71
A.2.2 FP64, size 360 . . . 73
A.3 Version 2, 5 threads, 100 streams . . . 75
A.3.1 FP32, size 64 . . . 75
A.3.2 FP32, size 360 . . . 77
A.3.3 FP64, size 64 . . . 79
A.3.4 FP64, size 360 . . . 81
A.4 Version 2, 10 threads, 50 streams . . . 83
A.4.1 FP32, size 64 . . . 83
A.4.2 FP32, size 360 . . . 85
A.4.3 FP64, size 64 . . . 87
A.4.4 FP64, size 360 . . . 89
A.5 Version 2, 20 threads, 25 streams . . . 91
A.5.1 FP32, size 64 . . . 91
A.5.2 FP32, size 360 . . . 93
A.5.3 FP64, size 64 . . . 95
A.5.4 FP64, size 360 . . . 97
A.6 Version 2, 25 threads, 20 streams . . . 99
A.6.1 FP32, size 64 . . . 99

A.6.2 FP32, size 360 101
A.6.3 FP64, size 64 103
A.6.4 FP64, size 360 105
A.7 Version 3, 1 thread, 500 streams 107
A.7.1 FP64, size 64 107
A.7.2 FP64, size 360 109
A.8 Version 3, 2 threads, 250 streams 111
A.8.1 FP32, size 64 111
A.8.2 FP32, size 360 113
A.8.3 FP64, size 64 115
A.8.4 FP64, size 360 117
A.9 Version 3, 5 threads, 100 streams 119
A.9.1 FP64, size 64 119
A.9.2 FP64, size 360 121
A.10 Version 3, 10 threads, 50 streams 123
A.10.1 FP32, size 64 123
A.10.2 FP32, size 360 125
A.10.3 FP64, size 64 127
A.10.4 FP64, size 360 129
A.11 Version 3, 20 threads, 25 streams 131
A.11.1 FP32, size 64 131
A.11.2 FP32, size 360 133
A.11.3 FP64, size 64 135
A.11.4 FP64, size 360 137
A.12 Version 3, 25 threads, 20 streams 139
A.12.1 FP32, size 64 139
A.12.2 FP32, size 360 141
A.12.3 FP64, size 64 143
A.12.4 FP64, size 360 145

Acronyms

BLAS	Basic linear algebraic subprograms
CPU	Central processing unit
CUDA	Compute unified device architecture
DOA	Direction of arrival
ESPRIT	Estimation of signal parameters via rotational invariance techniques
FFT	Fast Fourier transform
FinFET	Fin field-effect transistor
FLOPS	Floating point operations per second
FMCW	Frequency modulated continuous wave
FP	Floating point
FPGA	Field programmable gate array
GPU	Graphics processing unit
INT	Integer
kB	Kilobytes
L1	Level 1 cache
L2	Level 2 cache
MMEMP	Modified matrix enhancement and matrix pencil
MUSIC	Multiple signal classification
PRF	Pulse repetition frequency
PRI	Pulse repetition interval
SM	Streaming multiprocessor
STAP	Space time adaptive processing
SVD	Singular value decomposition
TPC	Texture processing cluster

1
Introduction

In modern sensor signal processing, high efficiency of embedded systems is required for the ever-increasing demand for faster computing. For antenna arrays used in radar technology with thousands of small digital sensors, each sampling at a rate of several MHz, the amount of data gathered by each element each second is very high which further increases the need for computing power. The data is collected in batches that stream through the digital processing chain. Thus, it is important that the data processing has low latency. However, this efficiency generally comes at the expense of high power consumption. Due to factors such as cost and available space in a whole system, it is also generally desirable that these systems are small. Creative solutions and technologies are required to find a satisfactory trade-off between the computing power, power consumption, the physical space required for the processing unit and the cost for any given implementation.

In this thesis, we explore whether graphics processing units (GPUs) are suitable for handling the huge amount of data that multi-channel sensor signal processing entails. GPUs have very many small cores that work in parallel which enables high efficiency when dealing with embarrassingly parallel problems [1]. This investigation will be done by implementing a direction of arrival (DOA) algorithm, called estimation of signal parameters via rotational invariance techniques (ESPRIT) [2], on a GPU and attempt to maximize its performance. The algorithm will be implemented in compute unified device architecture (CUDA) C/C++: a software platform that enables GPU programming provided by NVIDIA [3]. ESPRIT serves as a challenging representative algorithm, meaning it will be used by us to investigate if other similar algorithms for the same application can be efficiently implemented on a GPU and what the GPU can handle. Bottlenecks and areas of improvement are investigated in the implementation of the algorithm to improve the performance and utilization of the GPU. The thesis is linked with industry at the company SAAB situated in Gothenburg.

Detailed below is a more concrete problem formulation, related works, the goals and objectives of the thesis and the scope and limitations of the thesis.

1.1 Problem description

The problem is that large amounts of data are collected for radar applications and must be processed efficiently. The data is also collected in batches that are streaming through

the digital processing chain. For this reason, the data must be processed fast with a low-latency requirement so that other applications in the radar system, which are dependent on the data, can maintain proper functionality.

Therefore, we want to investigate if GPUs are suitable for dealing with the challenge of handling high-intensity data processing with high throughput. Further, we want to see if the GPU can be used for radar signal processing applications with a representative algorithm. We also investigate how the utilization of the GPU can be maximized to handle as much data as possible while maintaining proper functionality.

By addressing these problems, further insight can be gained about the capabilities of the GPU. The answers may also give insight into what other algorithms can be implemented, how parameters of the data sets can be changed while still meeting performance requirements, and how powerful the GPU has to be while maintaining good functionality and precision. This insight into how powerful the GPU has to be can affect the selection of GPUs used in future projects.

1.2 Related works

Previous work has been done using GPUs for parallel applications. The following sections contain previous research on GPUs, deep learning, radar signal processing, the ESPRIT algorithm, and other related contexts.

1.2.1 GPU usage

The GPU is very promising for handling parallel problems. It was found that GPUs were suitable for applications with high arithmetic intensity in parallel problems [4]. Here, the authors compared the performance of a central processing unit (CPU) to a GPU and found that the GPU in some cases achieved up to 100x better performance compared to the CPU in raw calculation power. The limiting factor was the memory transfer bandwidth, which lowered the performance of the GPU substantially. Although the thesis is now quite dated with the improvements in technology since then, it showcases that the GPU has much to offer in applications where parallelism can be implemented for calculations with high arithmetic intensity.

Another master thesis investigated whether a field programmable gate array (FPGA) was suitable for implementing space time adaptive processing (STAP) and also compared the FPGA to the GPU [5]. It was found that the FPGA was insufficient in meeting the requirements of handling the radar data in the range of 1GFLOPS-50TFLOPS (giga/tera floating point operations per second). The processing power of 300 units of the Virtex-7 FPGA device had to be used which is unrealistic for power, area, and cost reasons. On power efficiency, it was found that the FPGA performed better than the NVIDIA GTX 260 GPU. However, no conclusions could be drawn about whether the FPGA or the GPU performed better considering both power consumption and performance. This was due

to a lack of information on the power consumption of STAP on the GPU where only utilization could be found.

1.2.2 GPU usage in other contexts

In [6], a CPU and a GPU were compared in deep learning applications. It was found that when comparing the CPU to the GPU in the time taken for computations, the GPU outperformed the CPU in most calculations. The computation time gap also increased with batch size, further highlighting the strength of the GPU when handling large amounts of data that can be processed in parallel.

1.2.3 ESPRIT

ESPRIT is an algorithm that was first proposed in 1985 as an algorithm for estimating signal parameters in noisy environments [7]. The algorithm utilizes a rotational invariance in signal subspaces produced by a sensor array which consists of sensor elements displaced by a known displacement factor in one dimension. Considering one planar wave produced by a source point that impacts a sensor array, the wave will coincide with each sensor element with a phase shift. By finding this phase shift between the elements, one can ascertain the DOA of the source point.

The ESPRIT algorithm has previously been used for radar applications. An implementation of a 2D version of ESPRIT that calculates both range and velocity for multiple targets in frequency modulated continuous wave (FMCW) radar shows great promise in estimating the targets' parameters [8]. It had greater accuracy in estimating both velocity and range for multiple targets compared to other algorithms such as fast Fourier transform (FFT), multiple signal classification (MUSIC), and Modified Matrix Enhancement and Matrix Pencil (MMEMP). However, the article does not investigate the computation effort of the algorithm.

1.2.4 Radar signal processing

Other alternatives to ESPRIT have been used for handling radar data on GPUs. ESPRIT and MUSIC were compared to a recently proposed algorithm, MARS, which builds on a maximum likelihood estimation for achieving high resolution when measuring DOAs [9]. MARS was implemented using the CUDA toolkit on an NVIDIA Tesla V100 GPU and achieved a prediction of several DOAs within one-degree precision while having a computational time under one millisecond. Both MUSIC and ESPRIT had worse precision and higher computation time compared to MARS. MUSIC was also implemented on a GPU. The article does not specify on what platform ESPRIT was implemented. Therefore, it is difficult to draw any conclusions on ESPRIT's viability on GPUs. However, the article shows that GPUs can be used for handling radar data to accelerate computations.

1.3 Goals

With the above-mentioned works on the GPU, ESPRIT, and other related contexts, we arrive at the goal of this thesis.

The goal of this thesis is to implement the 2D-ESPRIT algorithm on NVIDIAs Quadro RTX 6000 GPU and investigate how high utilization can be achieved with radar data from a simulated antenna array. We try to increase the utilization of the GPU by scaling up the amount of data handled by the algorithm and by trying to exploit the parallel nature of the GPU. The amount of data is increased by varying in-signals, sensor elements, and samples taken.

The purpose of the thesis is to identify the limits of the GPU to explore if the GPU is a viable option when selecting processors for radar signal processing applications.

1.4 Objectives

The main objectives of this thesis are as follows:

- Implement an initial 2D version of ESPRIT based on a MATLAB model.
  - Increase the amount of data the GPU must handle before it fails performance criteria.
  - Identify bottlenecks and areas of improvement of the implemented algorithm.
- Improve the previously implemented ESPRIT algorithm by:
  - Attempting to resolve previously identified bottlenecks.
  - Improving memory management and transfer time.
  - Increasing parallelism where possible.
- Investigate the differences between the initial and improved version of the implemented algorithm and monitor the GPU utilization, latency and data size.
- Identify limitations of the GPU in the implementation.

This workflow is conducted iteratively, where gradual improvements to the algorithm are tested against a previously failed test case to identify the new limit. This is done to try to push the GPU to its limits.

1.5 Scope and limitations

This thesis does not cover whether one algorithm or another is best for handling radar data. ESPRIT is assumed to be a suitable algorithm for processing data in radar appli-

cations. Investigation of how to improve ESPRIT as an algorithm is not done.

This thesis does not use collected radar data but uses simulated data instead. In practice, this does not have much impact on the experiments since the data is the same size as it would have been if it had been collected from an actual antenna array. It actually is beneficial to simulate with a known data set, because then the output of the system can be predicted.

CUDA C/C++ is the language used for implementing the algorithm. Therefore, this thesis does not consider other languages, such as OpenGL.

Due to time constraints, many functions required for the algorithm will not be implemented from scratch but we will use functions already implemented from established libraries. Also, the solver for the eigenvalues used in the implemented algorithm is used on the CPU and not on the GPU. This is due to a solver not already existing for the GPU for this function. However, the function is not the heaviest part of the algorithm and will not affect the timing results to any large extent.

1.5.1 Thesis outline

In chapter two of this thesis, the relevant theory to understand the thesis work is explained. In the third chapter is a description of the workflow of the thesis work. In chapter four, the results of the work is presented which is followed by a discussion of the results in chapter five. Finally, the conclusion of the thesis is in chapter six. Additional results, which were not found in the results section, can be found in Appendix A.

2

Theory

In this chapter, the relevant theory for the thesis project is presented. This includes an overview of how radar works and how the 2D-ESPRIT algorithm can use the data gathered by radar sensors. Moreover, information on CUDA C/C++ and existing libraries used in the implementation of ESPRIT is shared. The Quadro RTX 6000 GPU specifications and how GPUs function will also be shared. Finally, there will be a section on how an estimate of the GPU utilization is calculated.

2.1 Radar Background and Theory

The radar concept was originally demonstrated in the late 19th century when Heinrich Hertz demonstrated that electromagnetic waves could be reflected by a surface [10, ch. 1.1.]. It was later researched further during World War 2, when it was developed to survey the airspace for flying aircrafts. Radar as a technology has since transitioned from analog to digital signal processing, enabling more refined data processing techniques that provide more accurate readings of the airspace. However, this has also increased the required computation power immensely.

2.1.1 Radar fundamentals

Radar works by transmitting an electromagnetic wave that later reflects on an object's surface. This reflected wave is received by an antenna sensor and is used to get information about the distance between the antenna and the object using

$$R = \frac{c \cdot t}{2}, \tag{2.1}$$

where c is the speed of light and t is the time between the transmission and the reception of a reflected pulse [11].

The power of a reflected signal is determined by

$$P_r = \frac{P_t G A_e \sigma}{(4\pi)^2 R^4} \tag{2.2}$$

where P_t is the transmitted power, G is the gain of the antenna, A_e is the aperture size of the antenna receiving the echo, σ is the target cross section and R is the distance to the

reflecting object. Worthy of note is that reflecting power reduces with R^4, meaning that objects farther away reflect much less power and are thus harder to detect. Furthermore, the reflected signal strength is diminished if the cross-section of an object is small.

In most radar systems, the transmitting and receiving antenna is the same physical antenna. Such a configuration is called a monostatic radar. The pulse is first transmitted and then the antenna listens for a reflecting echo. The frequency of the pulse transmissions is called the pulse repetition frequency (PRF). If the rate is high, the resolution of the radar is increased due to more frequent pulse transmissions. However, targets at a longer distance than the intended listening range can still be detected. In such a case, the reflecting wave from these distant targets can be received in a later pulse interval, leading to range ambiguity. An illustration of how a distance can be perceived versus its actual distance is found in Figure 2.1.

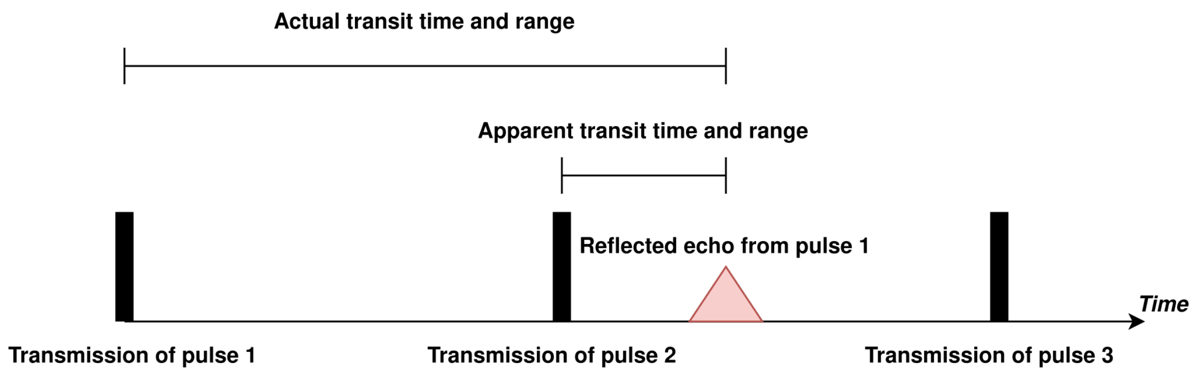


Figure 2.1: Perceived transmit time vs actual transmit time.

The monostatic radar system can be extended to contain multiple antenna sensor elements that listen for echoes in the airspace. The elements can be placed in various configurations depending on the intended use case of the radar system, which enables different signal processing techniques. A common configuration is a 1-D array.

2.2 Data gathering

To analyze the reflecting pulse, samples must be obtained. These are modeled in a three-dimensional matrix for an antenna array, with the axes fast-time, slow-time, and antenna elements. These are explained in further detail below.

2.2.1 Sampling

To find the distance to a reflecting target's surface from a single antenna element, samples are collected at a high rate and are stored in so-called range bins [10, ch. 3.2.]. This is performed L times within a pulse repetition interval (PRI) (the inverse of the pulse repetition frequency, PRF). The number of samples collected, L, will depend on the sampling rate (PRF) within the PRI, and the samples are stored along the so-called fast-time axis.

Sampling is performed over multiple PRIs, which forms a two-dimensional array [10, ch. 3.3.]. The axes are generally referred to as the fast-time and slow-time axes because the sampling rate within a PRI is much higher compared to the PRF. Therefore, the frequency of samples taken along the fast-time axis (or range bin axis) is much higher than the frequency of transmitted pulses along the slow-time axis.

The resulting two-dimensional sample array is shown in Figure 2.2, where the dotted arrow line beneath the antenna sensor shows how a recently collected range bin sample is added to its corresponding PRI to form the two-dimensional matrix.

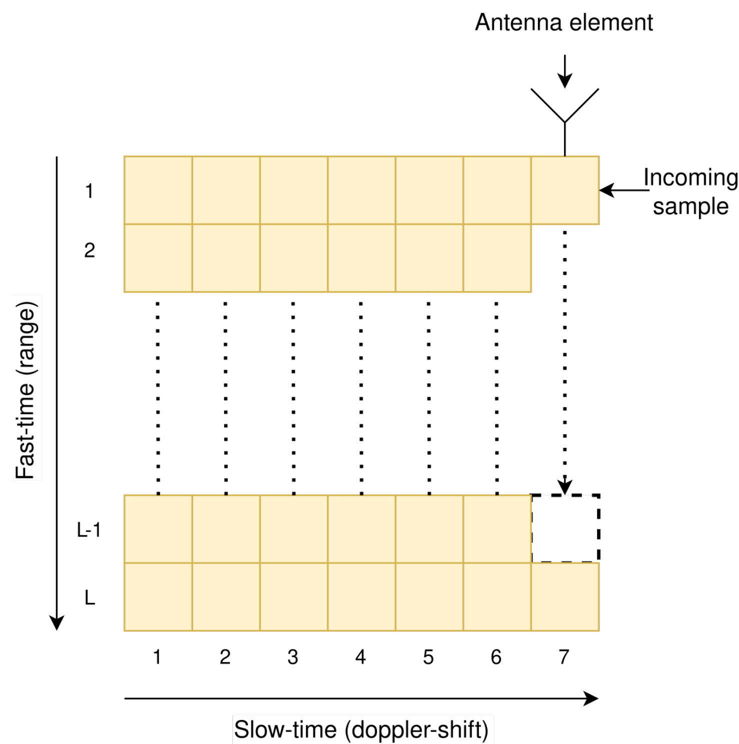


Figure 2.2: Fast-time and slow-time sampling storage from a single sensor element.

The two-dimensional array contains all range bin samples collected over several PRIs by a single sensor. This array can be extended into a three-dimensional array to cover all samples collected by several antenna sensor elements, as seen in Figure 2.3.

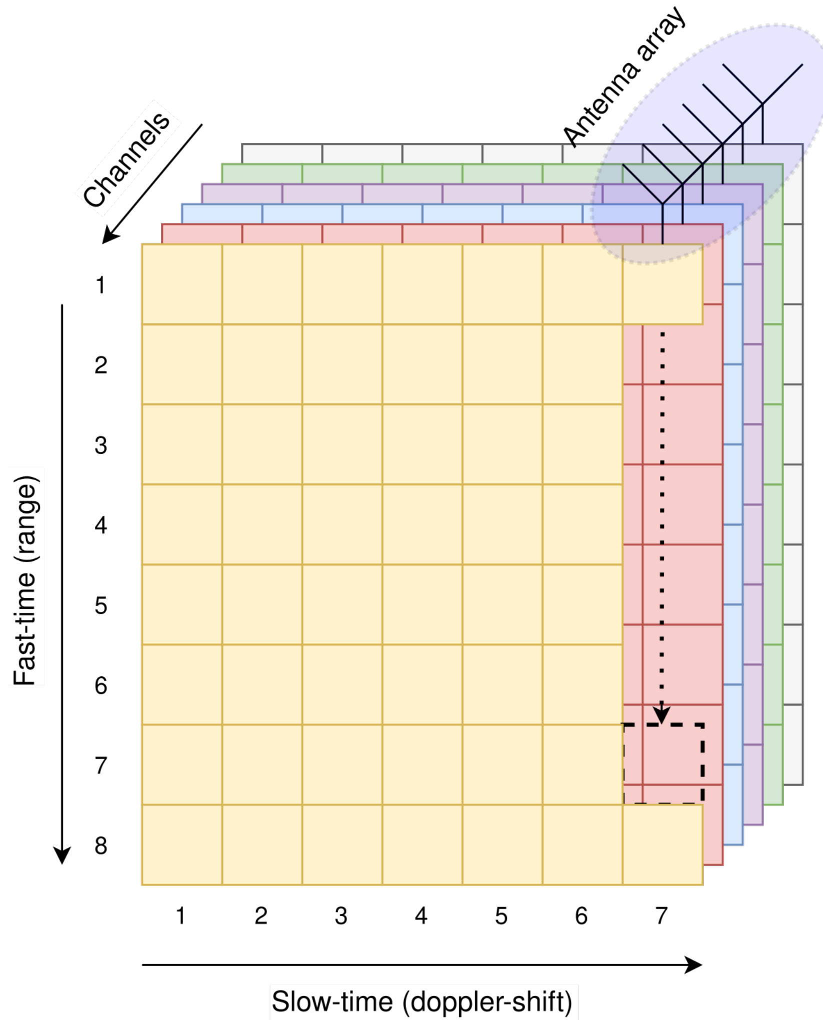


Figure 2.3: Fast-time and slow-time sampling storage from multiple sensor elements.

Here, the three-dimensional matrix is formed by six sensors, each collecting eight samples within each PRI, throughout seven consecutive PRIs.

2.3 ESPRIT

ESPRIT is an algorithm that utilizes an underlying rotational invariance of a signal space to estimate an incoming signal's parameters [7]. When used in a sensor array, relevant parameters to be estimated may be DOA, doppler shift of an arriving signal, and distance between the object and the array.

Consider a sensor array with a known displacement of the individual elements. The transmission and detection of an electromagnetic wave of the sensor array are done in the same physical location, i.e. a monostatic radar system with multiple sensor elements. This array transmits a pulse and then listens for an echo from a reflecting object. If the distance to the object is large, the reflected wave can be viewed as a planar wave that impacts the sensor array with a phase shift. This phase shift is the direction from where

the target was detected, the DOA. An illustration of the system described can be found in Figure 2.4.

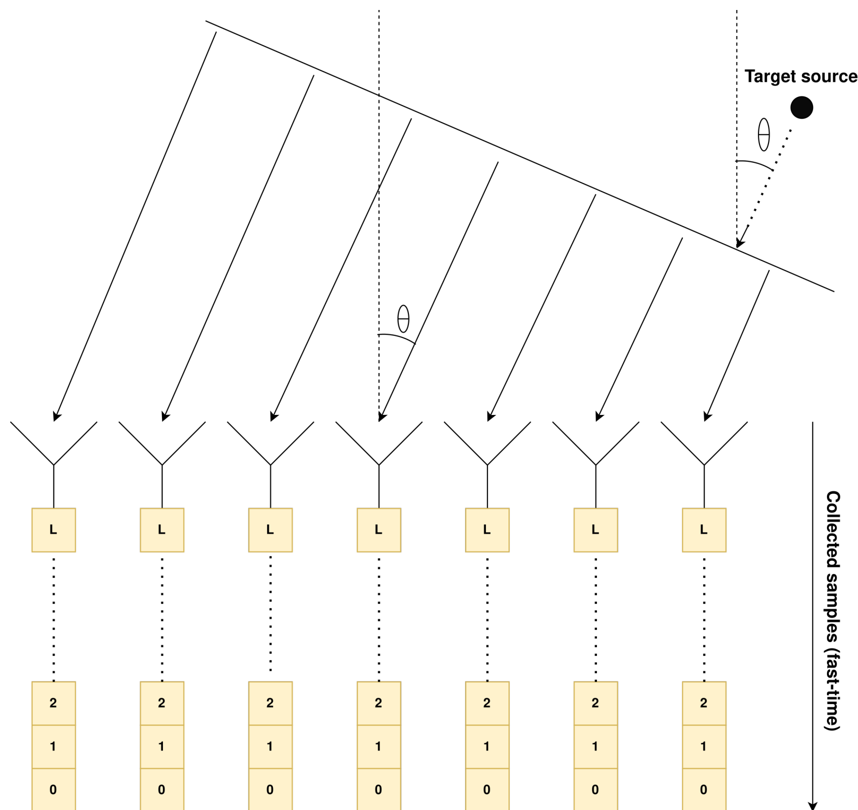


Figure 2.4: Sensor array sampling from a detected target.

Here, a planar wave falls into a one-dimensional sensor array where each antenna sensor continuously collects fast-time samples.

The reflected pulse can be generalized as

$$y[t] = \sum_{i=1}^{K} a_i(\theta) x_i[t] + n_i[t] \tag{2.3}$$

where K is the number of contributions of a signal at time t, $a(\theta)$ are weights of the incoming signal, $x(t)$ is the incoming signal and $n(t)$ is added Gaussian white noise. When estimating the DOA of a signal to a sensor array, θ can either be the azimuth or the elevation angle of an incoming signal, depending on the configuration of the sensor array.

The ESPRIT algorithm stores the samples collected from one antenna element in a vector. This vector is divided into smaller sub-vectors which are used to form matrix **A**. This is shown in Figure 2.5 and Figure 2.6.

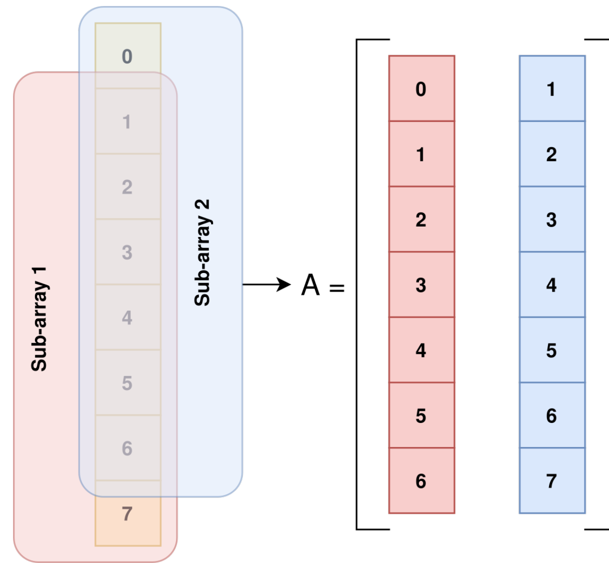


Figure 2.5: Dividing the samples into two sub-arrays to form matrix **A**.

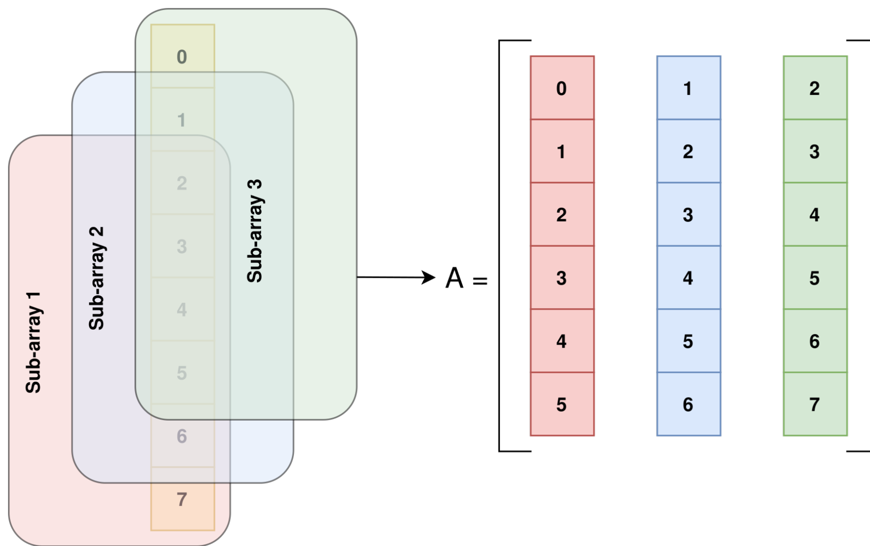


Figure 2.6: Dividing the samples into three sub-arrays to form matrix **A**.

Here, the total number of samples is eight which can form two or more sub-arrays. Forming more sub-arrays can provide more accurate results but increases the size of the matrix **A** and thus also the computational effort of calculating the auto-correlation matrix described below.

The next step is calculating the auto-correlation of the matrix **A**. This is to increase the magnitude of the detected target pulse and decrease the impact of noise in the signal. The auto-correlation is calculated by

$$\mathbf{R}_{yy} = \mathbf{A}\mathbf{A}^* \tag{2.4}$$

where $\mathbf{A}^*$ represents the complex conjugate transpose of matrix **A**.

The next step is to extract the signal subspace from the autocorrelation matrix $\mathbf{R}_{yy}$. This can be done by performing singular value decomposition (SVD) on the matrix. This decomposes the auto-correlation matrix of dimensions $m \times m$ into three matrices

$$\mathbf{R}_{yy} = \mathbf{U}\Sigma\mathbf{V}^* \tag{2.5}$$

where $\mathbf{U}$ is a $m \times m$ complex-valued matrix, Σ is a $m \times m$ matrix with real values on the diagonal and zeroes everywhere else and $\mathbf{V}^*$ is a $m \times m$ complex-valued matrix. Both $\mathbf{U}$ and $\mathbf{V}^*$ describe rotations while Σ is a scaling. From Equation (2.3), we can divide the SVD into two parts

$$\mathbf{R}_{yy} = \mathbf{U}\Sigma\mathbf{V}^* = \mathbf{U}_S\Sigma_S\mathbf{V}_S^* + \mathbf{U}_N\Sigma_N\mathbf{V}_N^* \tag{2.6}$$

where S and N refer to the signal and noise subspaces respectively. From Equation (2.3), we assume K contributions to the incoming signal. Thus, $\mathbf{U}_S$ comprises the first K columns of $\mathbf{U}$ that form the signal subspace.

Next, $\mathbf{U}_S$ is divided into two new matrices. This is done by forming the matrices

$$\mathbf{J}_1 = [\mathbf{I}_{M-1} \quad \mathbf{0}] \tag{2.7}$$

$$\mathbf{J}_2 = [\mathbf{0} \quad \mathbf{I}_{M-1}] \tag{2.8}$$

where $\mathbf{I}_{M-1}$ is an identity matrix of size $(M-1)$ and $\mathbf{0}$ is a column vector of zeroes. Through these, we form

$$\mathbf{S}_1 = \mathbf{J}_1\mathbf{U}_S \tag{2.9}$$

$$\mathbf{S}_2 = \mathbf{I}_2\mathbf{U}_S. \tag{2.10}$$

Now, these matrices form the linear equation system

$$\mathbf{S}_2 = \mathbf{S}_1\mathbf{P} \tag{2.11}$$

where the matrix $\mathbf{P}$'s eigenvalues are the complex-valued incoming signals. Here, it is possible to solve for $\mathbf{P}$ and thus extract the sought-after parameters of the signal. By performing, for instance, a least-squares solution on this equation system, we can find the matrix $\mathbf{P}$. Through eigen decomposition on this matrix $\mathbf{P}$,

$$\mathbf{P} = \mathbf{Q}\Lambda\mathbf{Q}^{-1} \tag{2.12}$$

we acquire the matrix Λ which contains the eigenvalues on its main diagonal. The eigenvalues have the form of

$$\lambda_k = \alpha_k e^{j\theta_k} \tag{2.13}$$

where θ_k is the sought-after angle of arrival.

ESPRIT can be extended into more dimensions to evaluate other parameters. This is done by switching the data set of the algorithm to instead take a 2D data set. From Figure 2.3, the DOA of a target at a certain range can be found by choosing the data set along the channel axis, and to find the velocity of a target the data set is chosen along the fast-time axis. Thus, we estimate a signal's parameters in two dimensions, called 2D-ESPRIT.

For 2D ESPRIT, the axes chosen are used to form the **A** matrix. Instead of forming **A** from sub-arrays, the 2D data set is split into sub-matrices to form matrix **A**, as shown in Figure 2.7.

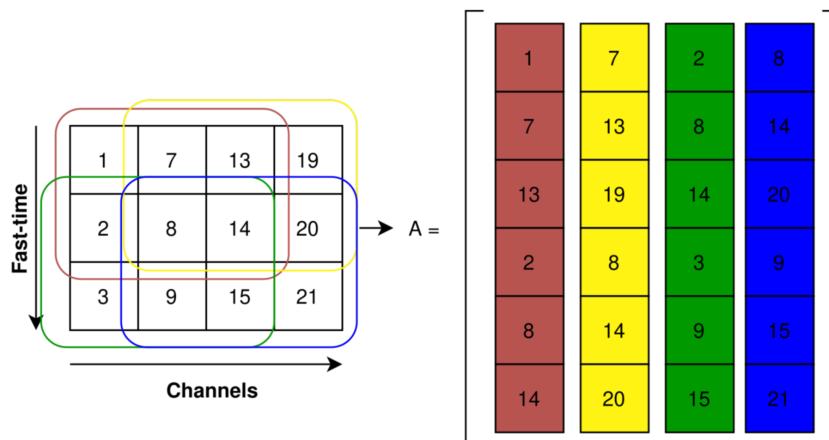


Figure 2.7: Forming matrix **A** in 2D over axes fast-time and channels.

Here, four submatrices are formed from the 3×4 data set. After this matrix has been formed, the following steps are the same as in 1D-ESPRIT. By forming the matrix **A** with different sized submatrices which results in **A** being a $M \times N$ matrix, the resulting covariance matrix $\mathbf{R}_{yy}$ will have dimensions $M \times M$ which will impact the computational effort required for all succeeding steps of the algorithm.

2.4 GPUs

A GPU is a processing unit that was originally designed to speed up the rendering of 3D graphics [12]. Its architecture has many cores, built for parallel applications with smaller concurrent computations. It differs from the CPU, where the CPU generally has fewer, but more powerful cores [1].

The Quadro RTX 6000 GPU builds on NVIDIA's Turing architecture [13]. It has 36 texture processing clusters (TPCs), each consisting of two streaming multiprocessors (SMs). Within one SM are 64 integer (INT) and 64 floating point (FP) CUDA cores, both types of cores designed for handling integer or floating point 32-bit operations respectively. The SM also contains two FP64 units for handling floating point 64-bit operations and eight tensor cores that can handle either INT4, INT8, or FP16 bit precision operations.

The peak FP32 floating point operations (FLOPS) specified for the GPU is 16.3 TFLOPS.

The GPU also has several cache memories. The cache is a smaller memory that stores frequently accessed data to decrease the data fetching time from the main memory. The different levels of the cache indicate the proximity of the processing unit, where a level 1 (L1) cache is closer than a level 2 (L2) cache. The higher-level cache is bigger but farther away. Therefore, it has a higher data fetching time than the lower-level cache.

Each SM has 96 kilobytes (kB) in a L1 cache, which can be allocated as a portioned cache and shared memory. The shared memory can be accessed through CUDA and the cache is fully handled by the hardware. Thus, the shared memory can be faster than the cache if managed properly.

The entire chip has a shared L2 cache of 6144 kB memory space. The system clock is 1455 MHz and the transistor count, which is built on 12 nm fin field-effect transistor (FinFET) technology, is 18.6 billion. The total board power consumption is 295 Watts. A summary of the features is found in Table 2.1.

Table 2.1: Quadro RTX 6000 board features.

Feature	**Number**
TPC	36
SM	72
FP32 cores / SM	64
INT32 cores / SM	64
Tensor cores / SM	8
FP32 cores total	4608
INT32 cores total	4608
Tensor cores total	576
Peak FP32 FLOPS	16.3 TFLOPS
L1 cache / SM	96 kB
L1 cache total	6912 kB
L2 cache total	6144 kB
Clock base frequency	1455 MHz
Transistor count	18.6 billion
Total board power	295 Watt

2.5 CUDA C/C++

In this section, a short background about CUDA C/C++ is given. Thereafter, relevant concepts regarding CUDA C/C++ used in the project are explained. Finally, the libraries used in the project to build several functions are introduced.

CUDA C/C++ is a programming platform for enabling programming on GPUs provided by NVIDIA with its initial release in 2006 [14]. It was initially developed for C but has since extended to support other languages, such as C++. The interface allows programmers to directly access a GPU for general-purpose GPU computing to speed up the processing of heavy workloads by utilizing the many cores of a GPU.

2.5.1 Programming in CUDA C/C++

The CUDA C/C++ language syntax is much the same as that of C and C++, where one can write a basic program in C++ and it compiles with CUDA. The main difference is that the programmer decides whether a function is executed on the host (CPU) or the device (GPU). If a function is to be executed on the device, a so-called kernel function is created with the specifier __GLOBAL__ which can be called from the host. To make a function only callable by the device, the __DEVICE__ specifier may be used. These two specifiers are shown in Listing 2.1.

```
__global__ void my_func();
__device__ void my_GPU_func();
```

Listing 2.1: Kernel function declaration.

The call to a kernel function from the host is shown in Listing 2.2.

```
my_func<<<x, y>>>();
```

Listing 2.2: Kernel function call from the host.

The X and Y arguments are the block size and thread size of the function call, and are used to specify how many threads should be within each block and how many blocks that should be called to execute the function. A block is therefore an abstraction used to group threads. A streaming multiprocessor within the GPU executes threads in warps which consists of 32 threads per warp. Therefore, to maximize the performance of the GPU computation and utilization of the multiprocessors, the number of threads per block should align with the warp size, i.e. divisible by 32. Further, the ratio of number of blocks per function and threads per block can impact the computation efficiency and therefore the utilization of the GPU.

The kernel call is called from the CPU asynchronously, meaning that the CPU will not wait for the device to finish its function before stepping further in the code unless explicitly told to wait. Therefore, proper synchronization between the host and device is important at vital parts of the program to ensure correct functionality. To synchronize the CPU and GPU, the `cudaDeviceSynchronize()` function can be called which will tell

the host to wait for the device to finish preceding instructions before stepping further in the code. However, too much synchronization will lower a program's efficiency and increase processor idle time which is to be avoided.

Indexing is conducted somewhat differently on the GPU compared to as if written in C++ with the use of blocks and threads. Within the kernel function, the desired index is calculated through the **threadIdx.x**, **blockIdx.x** and **blockDim.x** syntaxes. The index of the currently executing thread is found through **threadIdx.x**, the current block through **blockIdx.x**, and the number of threads within a block through **blockDim.x**. An example of finding an index is shown in Figure 2.8.

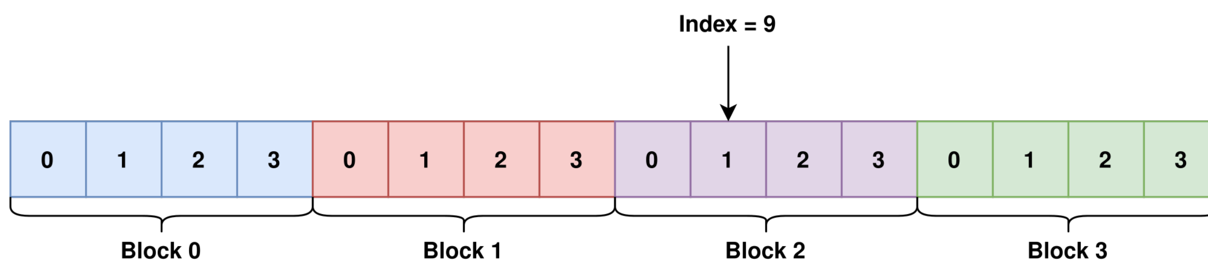


Figure 2.8: Indexing in CUDA.

Here, 4 blocks consist of 4 threads each. To access index 9, one would use block index 2 with thread index 1. The index is calculated with **index = threadIdx.x + blockIdx.x × blockDim.x**, where the block dimension would be 4 in Figure 2.8. The term **blockIdx.x × blockDim.x** essentially calculates the offset from the large data set to the start of a given block. The appended **.x** specifies that the sought values are given along the x-axis, implying that indexing can be done in more dimensions.

2.5.2 CUDA streams

One important aspect of achieving high utilization of the device is using CUDA streams. Streams can be compared to threads, where the streams are pieces of code that can be concurrently executed on the GPU to increase the use of parallelism.

By default, kernel calls are called into the default CUDA stream, meaning that CUDA calls are executed sequentially. However, to increase utilization of the GPU, it is desirable to increase concurrent execution of simultaneous streams. Observe the following example in Listing 2.3.

```
1 #include <stdio.h>
2
3 __global__ void my_func() {
4     printf("Hello from GPU");
5 }
6
7 int main() {
8     int num_streams = 2;
9     cudaStream_t streams[num_streams];
```

```
10     for (int i = 0; i < num_streams; i++) {
11         cudaStreamCreateWithFlags(&streams[i], cudaStreamDefault);
12         my_func<<<1,1,0,streams[i]>>>();
13     }
14     cudaDeviceSynchronize();
15     return 0;
16 }
```

Listing 2.3: Two concurrent CUDA streams.

In this example, two streams are initialized and dispatched to execute the function `my_func()` concurrently in two CUDA streams. The `cudaDeviceSynchronize()` call synchronizes the CPU and GPU, so the CPU waits for both stream's execution before stepping further in the code. To synchronize a single stream with the CPU, the call to `cudaStreamSynchronize()` can be invoked.

In this project's implementation of 2D-ESPRIT, multithreading is used beside CUDA streams to further increase the parallelism of the algorithm. This is done by multiple threads launching multiple instances of CUDA streams concurrently to further increase the utilization of the GPU. Thus, the data set to process must be correctly divided and assigned. This is shown in Figure 2.9.

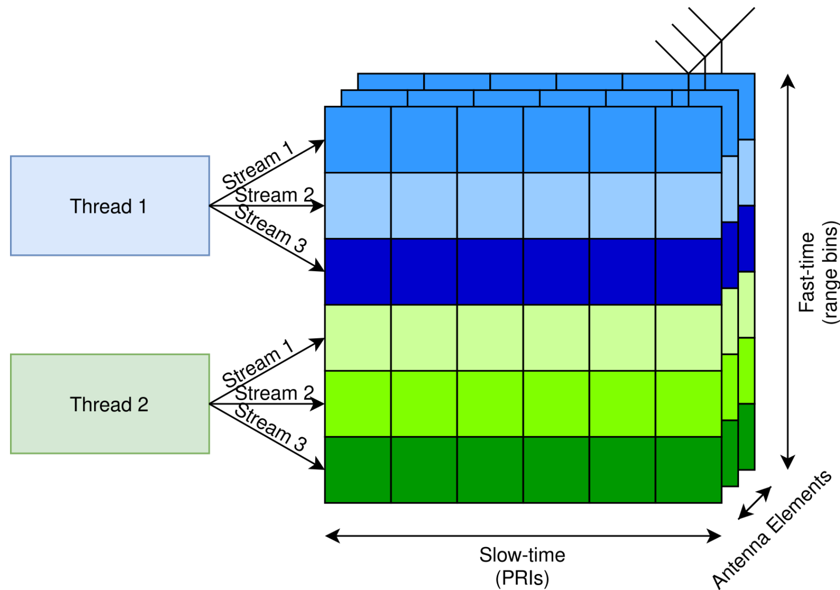


Figure 2.9: Two threads dispatching three streams each to execute one run of the 2D-ESPRIT algorithm.

In this figure, we see two threads each launching three CUDA streams. Each CUDA stream is assigned a 2D slice of the 3D data set. Each CUDA stream is then used to dispatch functions on the device and transfer memory between the host and the device.

2.5.3 Supported libraries

NVIDIA has support for algebraic functions and solvers through their GPU-accelerated libraries. These include cuBLAS and cuSOLVER, which are implementations of basic linear algebraic subprograms (BLAS) and linear system and eigen solvers respectively. cuBLAS chiefly consists of more fundamental arithmetic computations, such as matrix multiplication, whereas the cuSOLVER library implements LAPACK-like features on the GPU, such as SVD and least-squares solvers.

The implementation of ESPRIT is done by using several functions from the cuBLAS and cuSOLVER libraries. The functions mainly used in this thesis work are matrix multiplication and solvers for SVD and a least-square solution.

2.6 Utilization

To estimate the GPU utilization, the total floating point operations (FLOPS) of the entire algorithm should be estimated. The steps of the algorithm, as described in Section 2.3, can be summarized as forming matrix $\mathbf{A}$, calculating the covariance matrix $\mathbf{R}_{yy}$, SVD, two instances of extracting subspaces to form $\mathbf{S}_1$ and $\mathbf{S}_2$, a least-square solver and the solving for the eigenvalues. The time complexity of each function is summarized in Table 2.2, where m and n are the rows and columns of matrix $\mathbf{A}$ respectively and p is the number of extracted columns of the signal subspace, i.e. the number of estimated pulses.

Table 2.2: Functions required for ESPRIT with their corresponding estimated time complexity.

Function	**Required operations**
Form-A	mn
Calculate covariance matrix	$2mn^2$ [15, ch. 1.2.4.]
SVD	$21m^3$ [15, ch. 5.4.5.]
Form $\mathbf{S}_1$ and $\mathbf{S}_2$	$2mp$
Least-square solver	$2p^2(m - \frac{p}{3})$ [15, ch. 5.3.3.]
Eigensolver	$10p^3$ [15, ch. 7.5.6.]

Since we are dealing with complex numbers, a single multiplication of two complex numbers requires four floating-point multiplications and two additions. Thus, we multiply the time complexity with the factor 6 to gain an estimate for how many total FLOPS are required for computing the data set using complex numbers.

The formula for calculating the total number of required FLOPS in a single range bin is thus, using the values from Table 2.2 and using complex numbers,

$$
\begin{aligned}
FLOPS_BIN &= \sum Functions \\
&= 6(mn + 2mn^2 + 21m^3 + 2mp + 2p^2(m - \frac{p}{3}) + 10p^3) \\
&= nm(1 + 2n) + m(21m^2 + 2p(1 + p)) + 10p^3 - \frac{2p^3}{3} \\
&= nm(1 + 2n) + m(21m^2 + 2p(1 + p)) + \frac{28p^3}{3}.
\end{aligned}
\tag{2.14}
$$

Furthermore, the total number of FLOPS required for calculating the entire data set is $500 \times FLOPS_BIN$, since the entire data set used in the algorithm handles 500 range bins. The total number of FLOPS required for each covariance matrix data size is summarized in Table 2.3.

Table 2.3: Required FLOPS for 2D range bin slice and all range bins for each data set size parameters.

Covariance matrix size	**m**	**n**	**p**	**Range bin FLOPS**	**Total FLOPS**
64×64	64	667	15	3.75×10^8	1.88×10^{11}
180×180	180	598	15	1.51×10^9	7.54×10^{11}
360×360	360	403	15	6.58×10^9	3.29×10^{12}

When estimating the utilization of the GPU, the FLOPS/s metric is used. By finding the FLOPS/s of the program and dividing it by the peak FLOPS/s of the GPU, we gain an estimate of the GPU utilization. The formula is

$$
U = \frac{\frac{Total_FLOPS}{Execution_time}}{Peak_FLOPS} \tag{2.15}
$$

3

Workflow

In this chapter, the workflow of the project is explained. It begins with describing how the initial version of the ESPRIT algorithm was implemented and then how the work was conducted to improve upon the initial version to increase the utilization of the GPU.

3.1 Initial ESPRIT version

The project began by researching the ESPRIT algorithm to identify the relevant steps. When the steps had been identified, we started implementing an initial version of the algorithm using a data set in one dimension. This was done by investigating built-in functions in the cuBLAS and cuSOLVER libraries, such as herk (which performs $\mathbf{AA}^H$ and stores it in upper triangular form) and gesvd (which performs SVD on a matrix), and using them in the algorithm. When a step of the algorithm had been implemented in CUDA with a built-in function, such as the auto-correlation step, it was compared to the same step in MATLAB with the same data set to verify that it worked as intended. This was done for every step in the algorithm, which yielded similar results for CUDA and MATLAB.

When comparing the results of CUDA and MATLAB, we realized that some functions, such as SVD, yielded somewhat different results when implemented in CUDA and MATLAB. This is due to the functions decomposing the matrices slightly differently. However, we concluded it was correctly implemented despite yielding somewhat different results.

When every step of the algorithm had been implemented, a custom data set was run through every consecutive step and compared to the MATLAB implementation. When the algorithm yielded practically the same results, it was deemed correctly implemented in CUDA.

The next step of this initial version was to expand it into two dimensions. This was done by extracting a 24×60 slice from a $24 \times 60 \times 500$ data set and tweaking the steps of the algorithm to handle this data. The forming of matrix $\mathbf{A}$ and the auto-correlation step were the largest differences since this extracted sub-matrices of the extracted slice and converted them into columns of the $\mathbf{A}$-matrix. This resulted in an $\mathbf{A}$-matrix similar to the one implemented for one dimension. The main difference was that the matrix was much

larger which resulted in a much larger auto-correlation matrix $\mathbf{R}_{yy}$. It was also noted that by changing the sizes of the sub-matrices in forming $\mathbf{A}$, we could effectively change the computation effort of all succeeding steps to investigate different computational burdens. This is because, if $\mathbf{A}$ is $M \times N$, the resulting covariance matrix size becomes $M \times M$ which is shared by the succeeding computation steps of the algorithm. Thus, all succeeding steps are dictated by the size of M.

The consecutive steps of 2D-ESPRIT after forming $\mathbf{A}$ were practically the same. The entire data set of $24 \times 60 \times 500$ was handled by running the algorithm 500 times sequentially, iterating 2D-ESPRIT on each extracted 24×60 data slice.

The Quadro RTX 6000 supports computing for various bit sizes. As described in Section 2.4, the GPU has 32 FP32 for each FP64 computing unit. Thus, the algorithm was run with FP32 and FP64 bit sizes to observe the difference in how the GPU handled the two number representations.

3.2 Second ESPRIT version

For the improved version, it was observed that the implemented function for the auto-correlation step was very time-consuming. For the first run of the function, the herk function was quite fast, but for consecutive runs, it was very slow. The speed in consecutive runs was almost a hundred times longer, which led us to believe there was some bug with the function. In the second and improved version, the time spent for herk was reduced and had consistently the same value for all runs of the function.

Another improvement from the initial version was to reduce the data transfers from the host to the device and vice versa. There were many data transfers in the initial version to be able to debug the output of the compute steps of the algorithm. In the improved version, the data was not transferred between the consecutive steps of the function, thus reducing excessive data transfer times of the algorithm by having the data ready for calculation on the device.

Another difference was a different SVD implementation supplied by the cuSOLVER library. This did not, however, affect the time required for calculating the SVD.

The largest difference between the first and second versions was the implementation of multiple threads running multiple CUDA streams to further try to exploit the parallelism of the algorithm. Here, the entire data set was divided into 500 separate 2D data slices. Then, 500 CUDA streams were created on a single thread, so that one CUDA stream handled one instance of the algorithm. Then, this was further divided so that two threads created 250 CUDA streams each to further parallelize the algorithm. The first thread handled the first 250 range bin slices and the second the remaining data slices. Each configuration used the same principle, where each thread would be assigned $\frac{500}{\#threads}$ data

slices to handle, where each thread created one CUDA stream per data slice, as depicted in Figure 2.9. The configurations tested are listed in Table 3.1.

Table 3.1: Configurations of threads and CUDA streams tested.

Configurations
One thread, 500 CUDA streams
Two threads, 250 CUDA streams per thread
5 threads, 100 CUDA streams per thread
10 threads, 50 CUDA streams per thread
20 threads, 25 CUDA streams per thread
25 threads, 20 CUDA streams per thread

These configurations were tested to try to identify an optimal ratio where the total runtime required to handle the entire data set was minimized while still not overpushing the GPU. For some configurations, the least-square solver produced NaN values which told us that the program was unable to handle the computational burden of configuration. In the following version, changes were made to make the GPU pass more configuration tests with the same data set.

3.3 Third ESPRIT version

The third version had some changes compared to the second version. The changes were mostly trying to minimize excessive synchronization to further utilize the GPU by revising and removing excess calls to the synchronizing function `cudaStreamSynchronize()`. It was also was identified that the least-square solver was a major bottleneck of the algorithm due to it always using 50 iterative refinements, which occupied massive resources on the GPU. For the third version, the maximum amount of iterative refinements was set to five instead, which contributed to many streams per thread configurations being able to process the entire data set without failing as they had done in the second version.

4

Results

This chapter contains the results of the implemented algorithm. The algorithm was implemented in three different versions, with each version being improved upon from the previous version. All three versions handle the same 3D data set containing 24 antenna elements, 60 PRIs and 500 range bins. The 2D-ESPRIT algorithm is executed once on each range bin, meaning the algorithm is executed 500 times. The signal processing is thus performed on each range bin's corresponding 2D matrix. Moreover, the size of the matrix $\mathbf{A}$ in 2D-ESPRIT can be altered, as explained in Section 2.3. Thus, every configuration is run with three resulting covariance matrix sizes to evaluate performance with a larger data set per range bin. The three different computational sizes used were: 360x360, 180x380 and 64x64. For simplicity, these sizes will throughout this thesis be referred to as having a covariance matrix size of 360, 180, or 64 elements, respectively.

For each run, four different plots were generated. The first plot contains the mean time for each function used in the algorithm, including computation and various setup and memory times. The second contains only the computation time and the third only the memory time. The fourth plot contains the time between each function used in each run of the algorithm. For each plot, a corresponding scatter plot was generated with each range bin time to see the variation between consecutive runs. The total time in each plot refers to the wall clock time, meaning the total execution time of the program.

The functions used in the algorithm are summarized in Table 4.1.

Table 4.1: Function names and explanation.

Function name	**Explanation**
Form-A	The creation of matrix $\mathbf{A}$ used to calculate the covariance.
AUTOCORR-2D	Setup and calculation of $\mathbf{A}\mathbf{A}^T$.
SVD	The singular value decomposition of the covariance matrix $\mathbf{R}_{yy}$.
Extract-subarrays	Extract signal subspace from matrix $\mathbf{U}$ and setup equation system for least-squares solver.
LEAST-SQUARES	Least-square solver for finding matrix $\mathbf{P}$.
EIGENSOLVER	Solver for finding the eigenvalues of matrix $\mathbf{P}$.
RUN / STREAM	Entire range bin execution.

Every configuration is also run with complex FP32 and FP64 to see the execution time

differences.

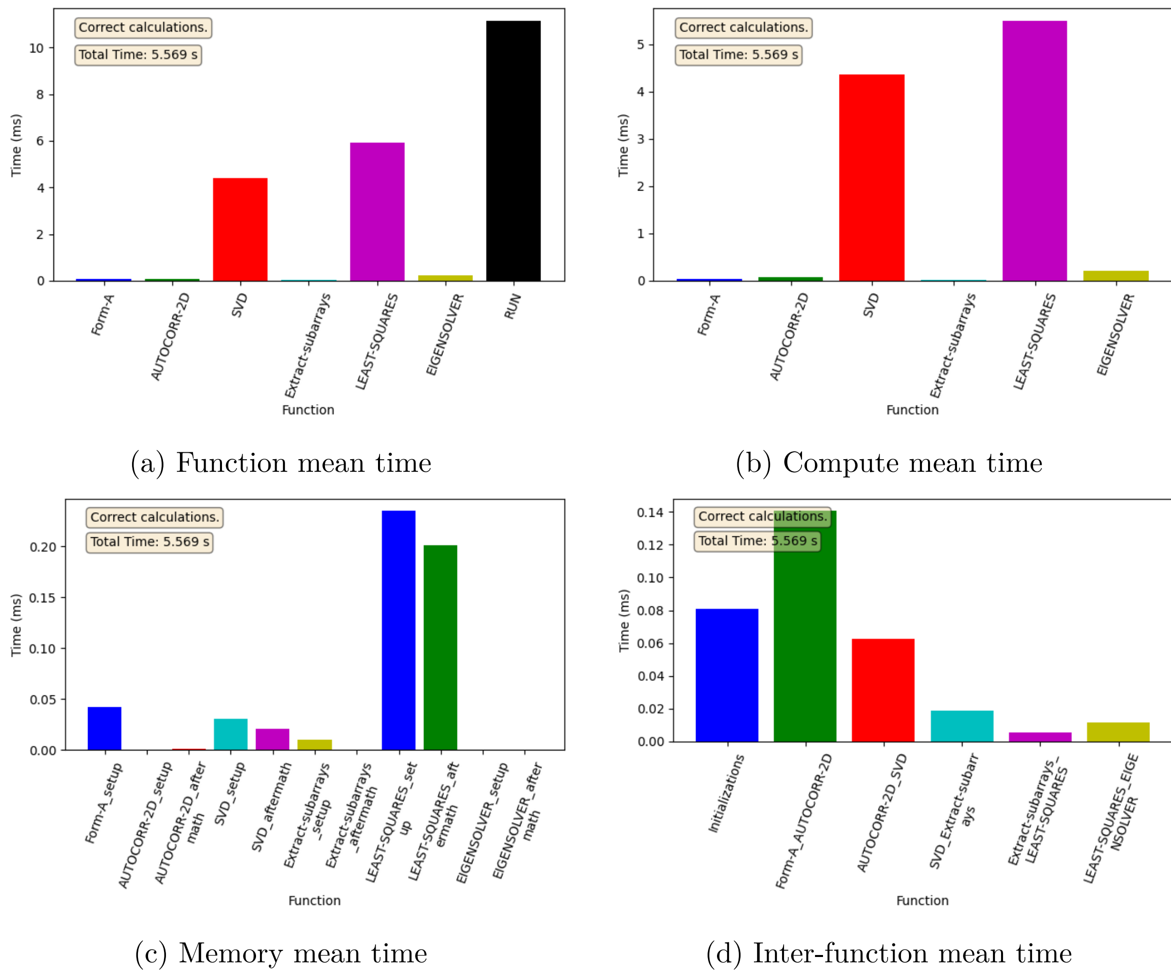
Below are the selected results from each version. The remaining plots generated for each configuration are found in Appendix A.

4.1 First version

The first version executes 2D-ESPRIT on each range bin sequentially. The results below are the results from the first version, run with different covariance matrix sizes and with FP32 and FP64. The different number representations were run to see the computing differences of having smaller or larger number representations.

4.1.1 FP32

The results for FP32 mean times and corresponding scatter plots for a covariance matrix size of 64 elements are shown in Figure 4.1 and Figure 4.2.



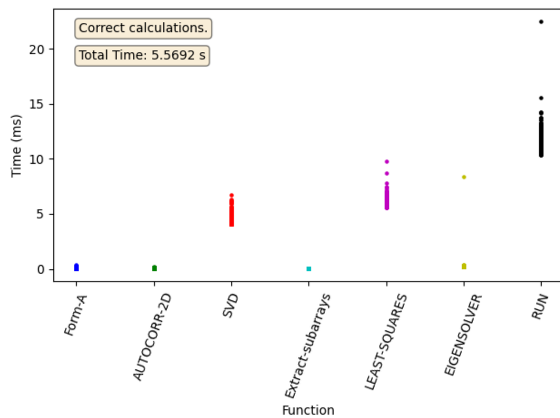
(a) Function mean time

(b) Compute mean time

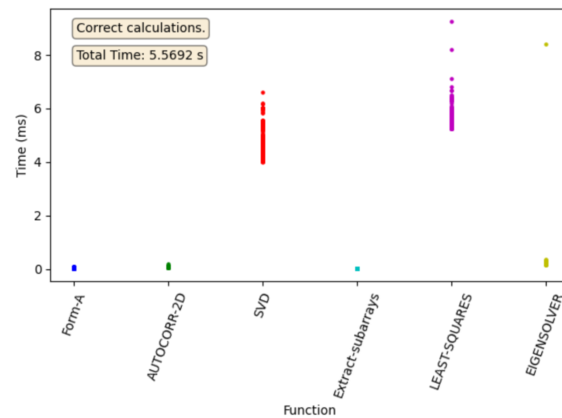
(c) Memory mean time

(d) Inter-function mean time

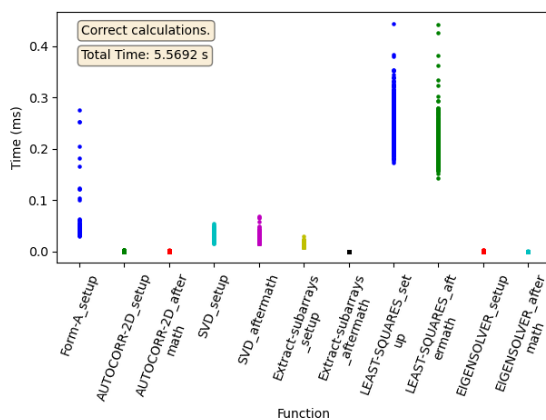
Figure 4.1: Mean times for the first version using FP32 with a covariance matrix size of 64 elements.



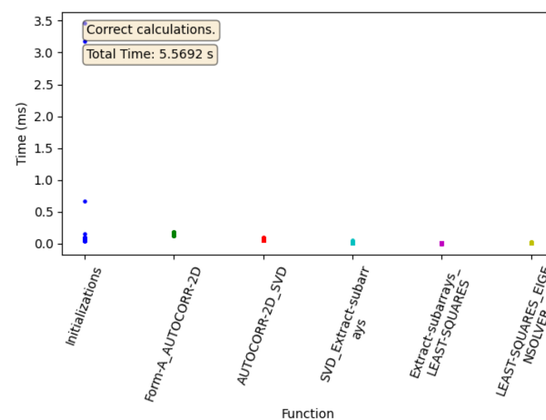
(a) Function mean time



(b) Compute mean time



(c) Memory mean time



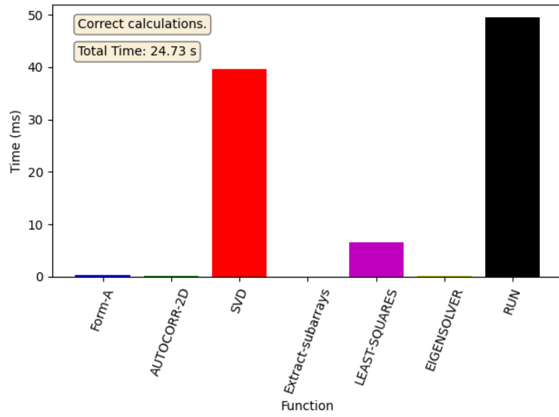
(d) Inter-function mean time

Figure 4.2: Scatter plots for the first version using FP32 with a covariance matrix size of 64 elements.

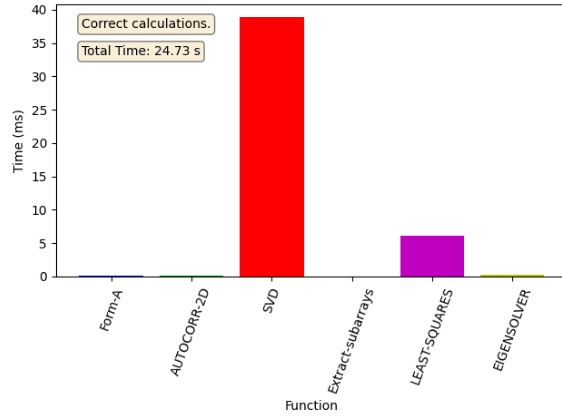
For the mean times, the computation time dominates the execution time of the algorithm. The two most heavy algorithms are the SVD and the least-squares solver. The other functions require very little time in comparison. Moreover, the inter-function times and times required for setting up function calls and freeing resources are comparatively small.

Considering the scatter plots, the Form-A, AUTOCORR-2D and Extract-subarrays functions require very little time to finish and have low spread in execution times. The EIGENSOLVER function is similar, however, one outlier can be observed at around 10 milliseconds (ms). The SVD and least-square solver lie around 4 to 8 ms and 5 to 10 ms respectively with quite a bit of spread in execution times. The memory times have a low spread and low overall times. The initialization stage for the inter-function times has some outliers at around 3-3.5 ms, otherwise, the times are low.

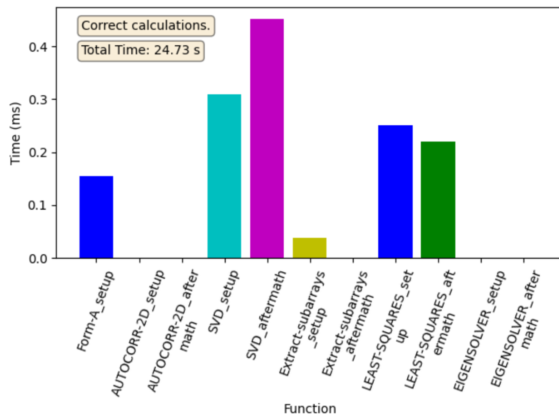
The results for a resulting covariance matrix size of 360x360 are shown in Figure 4.3 and Figure 4.4.



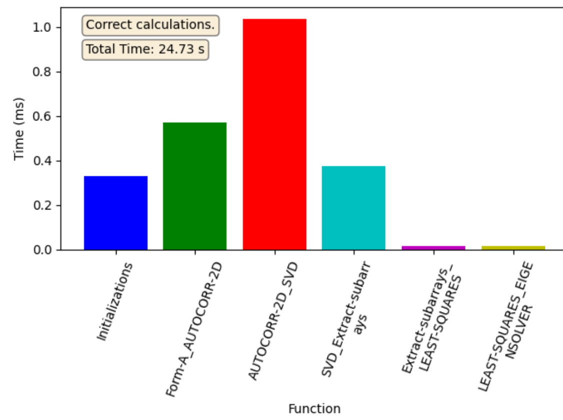
(a) Function mean time



(b) Compute mean time

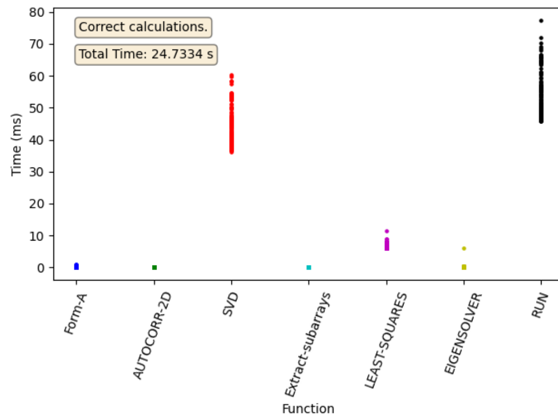


(c) Memory mean time

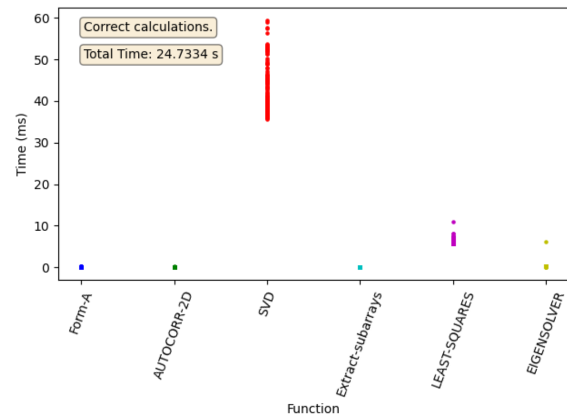


(d) Inter-function mean time

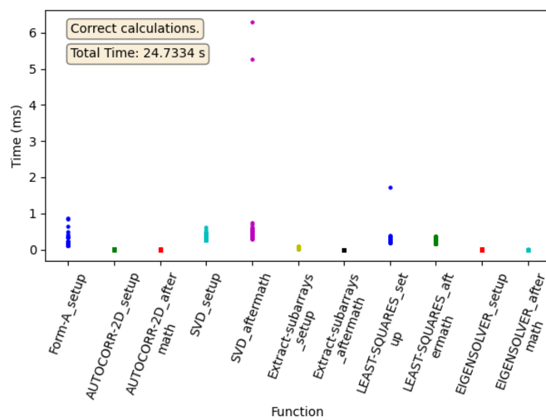
Figure 4.3: Mean times for the first version using FP32 with a covariance matrix size of 360 elements.



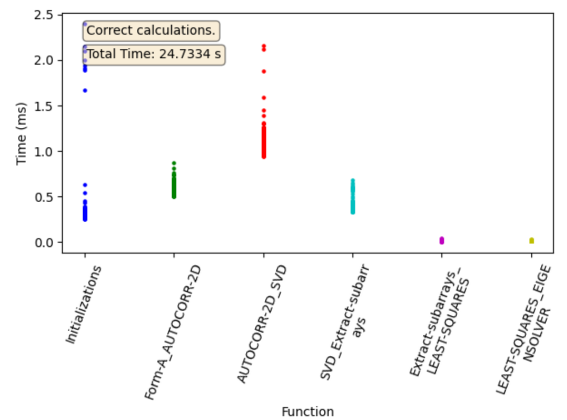
(a) Function mean time



(b) Compute mean time



(c) Memory mean time



(d) Inter-function mean time

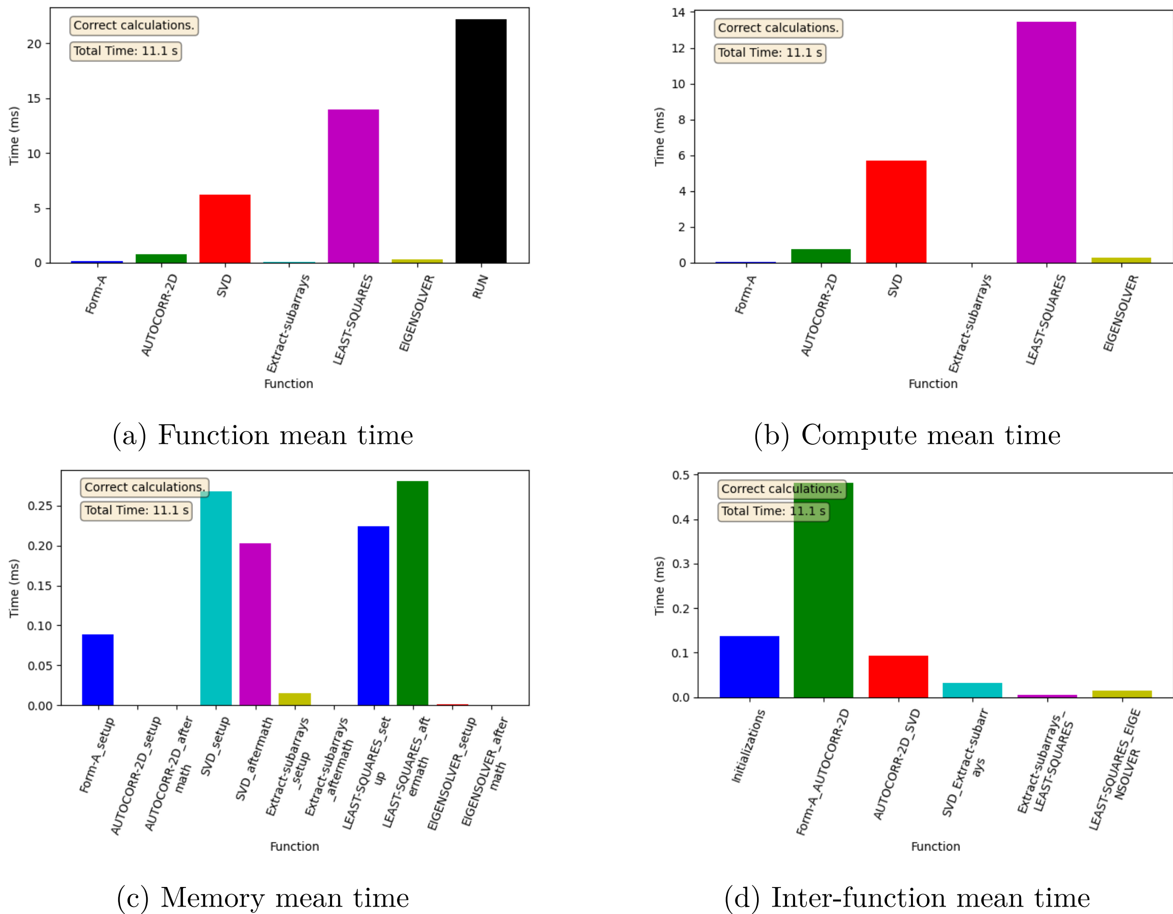
Figure 4.4: Scatter plots for the first version using FP32 with a covariance matrix size of 360 elements.

As can be seen in Figure 4.3b, we see a steep increase in time required for the SVD function because of the increased covariance matrix size. The least-square solver takes approximately the same time as before and the other functions are very small compared to the SVD and least-square solver. The memory times and inter-function times seen in Figure 4.3c and Figure 4.3d, are very small in comparison to the calculation times.

We observe a large spread for the SVD function ranging from 35 to 60 ms each. The other functions have a low spread, with the least-square solver lying around 8-10 ms. We can observe one outlier in the eigensolver function at around 10 ms, but has otherwise a low execution time.

4.1.2 FP64

Due to the increased size of the FP64 number representation and the capabilities of the Quadro RTX 6000 as described in Section 2.4, we should expect an increase in wall-clock times and calculation times compared to FP32. The mean times and scatter results for the ESPRIT-2D algorithm using FP64 with a covariance matrix size of 64 elements are shown in Figure 4.5 and Figure 4.6.



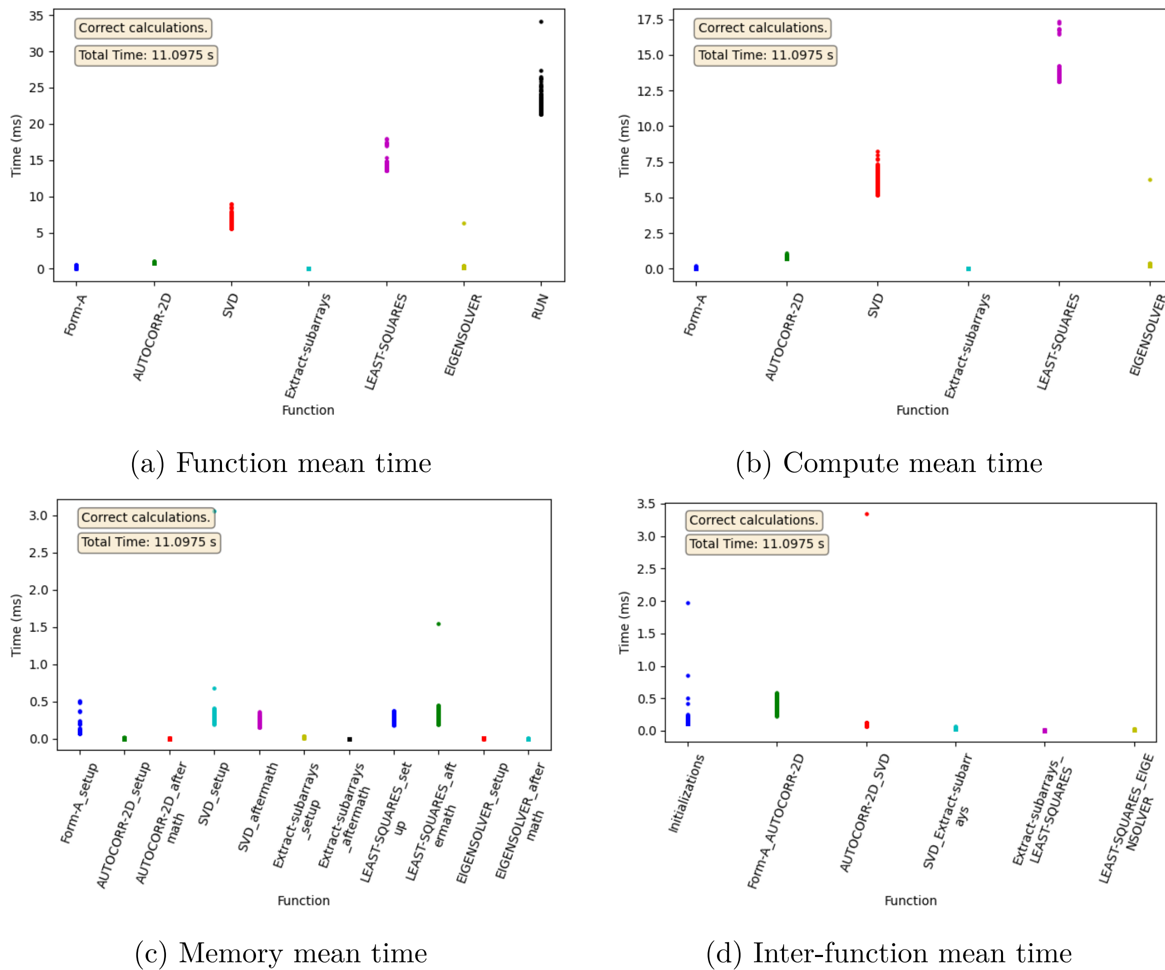
(a) Function mean time

(b) Compute mean time

(c) Memory mean time

(d) Inter-function mean time

Figure 4.5: Mean times for the first version using FP64 with a covariance matrix size of 64 elements.



(a) Function mean time

(b) Compute mean time

(c) Memory mean time

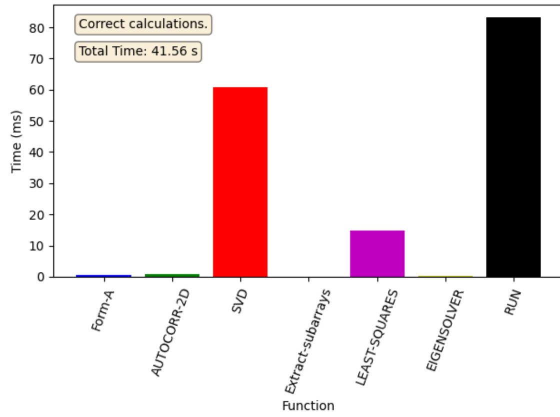
(d) Inter-function mean time

Figure 4.6: Scatter plots for the first version using FP64 with a covariance matrix size of 64 elements.

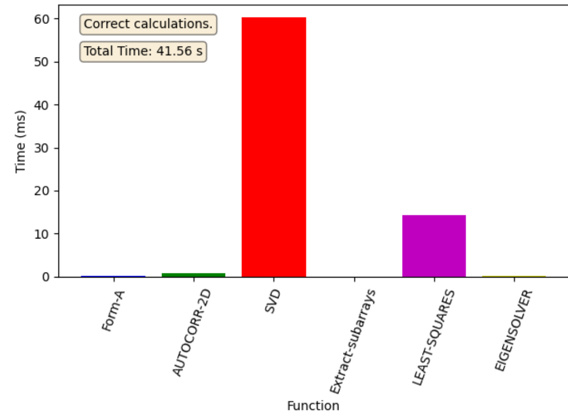
Here, the results are similar to that of the FP32 times only that the time required for each function is increased. The ratio of the SVD and least-square functions look very similar and the other functions' execution times are comparatively low. The memory and inter-function times are very low in comparison to the computation times.

For the scatter plots, the SVD and least-square functions require 5-10 and 15-20 ms respectively. Beside one outlier for the eigensolver function at around 8 ms, the execution times for the other functions are low. For the memory times, there is one outlier at 3 ms for the SVD setup time and one at 1.5 ms for the least-square solvers freeing of resources. For the inter-function times, one outlier around 3.4 ms can be observed between the auto-correlation function and the SVD. The initialization stage has some spread, but the other times are very similar and low.

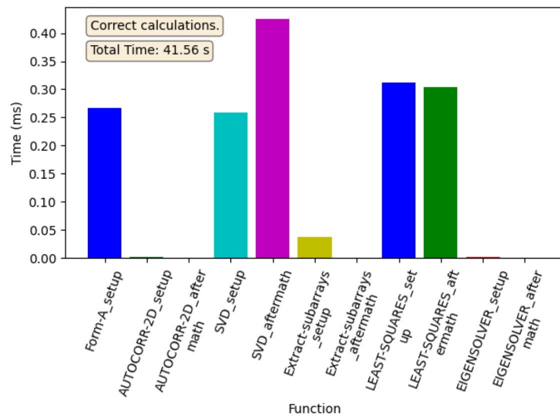
The corresponding plots for an increased covariance matrix size of 360 elements are shown in Figure 4.7 and Figure 4.8.



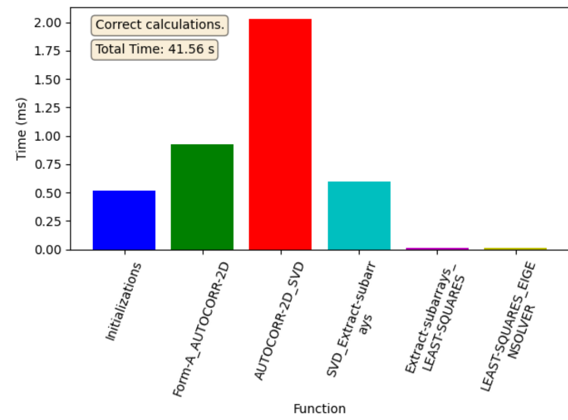
(a) Function mean time



(b) Compute mean time

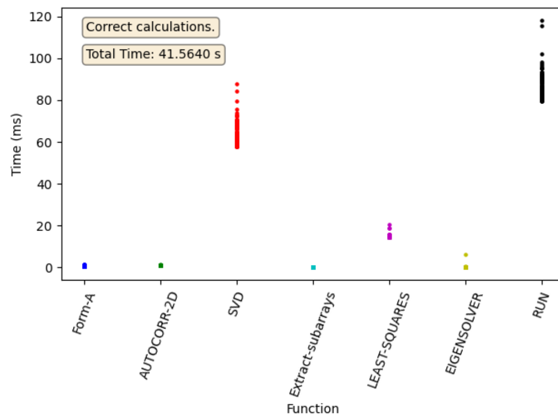


(c) Memory mean time

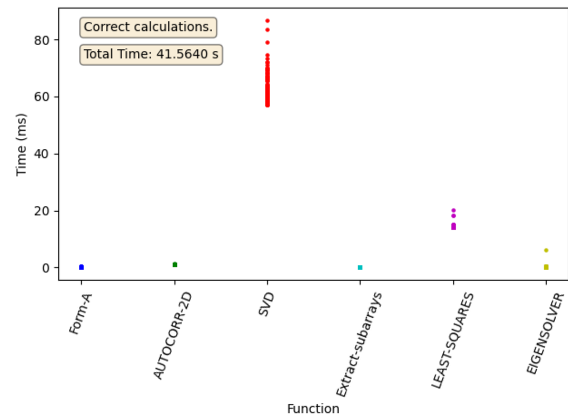


(d) Inter-function mean time

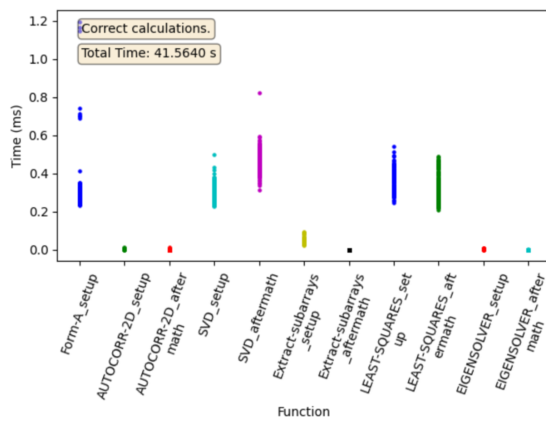
Figure 4.7: Mean times for the first version using FP64 with a covariance matrix size of 360 elements.



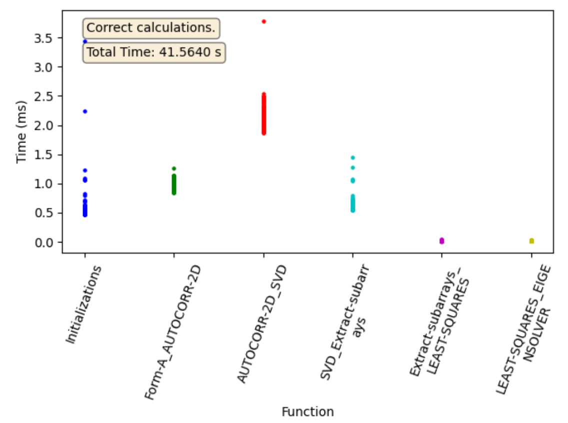
(a) Function mean time



(b) Compute mean time



(c) Memory mean time



(d) Inter-function mean time

Figure 4.8: Scatter plots for the first version using FP64 with a covariance matrix size of 360 elements.

As with the previous results, the FP64 plots are very similar to the FP32 plots only that the execution times are higher. The SVD requires around 60 ms and the least-square solver 15 ms. A single range bin run takes approximately 80 ms. Furthermore, the memory and inter-function times are low in comparison.

For the scatter plots, the SVD takes around 60-85 ms. The least-square solver takes around 20 ms and the other functions require very little time in comparison. The memory and inter-function times require very little time compared to the computation times.

4.1.3 Wall-clock times for the initial ESPRIT version

Figure 4.9 summarizes the total execution times of the entire data set for both FP32 and FP64 with covariance matrix sizes of 360, 180 and 64 elements.

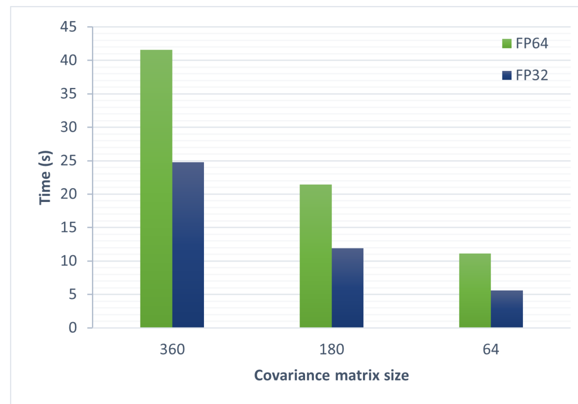


Figure 4.9: Initial ESPRIT version's wall-clock times for FP32 and FP64 using different data sizes.

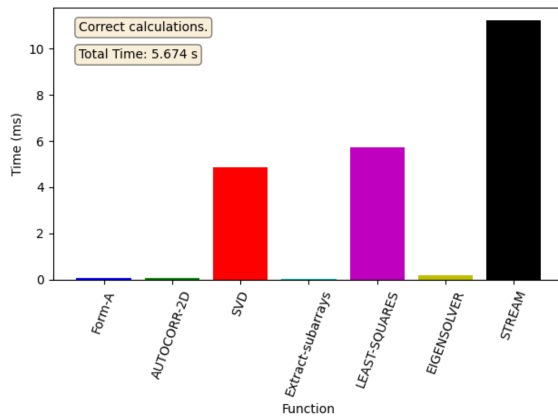
Here we see that using FP32 requires less time compared to FP64. Also, the time increases with an increased data size. The total execution time looks to have a linear correlation to the data size.

4.2 Second version

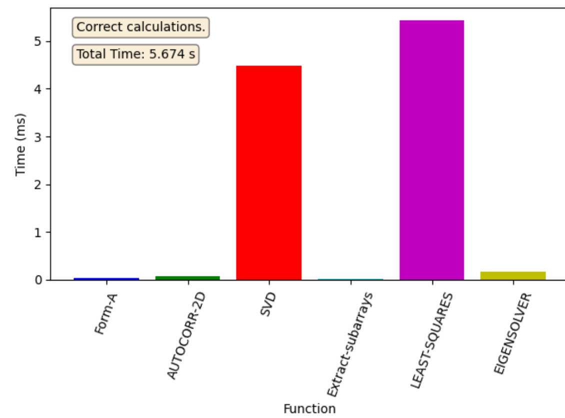
The second version executes 2D-ESPRIT on the entire data set by dividing each range bin on a separate CUDA stream as explained in Section 3.2. In this version, different ratios of threads and CUDA streams were tested to see if an optimal configuration could be found. Here follows a selection of the configurations tested.

4.2.1 One thread with 500 CUDA streams

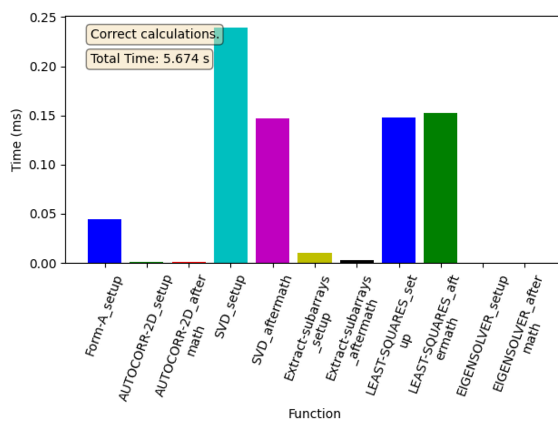
The results for FP32 with a configuration on one thread with 500 CUDA streams with a covariance matrix size of 64 elements are shown in Figure 4.10 and Figure 4.11.



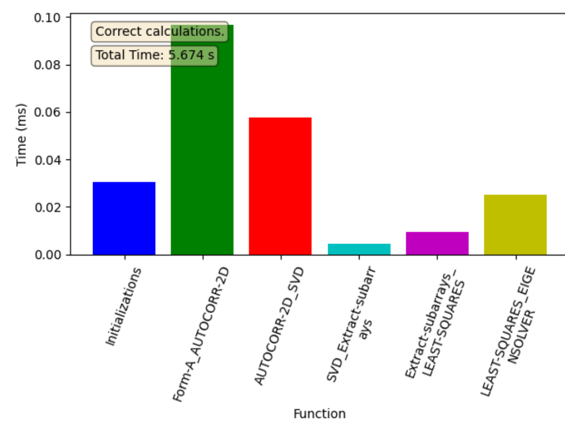
(a) Function mean time



(b) Compute mean time

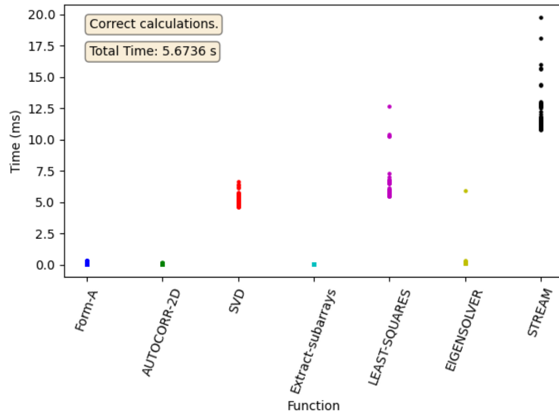


(c) Memory mean time

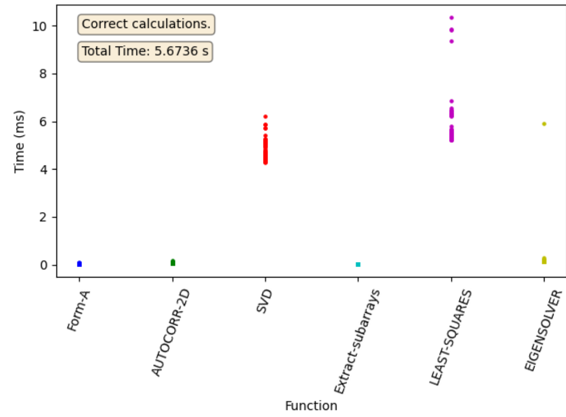


(d) Inter-function mean time

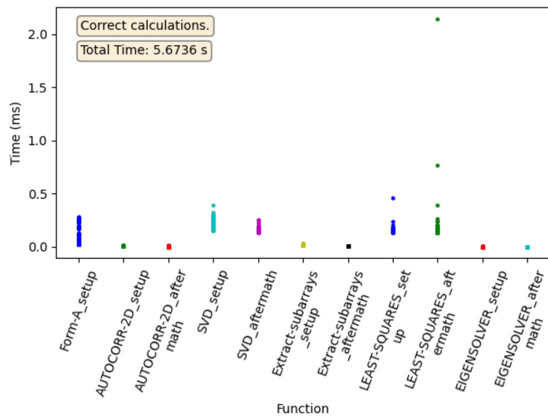
Figure 4.10: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is one thread and 500 CUDA streams.



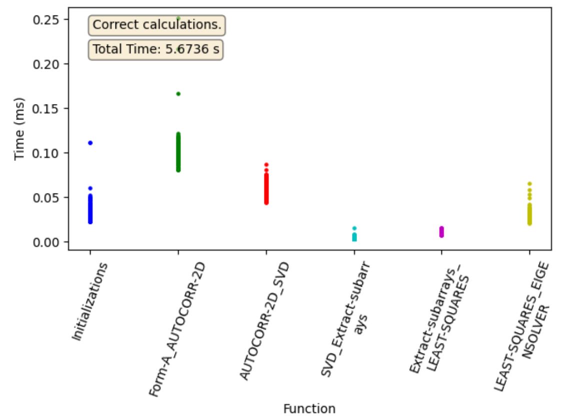
(a) Function mean time



(b) Compute mean time



(c) Memory mean time



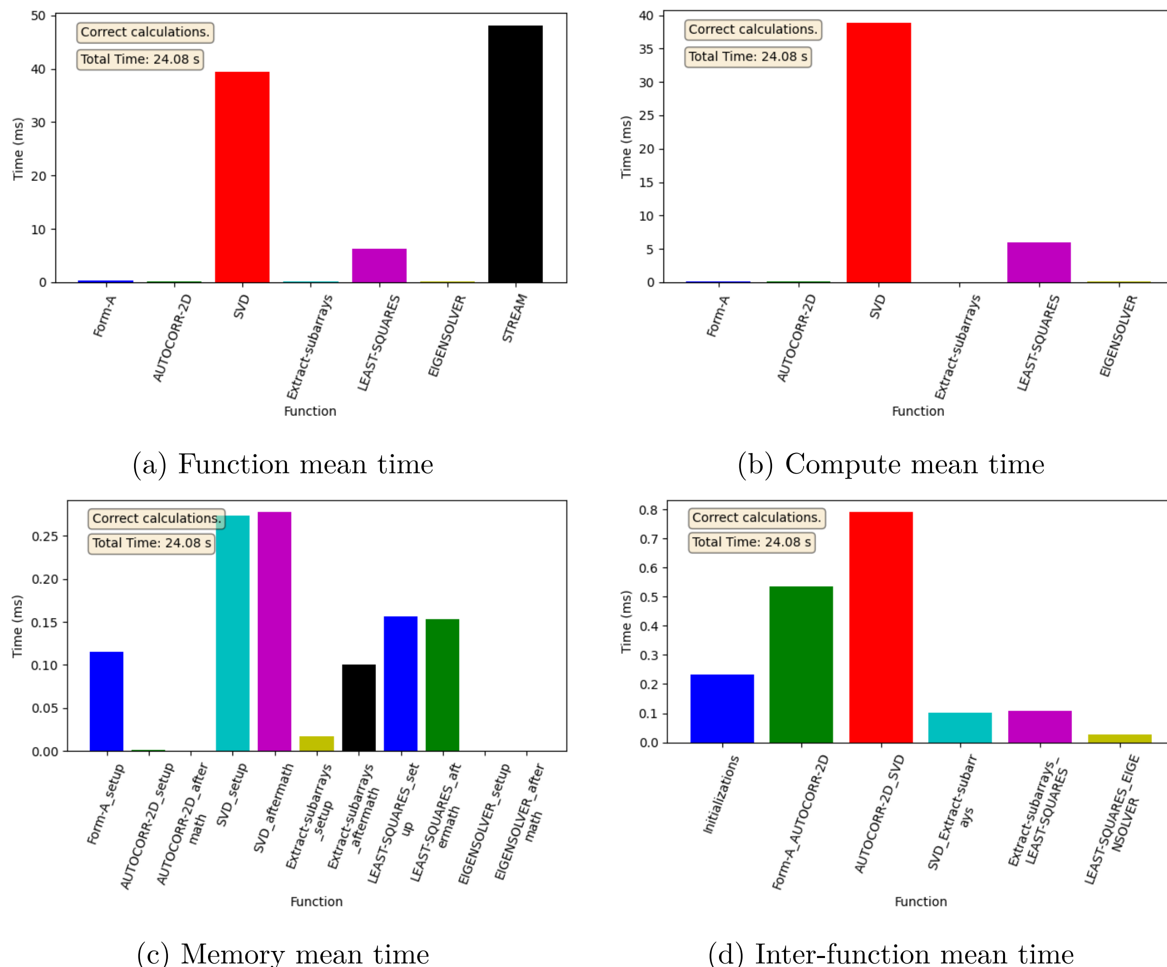
(d) Inter-function mean time

Figure 4.11: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is one thread and 500 CUDA streams.

In the second version, we see similar results to the first version only that the wall-clock time is reduced. The SVD and least-square solver are still the most computationally heavy functions and take the most time. The memory and inter-function times are still very low compared to the computation times.

Some outliers can be observed for the scatter plots, mainly for the least-square solver, resulting in some outliers for the stream times. The SVD has times ranging from 5-7.5 ms. There is one outlier for the eigensolver at approximately 7.5 ms, otherwise the times are low. The remaining functions have very low execution times. The inter-function times are generally very low. Outside from one outlier for the aftermath of the least-square solver in the memory time plot, the memory times are lower than 1 ms.

The results of an increased covariance matrix data size of 360 elements are shown in Figure 4.12 and Figure 4.13.



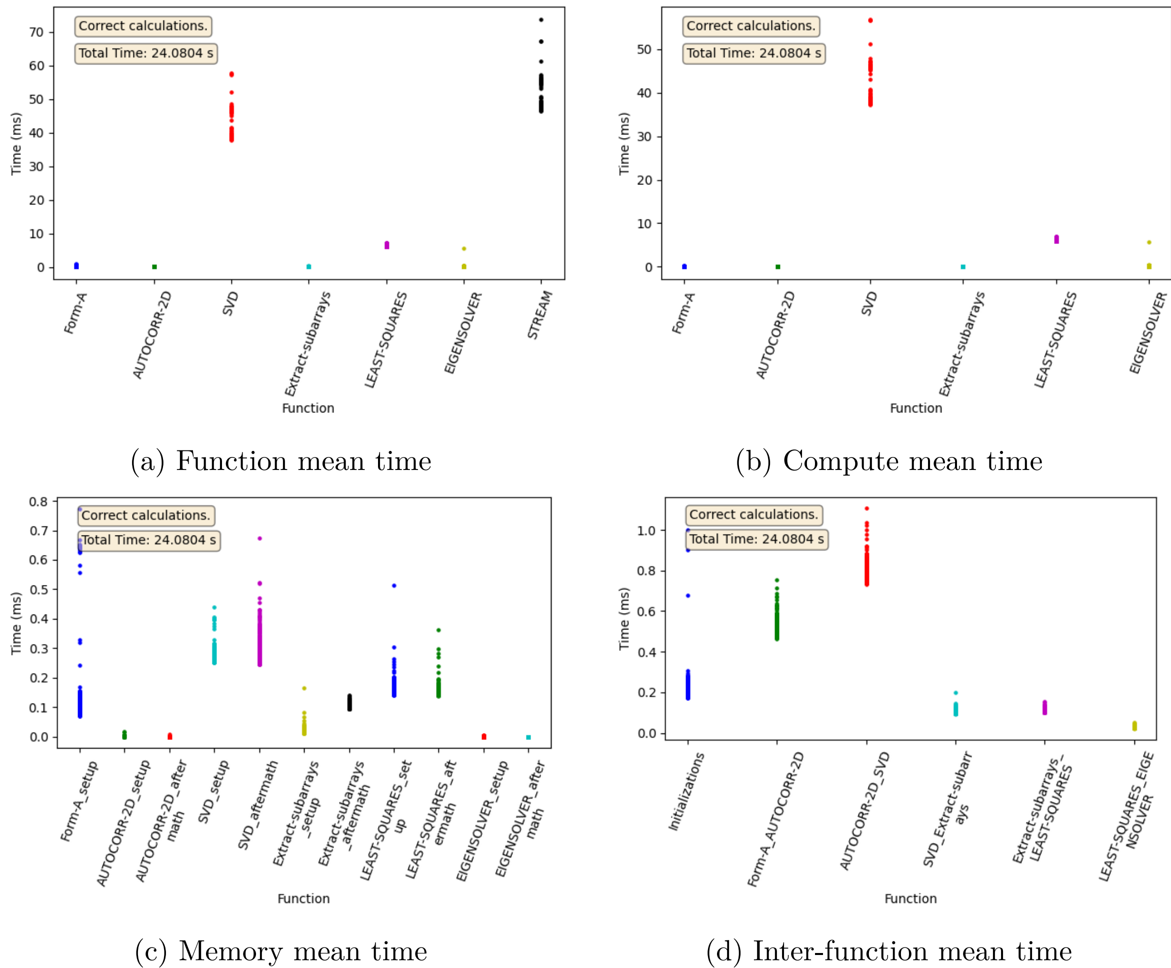
(a) Function mean time

(b) Compute mean time

(c) Memory mean time

(d) Inter-function mean time

Figure 4.12: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is one thread and 500 CUDA streams.



(a) Function mean time

(b) Compute mean time

(c) Memory mean time

(d) Inter-function mean time

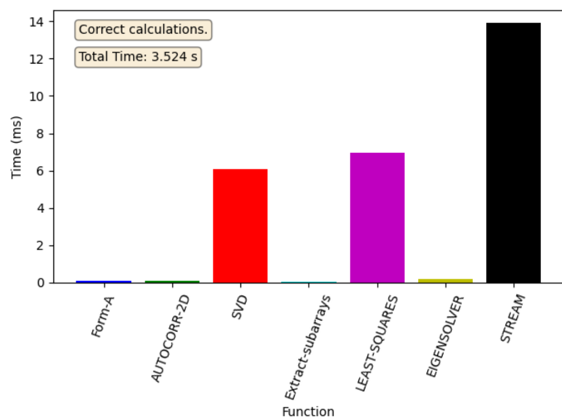
Figure 4.13: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is one thread and 500 CUDA streams.

With an increased covariance matrix data size, we see a steep rise in execution time for the SVD, with a mean time of 40 ms. The least-square solver lies around 8 ms, making the majority of the stream time the SVD. The other functions require very little time in comparison. The memory and inter-function times are all below 1 ms.

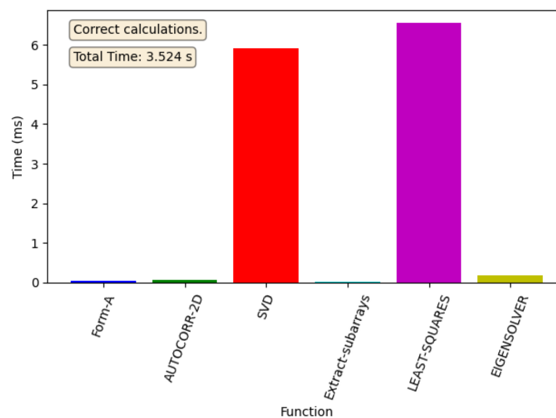
For the scatter plots, the SVD has a wide spread ranging from 35 to 60 ms which also impact the entire stream time. The eigensolver has one outlier at around 10 ms. Otherwise, the execution times for the other functions are quite even and do not have a large spread. Further, the memory and inter-function times do not have any significant outliers and have the max execution times at around 1 ms.

4.2.2 Two threads with 250 CUDA streams each

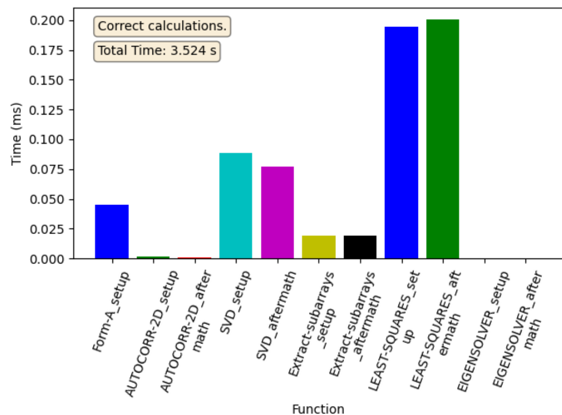
For Figure 4.14 and Figure 4.15, there are two threads which each dispatch 250 CUDA streams with a covariance matrix size of 64 elements on FP32.



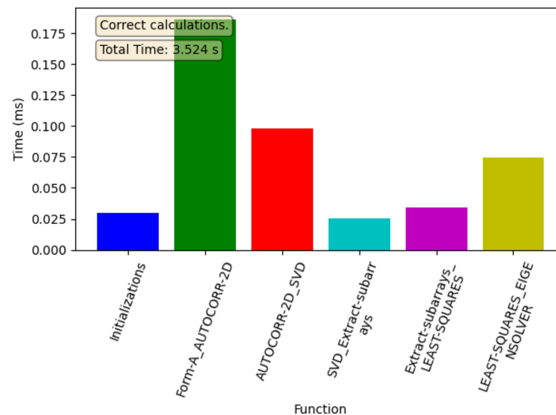
(a) Function mean time



(b) Compute mean time

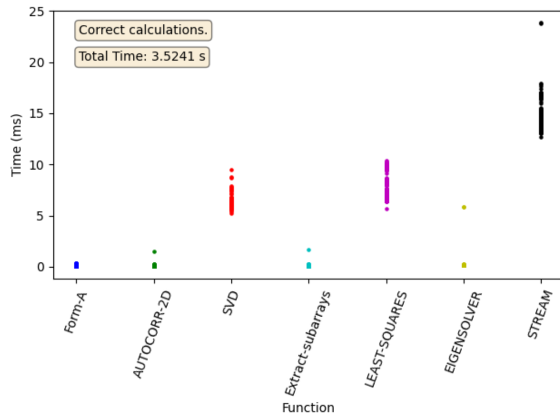


(c) Memory mean time

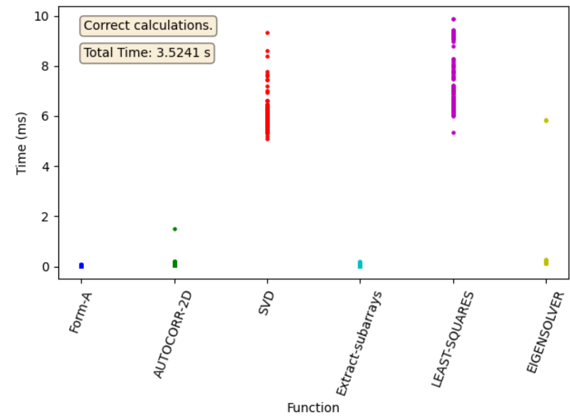


(d) Inter-function mean time

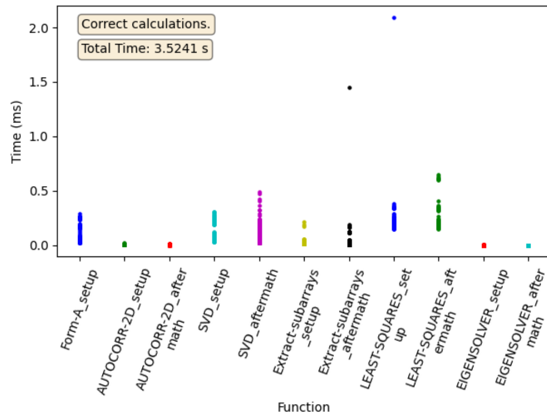
Figure 4.14: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is two threads with 250 CUDA streams per thread.



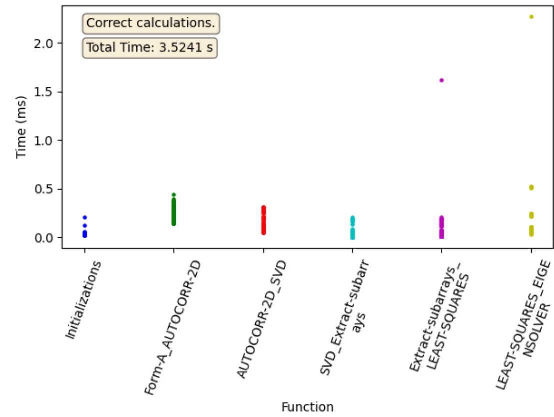
(a) Function mean time



(b) Compute mean time



(c) Memory mean time



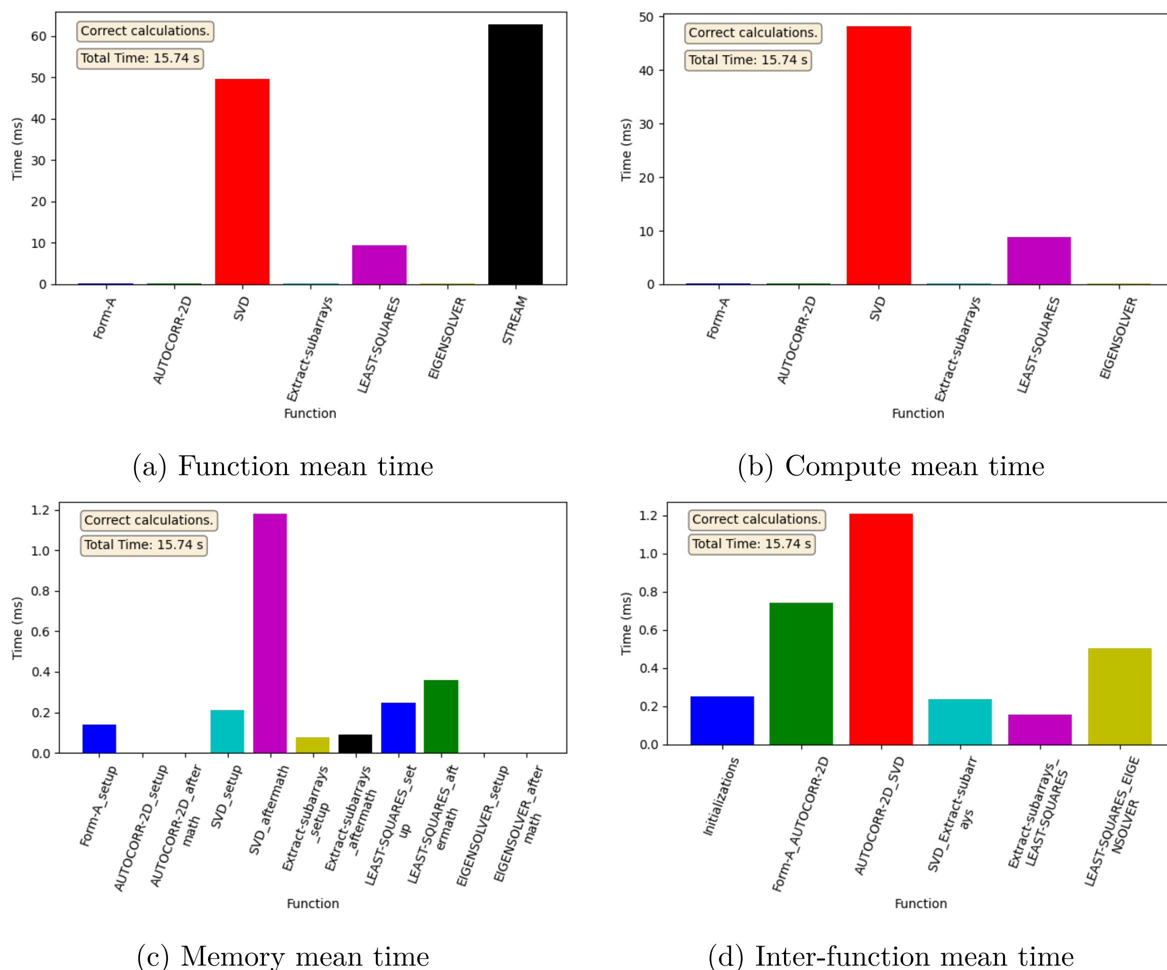
(d) Inter-function mean time

Figure 4.15: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is two threads with 250 CUDA streams per thread.

Here, the total execution time has decreased by approximately two seconds compared to one thread dispatching 500 CUDA streams. The mean execution times of the computation, memory and inter-functions show very similar results with the SVD and least-square solver being the dominant time consumers.

The scatter plot of both the SVD and least-square solver shows quite a wide spread. Some outliers can be observed for the memory and inter-function times which increase the uncertainty in the execution times for an entire stream run.

Figure 4.16 and Figure 4.17 show the results of an increased covariance matrix size of 360 elements on FP32.



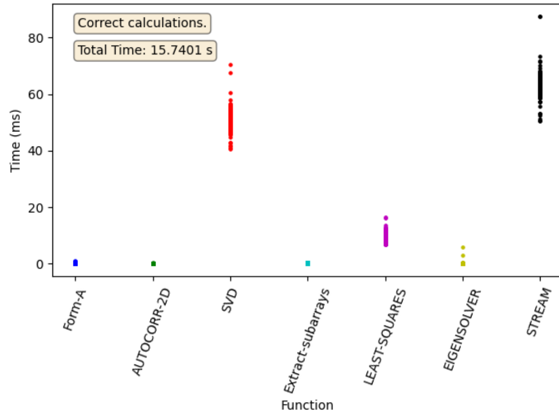
(a) Function mean time

(b) Compute mean time

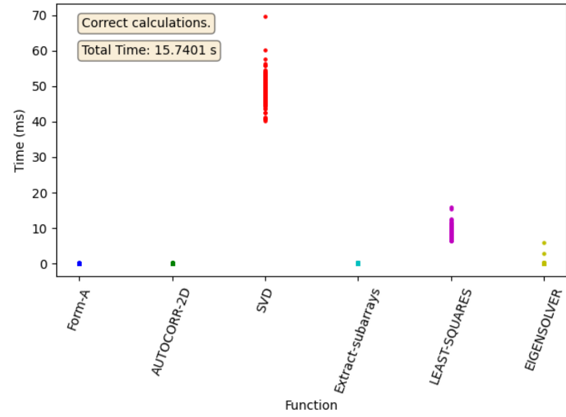
(c) Memory mean time

(d) Inter-function mean time

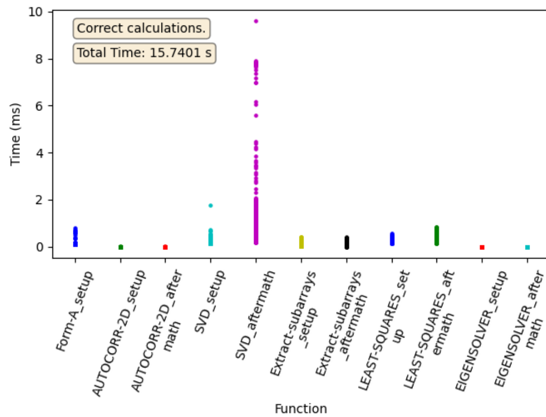
Figure 4.16: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is two threads with 250 CUDA streams per thread.



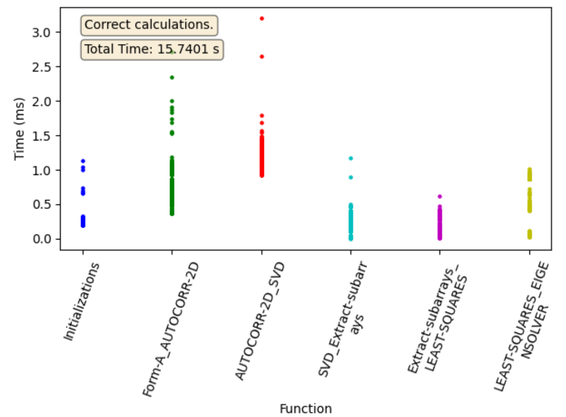
(a) Function mean time



(b) Compute mean time



(c) Memory mean time



(d) Inter-function mean time

Figure 4.17: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is two threads with 250 CUDA streams per thread.

For the increased data size, there is a spike in SVD execution time as before and the least-square solver looks approximately the same.

The scatter plots show quite a wide spread in execution times. The SVD has a wide spread from 40 to 70 ms. The aftermath of the SVD shows a very large spread, ranging from under 1 ms to almost 10 ms. This results in a high variance in the entire stream run time.

4.2.3 Wall-clock times for the second ESPRIT version

In Figure 4.18 we see two plots with the total run-times of different configurations using 64, 180 and 360 elements for the covariance matrix for both FP32 and FP64. The x above some bars in the bar graph show if one or more streams were unable to handle the workload in the given configuration.

(a) FP32 (b) FP64

Figure 4.18: Wall-clock times for each run configuration with both data types and different data sizes using the second version of the algorithm. The legend explains the run's ratio of threads and CUDA streams, where T and S stands for threads and CUDA streams respectively.

What can be observed is that the wall-clock time decreases when dividing the streams on more threads. A significant time reduction in the program's run-time can be observed by just using two threads. However, when reaching 10 threads each running 50 CUDA streams, no significant wall-clock time improvement can be observed, indicating that some sort of improvement saturation has been reached. Also, this configuration fails for all three covariance matrix sizes, which indicate that this version is the most computationally heavy configuration of all tested. By using FP64 instead of FP32, more runs are unable to handle the computations. Further, an increased covariance matrix size also results in more runs being unsuccessful and higher run-times.

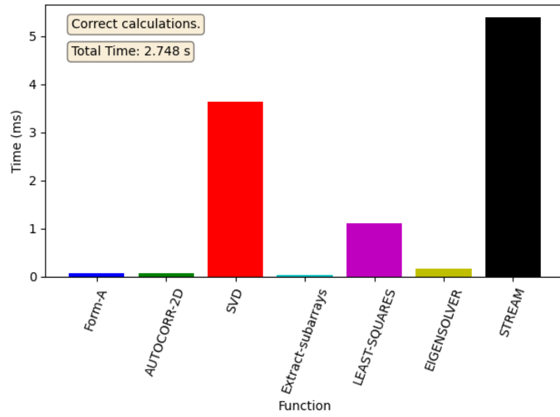
4.3 Third version

The third and final version of the algorithm was implemented with fewer synchronization calls and different parameters for the least-square solver to reduce the times required for the entire run to finish executing. The least-square solver uses iterative refinement to find a more precise solution, which slows down the total run-time of the program. The default number of allowed refinement iterations for the least-square solver is 50, but in this version it is set to five.

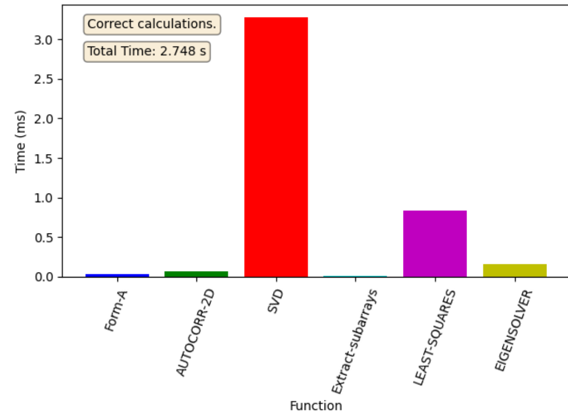
Here are the results for the third and final version.

4.3.1 One thread with 500 CUDA streams

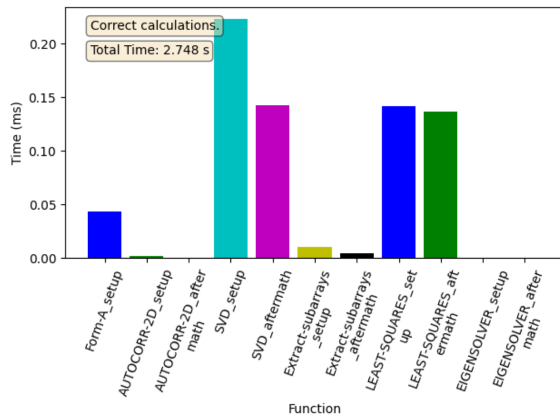
In Figure 4.19 and Figure 4.20, we see the mean times and the scatter plots for the third version's implementation with one thread launching 500 parallel CUDA streams, on a covariance matrix size of 64 elements using FP32.



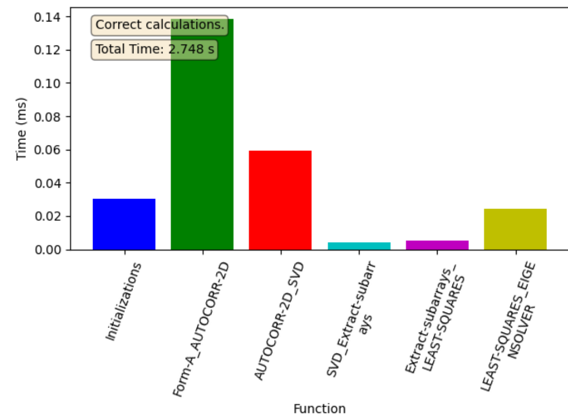
(a) Function mean time



(b) Compute mean time

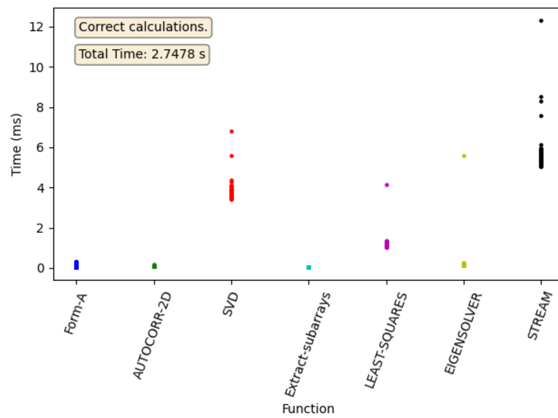


(c) Memory mean time

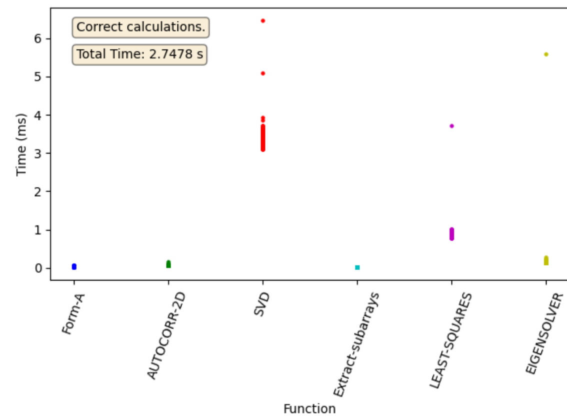


(d) Inter-function mean time

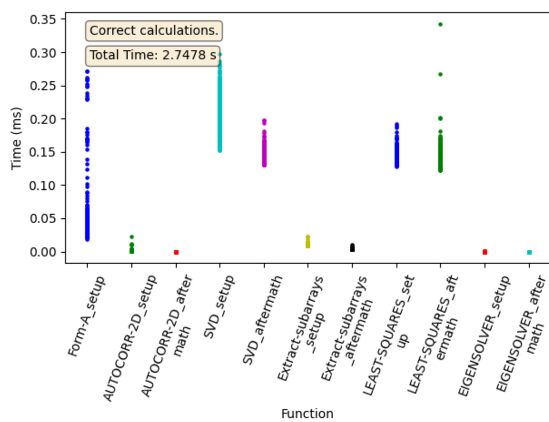
Figure 4.19: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is one thread and 500 CUDA streams.



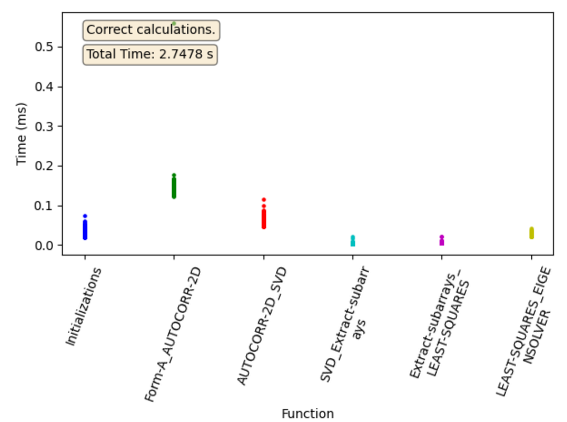
(a) Function mean time



(b) Compute mean time



(c) Memory mean time



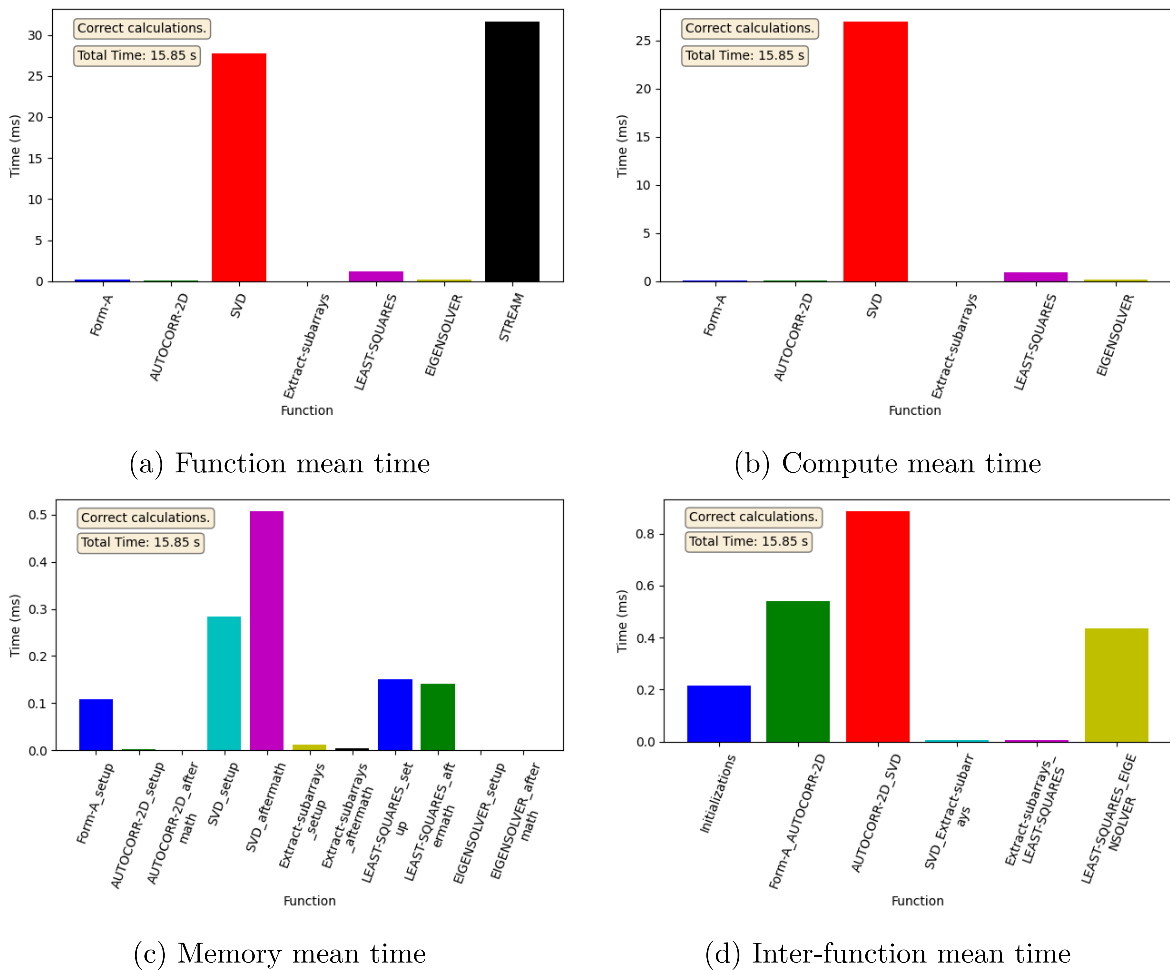
(d) Inter-function mean time

Figure 4.20: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is one thread and 500 CUDA streams.

The SVD require mostly the same execution time as before. However, the least-square solver is cut from a mean time of 7 ms to approximately 1 ms which lowers the total mean stream time to around 5.5 ms. The memory and inter-function times are very similar to that of the same configuration on version 2.

For the scatter plots, there are some outliers for the SVD, least-square solver and the eigensolver which make the stream execution time spread quite a bit. The memory and inter-function times have some spread, but the overall times are very low.

The results of the same configuration but with a larger covariance matrix size of 360 elements are shown in Figure 4.21 and Figure 4.22.



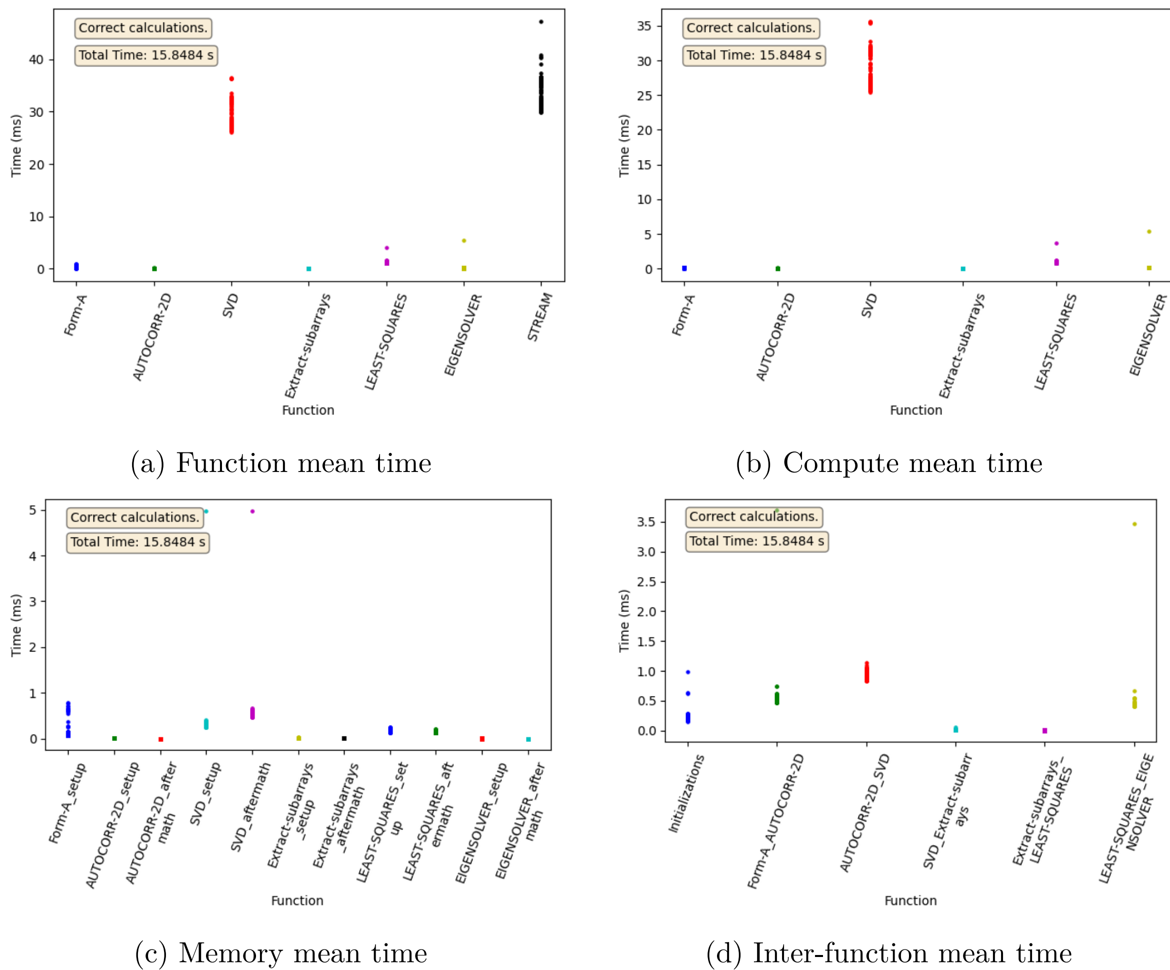
(a) Function mean time

(b) Compute mean time

(c) Memory mean time

(d) Inter-function mean time

Figure 4.21: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is one thread and 500 CUDA streams.



(a) Function mean time

(b) Compute mean time

(c) Memory mean time

(d) Inter-function mean time

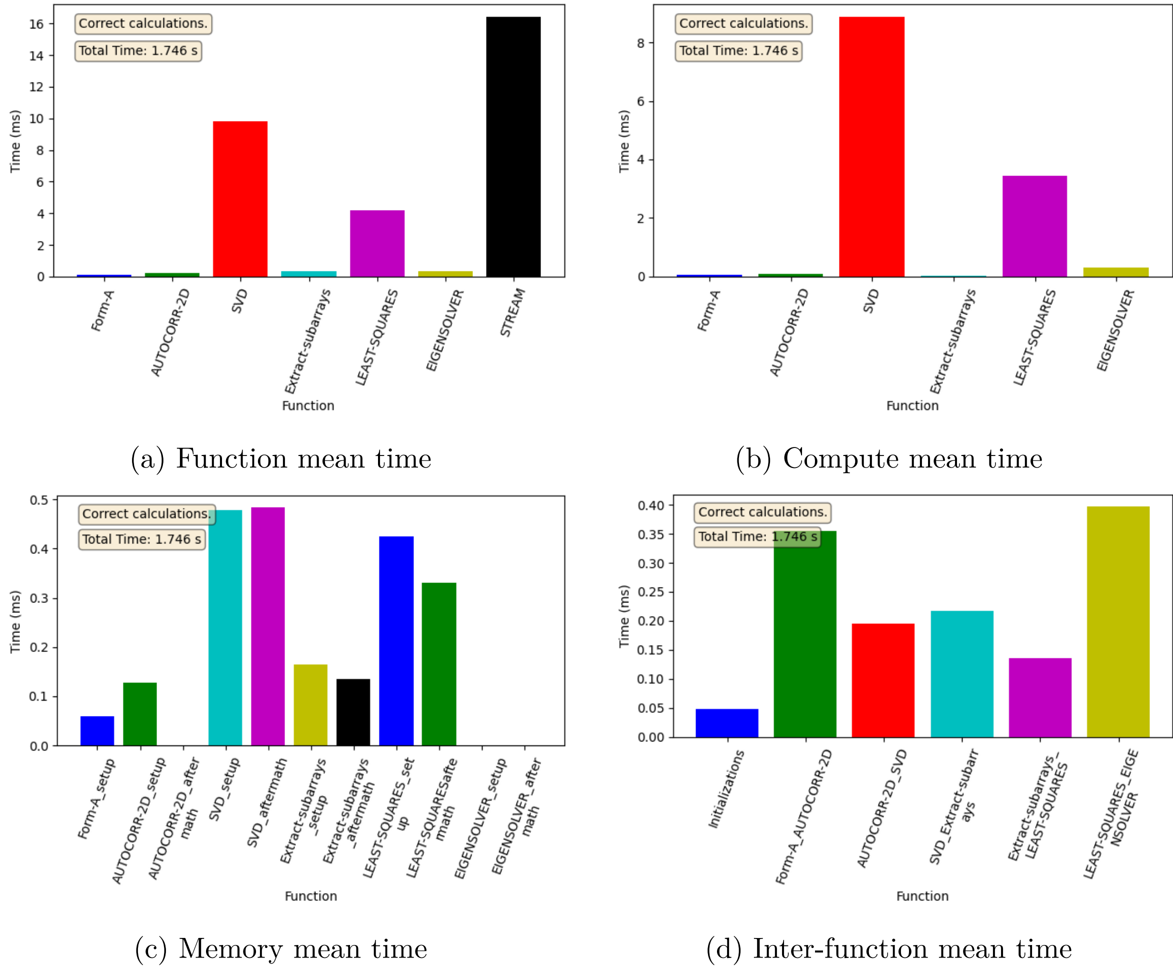
Figure 4.22: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is one thread and 500 CUDA streams.

Here, the SVD mean time increases substantially to around 28 ms. The least-square solver still lies around 1 ms and the other functions require very little time. Thus, the SVD is the dominant factor in the entire stream time which lies around 30 ms. The memory times are low, with a max mean time of half a millisecond for the aftermath of the SVD. The inter-function mean times are slightly higher with the time between the autocorrelation and SVD functions being the highest.

For the scatter plots, we find some outliers for the least-square and eigensolver. The SVD has the largest spread of 25 to 38 ms which also causes some spread for the total stream time. One outlier can be observed in the memory time for setup and aftermath of the SVD, at around 5 ms each. The inter-function times also have two outliers, one between the form-A and autocorrelation functions and one between the least-square and eigensolver functions, both around 3.5 ms.

4.3.2 Five threads with 100 CUDA streams per thread

In this configuration, there are five threads, each dispatching 100 CUDA streams each. The resulting times for a covariance matrix size of 64 elements using FP32 are found in Figure 4.23 and Figure 4.24.



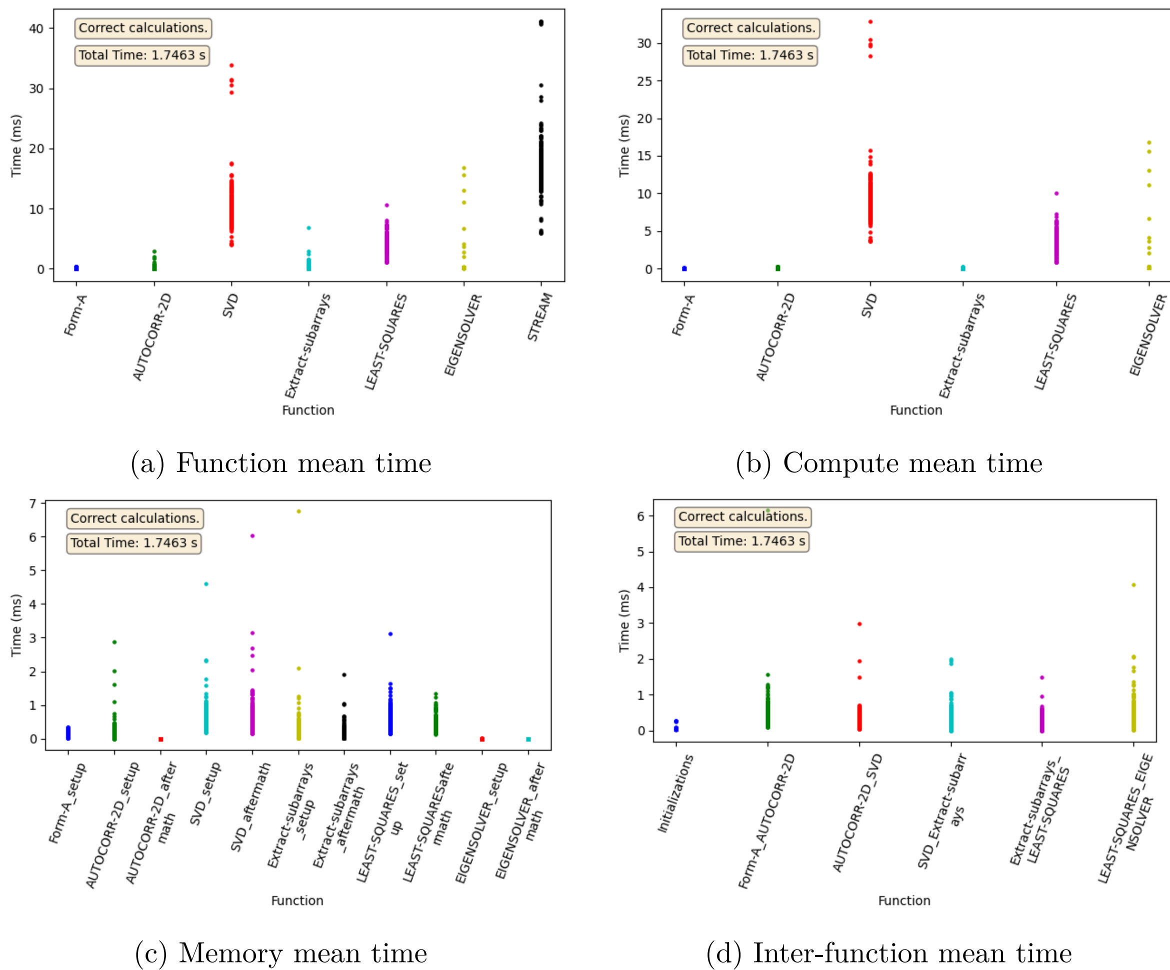
(a) Function mean time

(b) Compute mean time

(c) Memory mean time

(d) Inter-function mean time

Figure 4.23: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 5 threads with 100 CUDA streams per thread.



(a) Function mean time

(b) Compute mean time

(c) Memory mean time

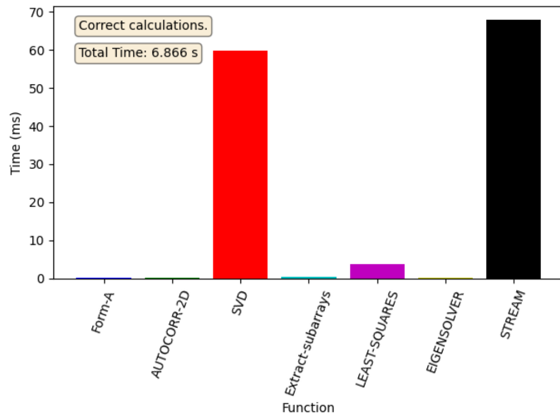
(d) Inter-function mean time

Figure 4.24: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 5 threads with 100 CUDA streams per thread.

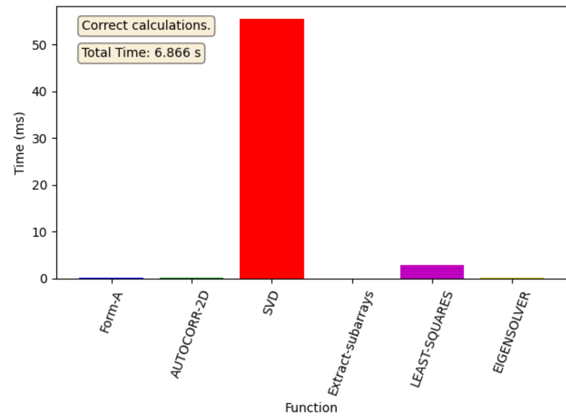
In this configuration, we observe an increased mean computation time for both the SVD and least-square solver compared to the same version with the same covariance data size with one thread launching 500 CUDA streams. However, the total run-time of the program is decreased by almost one second. The mean time for a single stream is around 16 ms. The remaining function means are low. For the memory time, the means do not exceed half a millisecond, and the inter-function mean times do not exceed 0.4 ms.

For the scatter plots, we observe a much larger spread for the SVD, least-square solver and eigensolver functions. The SVD range from 4 to 35 ms, the least-square solver from 1 to 10 ms, and the eigensolver from under 1 to 20 ms. All of this results in a high spread for the stream times, with around 5 ms being the lowest to around 40 ms. The majority of the memory times experience a large spread. Beside from the initialization stage in the inter-function scatter plot, most inter-function times also experience higher spread.

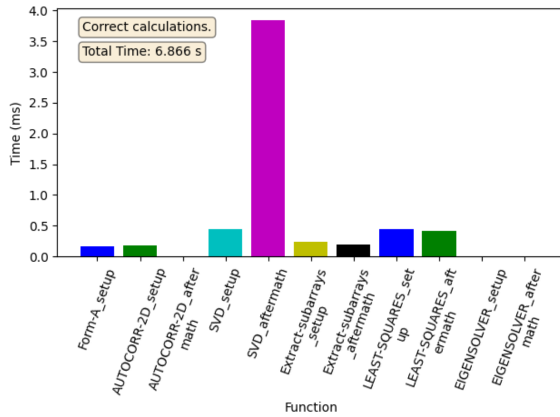
The results of an increased covariance matrix size of 360 elements are found in Figure 4.25 and Figure 4.26.



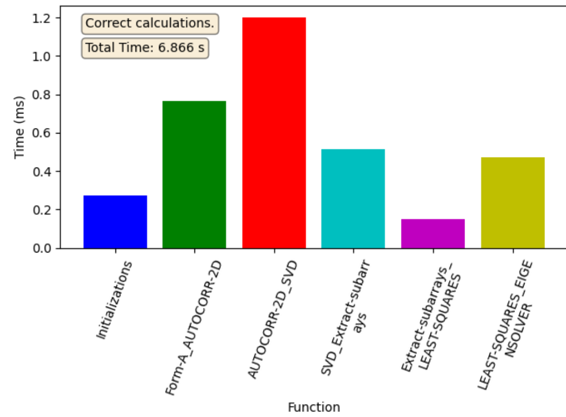
(a) Function mean time



(b) Compute mean time

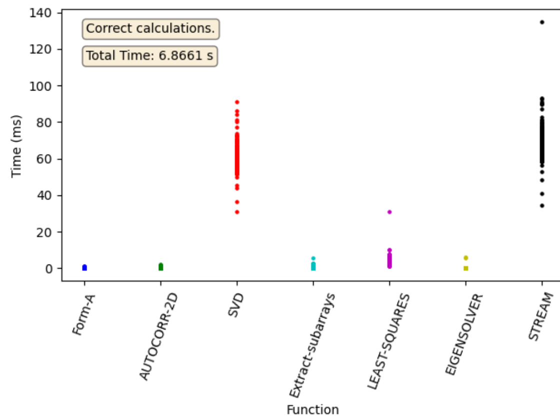


(c) Memory mean time

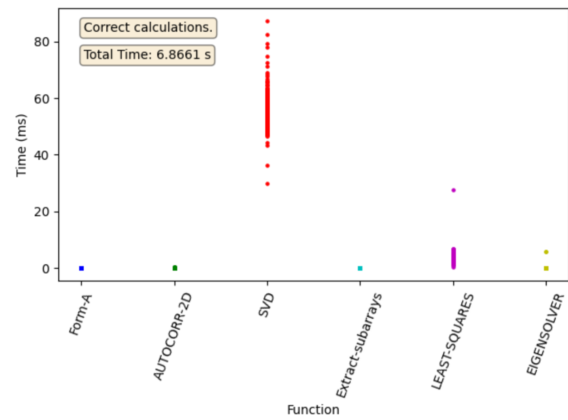


(d) Inter-function mean time

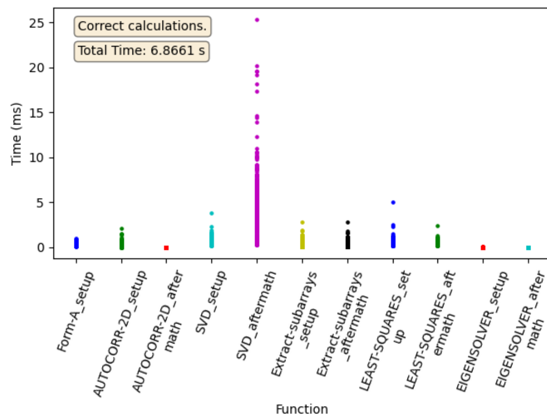
Figure 4.25: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 5 threads with 100 CUDA streams per thread.



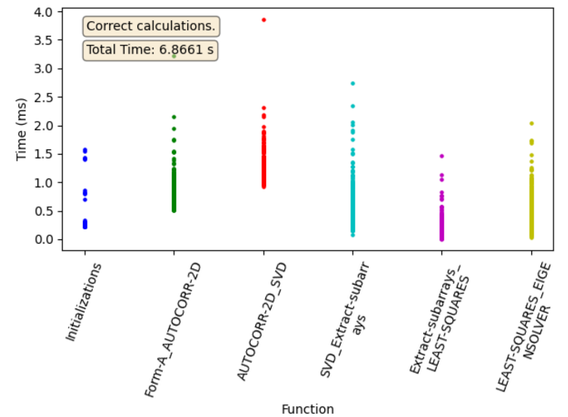
(a) Function mean time



(b) Compute mean time



(c) Memory mean time



(d) Inter-function mean time

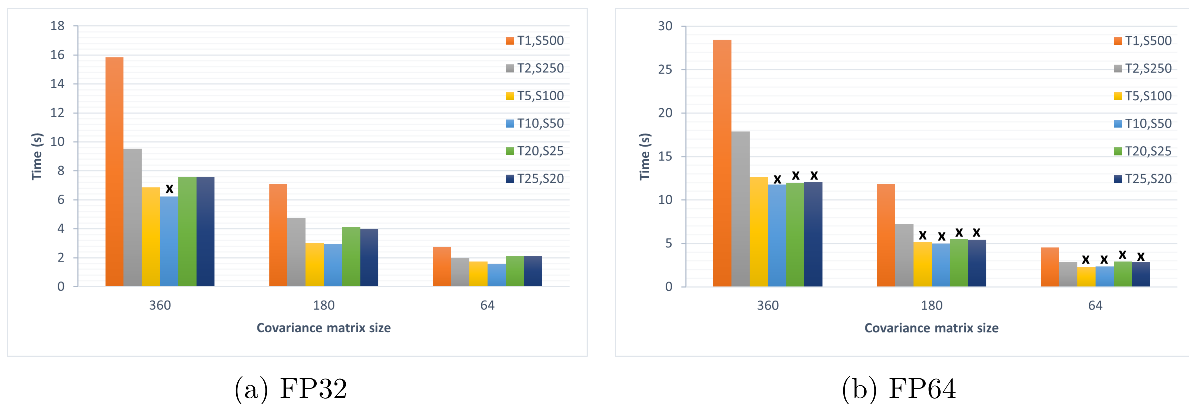
Figure 4.26: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 5 threads with 100 CUDA streams per thread.

Here, the SVD has a steep rise in mean total execution time. The least-square solver lies around 5 ms, making it barely 10 percent of SVD execution time. The aftermath of the SVD also has a large contribution of the execution time, where it lies around 3.7 ms. The other memory functions all lie below 1 ms each. The inter-function times are not high, with the time between the autocorrelation function and the SVD function being the highest at around 1.2 ms. The rest are less than 1 ms.

For the scatter plots, we observe a large variation for different functions, with the SVD having the highest with times ranging from 30 to 100 ms. Also, the aftermath of the SVD has times ranging from below one ms to 25 ms. Also, the variation in the inter-function times has increased, making the predictability of the entire stream time much lower.

4.3.3 Wall-clock times for the third ESPRIT version

The different configurations were executed with both FP32 and FP64 and the total run-times of the programs were recorded. This can be seen in Figure 4.27.



(a) FP32 (b) FP64

Figure 4.27: Wall-clock times for each run configuration with both data types and different data sizes using the third version of the algorithm. The legend explains the run's ratio of threads and CUDA streams, where T and S stands for threads and CUDA streams respectively.

Here, we notice a big drop in total run-time from just having one thread launching 500 CUDA streams to two threads launching 250 CUDA streams each for both data types. Further, we can observe that both five threads launching 100 CUDA streams each and 10 threads launching 50 CUDA streams each provide the best total timing results.

For FP32, most of the configurations can handle the computations except for having the largest covariance matrix size on the 10 threads launching 50 CUDA streams configuration. Otherwise, the remaining FP32 runs were all successful. Using FP64 we see a higher rate of failed runs, where all runs having 10 or more threads fail. Having 5 threads only pass the largest covariance matrix data size and having two or one threads pass all their runs.

4.4 Speed-up

The wall-clock time of each run was logged. This time was compared to investigate the relative speed-up of each version and configuration. The results are found in Table 4.2, Table 4.3, Table 4.4 and Table 4.5.

Table 4.2: Wall-clock time speed-up and comparison from version 1 using FP32 with covariance matrix size 64.

Version	**Configuration**	**Wall-clock time**	**Speed-up from first version**	**Success**
1	1T, 1S	5.569	1	Yes
2	1T, 500S	5.674	1.00	Yes
2	2T, 250S	3.524	1.58	Yes
2	5T, 100S	2.851	1.95	Yes
2	10T, 50S	2.912	1.91	Yes
2	20T, 25S	4.035	1.38	Yes
2	25T, 20S	4.206	1.32	Yes
3	1T, 500S	2.748	2.03	Yes
3	2T, 250S	1.98	2.81	Yes
3	5T, 100S	1.7463	3.19	Yes
3	10T, 50S	1.565	3.56	Yes
3	20T, 25S	2.131	2.61	Yes
3	25T, 20S	2.127	2.62	Yes

Table 4.3: Wall-clock time speed-up and comparison from version 1 using FP32 with covariance matrix size 360.

Version	**Configuration**	**Wall-clock time**	**Speed-up from first version**	**Success**
1	1T, 1S	24.73	1	Yes
2	1T, 500S	24.08	1.03	Yes
2	2T, 250S	15.74	1.57	Yes
2	5T, 100S	11.13	2.22	No
2	10T, 50S	10.57	2.34	No
2	20T, 25S	11.57	2.14	No
2	25T, 20S	11.22	2.20	No
3	1T, 500S	15.85	1.56	Yes
3	2T, 250S	9.52	2.60	Yes
3	5T, 100S	6.866	3.60	Yes
3	10T, 50S	6.218	3.98	No
3	20T, 25S	7.561	3.27	Yes
3	25T, 20S	7.586	3.26	Yes

Table 4.4: Wall-clock time speed-up and comparison from version 1 using FP64 with covariance matrix size 64.

Version	Configuration	Wall-clock time	Speed-up from first version	Success
1	1T, 1S	11.1	1	Yes
2	1T, 500S	10.78	1.03	Yes
2	2T, 250S	6.532	1.70	Yes
2	5T, 100S	4.537	2.45	Yes
2	10T, 50S	4.766	2.33	No
2	20T, 25S	5.762	1.93	Yes
2	25T, 20S	6.013	1.85	Yes
3	1T, 500S	4.52	2.46	Yes
3	2T, 250S	2.886	3.85	Yes
3	5T, 100S	2.253	4.93	No
3	10T, 50S	2.357	4.71	No
3	20T, 25S	2.911	3.81	No
3	25T, 20S	2.893	3.84	No

Table 4.5: Wall-clock time speed-up and comparison from version 1 using FP64 with covariance matrix size 360.

Version	Configuration	Wall-clock time	Speed-up from first version	Success
1	1T, 1S	41.56	1	Yes
2	1T, 500S	39.53	1.05	Yes
2	2T, 250S	25.94	1.60	Yes
2	5T, 100S	18.72	2.22	No
2	10T, 50S	18.7	2.22	No
2	20T, 25S	19.22	2.16	No
2	25T, 20S	19.69	2.11	No
3	1T, 500S	28.46	1.46	Yes
3	2T, 250S	17.9	2.32	Yes
3	5T, 100S	12.61	3.30	Yes
3	10T, 50S	11.78	3.53	No
3	20T, 25S	11.92	3.49	No
3	25T, 20S	12.06	3.45	No

The results show that we obtain at most a speed-up of 4.93 for the final version compared to the first implementation. However, this peak speed-up was achieved on a run that did not pass all its computations. So, the actual peak speed-up while still retaining full functionality would be around 3.6 for FP32 with both covariance matrix size of 64 and 360, and 3.85 and 3.3 for FP64 with covariance matrix sizes 64 and 360 respectively.

4.5 Utilization

With the wall-clock time of the program and the estimated total FLOPS of the program, as explained in Section 2.6, we estimate the utilization of the GPU. The data sheet of the Quadro RTX 6000 does not specify peak TFLOPS for FP64, therefore it is difficult to find the theoretical peak FLOPS for FP64. However, the GPU promises 16.3 peak

TFLOPS for FP32 [13] and has one FP64 for every 32 FP32 processing units. If we assume that peak FP64 computations is $\frac{16.3TFLOPS}{32} = 509GFLOPS$, we can calculate the utilization for FP64. Thus, we achieve the estimated utilizations listed in Table 4.6, Table 4.7, Table 4.8 and Table 4.9.

Table 4.6: Total utilization using FP32 with different configurations with covariance matrix size 64.

Version	**Configuration**	**Wall-clock time**	**Utilization**	**Success**
1	1T, 1S	5.569	0.21%	Yes
2	1T, 500S	5.674	0.20%	Yes
2	2T, 250S	3.524	0.33%	Yes
2	5T, 100S	2.851	0.40%	Yes
2	10T, 50S	2.912	0.40%	Yes
2	20T, 25S	4.035	0.29%	Yes
2	25T, 20S	4.206	0.27%	Yes
3	1T, 500S	2.748	0.42%	Yes
3	2T, 250S	1.98	0.58%	Yes
3	5T, 100S	1.7463	0.66%	Yes
3	10T, 50S	1.565	0.74%	Yes
3	20T, 25S	2.131	0.54%	Yes
3	25T, 20S	2.127	0.54%	Yes

Table 4.7: Total utilization using FP32 with different configurations with covariance matrix size 360.

Version	**Configuration**	**Wall-clock time**	**Utilization**	**Success**
1	1T, 1S	24.73	0.82%	Yes
2	1T, 500S	24.08	0.84%	Yes
2	2T, 250S	15.74	1.28%	Yes
2	5T, 100S	11.13	1.81%	No
2	10T, 50S	10.57	1.91%	No
2	20T, 25S	11.57	1.75%	No
2	25T, 20S	11.22	1.80%	No
3	1T, 500S	15.85	1.27%	Yes
3	2T, 250S	9.52	2.12%	Yes
3	5T, 100S	6.866	2.94%	Yes
3	10T, 50S	6.218	3.25%	No
3	20T, 25S	7.561	2.67%	Yes
3	25T, 20S	7.586	2.66%	Yes

Table 4.8: Total utilization using FP64 with different configurations with covariance matrix size 64.

Version	Configuration	Wall-clock time	Utilization	Success
1	1T, 1S	11.1	3.33%	Yes
2	1T, 500S	10.78	4.87%	Yes
2	2T, 250S	6.532	5.65%	Yes
2	5T, 100S	4.537	8.13%	No
2	10T, 50S	4.766	7.74%	Yes
2	20T, 25S	5.762	6.41%	Yes
2	25T, 20S	6.013	6.14%	Yes
3	1T, 500S	4.52	8.17%	Yes
3	2T, 250S	2.886	12.79%	Yes
3	5T, 100S	2.243	16.45%	No
3	10T, 50S	2.357	15.66%	No
3	20T, 25S	2.911	12.68%	No
3	25T, 20S	2.893	12.76%	No

Table 4.9: Total utilization using FP64 with different configurations with covariance matrix size 360.

Version	Configuration	Wall-clock time	Utilization	Success
1	1T, 1S	41.56	26.12%	Yes
2	1T, 500S	39.53	16.34%	Yes
2	2T, 250S	25.94	24.90%	Yes
2	5T, 100S	18.72	34.50%	No
2	10T, 50S	18.7	34.54%	No
2	20T, 25S	19.22	33.61%	No
2	25T, 20S	19.69	32.80%	No
3	1T, 500S	28.46	22.69%	Yes
3	2T, 250S	17.9	36.08%	Yes
3	5T, 100S	12.61	51.22%	Yes
3	10T, 50S	11.78	54.83%	No
3	20T, 25S	11.92	54.19%	No
3	25T, 20S	12.06	53.56%	No

The utilizations for FP32 peaks with the configuration of 5 threads and 100 CUDA streams per thread for the third version while still passing all computations. The results for FP64 must be taken with a grain of salt, since here we assume that we can achieve peak FLOPS of 509 GFLOPS, but no specification has been made by NVIDIA. Nevertheless, the peak for FP64, while passing all computations, is with the configuration 2 threads with 250 CUDA streams per thread for the third version.

5

Discussion

This chapter discusses the results of the thesis conclusions that can be drawn. There is also a section that covers societal, ethical, and ecological aspects.

5.1 Version results

For each implemented version, the wall-clock time for handling the same data set is decreased. This is due to increased parallelism by dividing the data set on CUDA streams which can be seen by comparing the wall-clock times of version one, two and three in figures 4.9, 4.18 and 4.27 respectively. Moreover, changing the least-square solver to only have five allowed refinement iterations proved to reduce time further when comparing the second to the third version. However, this change comes at the expense of the accuracy of the algorithm.

The mean times of the functions increase when using FP64 instead of FP32 as seen in most mean charts. This is to be expected due to both the data being larger as a result, and that the GPU has more FP32 compute units than FP64. However, the results from the first version's wall-clock times in Figure 4.9 show that FP32 is approximately only twice as fast as when using FP64. Since there are 32 FP32 per one FP64 computation unit, this leads us to believe that the library functions do not always utilize the hardware capabilities of the GPU.

There seems to be a large drop in required execution time when threads divide the range bin runs on CUDA streams. However, there is a limit at 10 threads with 50 CUDA streams per thread. By using more threads than that, the total run-time increases. To find an optimal configuration, there seems that the user must experiment to find an optimal configuration. Moreover, more threads with fewer CUDA streams per thread do not automatically correlate with a faster execution time and a more utilized GPU.

The memory transfer times generally show that there is not much transfer time of data compared to computation time. Since it has been shown that it is favorable for a GPU to have a high computational effort contra a high memory transfer rate, the implementations play into the GPUs strengths.

However, for each improved version, the mean time for each function used is generally increased even though the wall-clock time is decreased. This is arguably due to the GPU processing multiple instances of each function simultaneously, but not caring to process them in order. This means that the total workload on the GPU is increased but at the expense of worse throughput latency for each range bin, prioritizing total execution time of the program but not total execution time of each range bin.

By comparing the versions, we find relative speedups which are shown in tables Table 4.2, Table 4.3, Table 4.4 and Table 4.5. Here, we see that if the GPU is pushed too hard with either a large data size or many threads dispatching many CUDA streams, the run can fail which is undesirable. Thus, it is important to explore different configurations and data sizes to achieve peak run-time of the program while still ensuring proper functionality.

5.2 Utilization

When estimating the utilization of the different versions with different configurations on FP32, we find that the first version achieves a very low utilization of the GPU. Then, with succeeding versions, we find an increasing estimated utilization with different configurations as can be seen in Table 4.6 and Table 4.7. We achieve the best estimated utilization for the third version with five threads and 100 CUDA streams with covariance matrix size of 360 elements, at 2.94 % utilization. It is important to note, however, that the third version has fewer refinement iterations for the least-square solver which also impacts the number of total FLOPS compared to the first and second versions. It is not specified what a refinement iteration requires in terms of FLOPS and can therefore not be estimated. Thus, the first and second versions can contain more computations than the third version, but exactly how many more FLOPS of computations this results in can not be estimated. Fewer refinement iterations can also impact the accuracy of the solver. This trade-off was, however, not investigated further.

When examining the estimated utilization of the different configurations on FP64, we see a radically increased utilization compared to FP32. The most utilization is reached is 51.22% utilization for the third version using 5 threads with 100 CUDA streams on a convariance matrix size of 360 elements as seen in Table 4.9. The calculated utilizations of the FP64 runs must, however, be taken with a grain of salt, since the theoretical FLOPS are not stated by NVIDIA, only assumed to be $\frac{1}{32}$ that of the FP32 theoretical FLOPS. The theoretical FLOPS can be much higher, which would vastly impact the calculated utilization. Therefore, most assumptions are based on the FP32 results since these are more reliable and because the GPU is more tailored at handling FP32 computations than FP64.

The dominant function in the algorithm is the SVD which increases in computation time as the size of the covariance matrix size grows. This, in turn, affects the utilization of the algorithm. If the SVD is improved, the total utilization of the GPU can be increased massively. We also see that with increasing data size, that the total utilization of the GPU

increases when comparing the utilization of the runs with different covariance matrix sizes. This leads us to believe that the SVD function has overhead which impacts the wall-clock time. When increasing the data size and thus the computational effort, the impact of the overhead decreases, and the utilization increases.

There does not exist an eigensolver implemented on the GPU that handles asymmetrical matrices, which made the function being executed on the CPU instead. This makes the estimated utilization numbers slightly lower in reality. However, since the eigensolver step of the algorithm is not the most heavy step from a computing perspective, it was deemed reasonable to include it in the utilization estimate of the entire algorithm to be able to use the total run-time of the program.

Moreover, the other functions used in the algorithm, except for the least-square solver, does not require much time. This can indicate that the GPU is a valuable asset with good computing capabilities for some function implementations in a radar sensor digital processing chain.

5.3 Issues

One of the main issues of the work was that there were not many tools for evaluating the performance of the GPU. NVIDIA provides some, mainly NVIDIA NSight systems with its subsystems [16]. These, however, do not provide very much information of the given library functions used throughout this project. It is therefore very difficult to identify points of improvement in the code to further increase GPU utilization.

By not being able to analyze the code and extract the utilization, there are many difficulties in improving the code and analyzing to what extent the GPU is utilized. Therefore, the work was limited to working with timers placed in the code and logging the time required for executing blocks of code. This works quite well when having one thread executing several CUDA streams. When multithreading, however, the timers start some irregular behavior, where the total run-time of the program decreases while the mean-time of all functions increases. One hypothesis is that the GPU manages many tasks in parallel but does not care about throughput, so when many tasks get dispatched to the GPU it works on many tasks at once but does not finish tasks in order. Thus, the GPU has a higher active time which lowers the total run-time of the program, even though the mean-time of the functions increases.

Further, when many threads dispatch many CUDA streams at once to the GPU, the GPU becomes overloaded and can not handle all computations, making some of the range bin runs fail. This is backed by the results of Figure 4.18 and Figure 4.27. Here, the main differences are the lower allowed max iterations for the least-square solver (lower computational burden) and less redundant synchronization. For the same configurations, the third version passes more runs than the second run, hinting that the GPU becomes overloaded in refinement iterations and thus is unable to handle the entire computational

burden. Also, the amount of failed runs increases when using a larger data size and number representation.

A final point concerning the execution times is that the timers show a large spread of each function's computation time when using multiple threads and CUDA streams. Therefore, the throughput of a single range bin's digital processing chain is very unpredictable which is undesirable from a data processing viewpoint. This also impacts the total run-time of each program, making it difficult to foresee how much time is required to process all of the data.

Another issue was the lack of control of function calls. The major bottleneck of the algorithm is the SVD, that has a steep rise in its computational burden when the covariance matrix data size increases. Therefore, to further push down time of each stream and improve the throughput of the digital processing chain, the SVD has to become more efficient. However, the cuSOLVER library does not offer many ways of parameterizing the function other than using similar implementations of the same function which do not differ much in terms of efficiency. There is also no clear way of analyzing the function to know what happens "under the surface". Therefore, it is very difficult to improve this bottleneck without implementing the function from scratch which was beyond the scope of this thesis work and can prove to be difficult and time consuming from an engineering standpoint.

Yet another difficulty during the thesis work was that the computer which ran the simulations were shared by multiple users. There were several Quadro RTX 6000 GPUs which could be used, so it was important before a run to see which one was available to acquire more reliable results. However, the CPU on the computer was shared which impacted the total run-time massively from run to run of the same program. It was thus decided that each thread be assigned to its own processor core. This made the program sometimes not achieve the fastest time from run to run, but the results were stable which was deemed more important.

5.4 Improvements and future work

Several improvements can be made to further decrease the required run-time of the programs and thus increase the efficiency of the algorithm and GPU. The largest and most critical improvement is to develop a custom-made SVD or alternative that is more suited to handle the radar data. The function has proved to be the largest bottleneck and time consumer of the algorithm and is thus the most important function to work on. The other function, the least-square solver, is also worth tweaking and looking over since it also has a substantial contribution to a range bin's total execution time. There are several versions of this algorithm already implemented that can turn out to be better than the two tested from the first version and the second and third versions respectively. There are also SVD implementations that are batched, where several SVDs are performed on several matrices in parallel, which can further increase the computational effort of the algorithm and thus increase the utilization of the GPU.

Further, there are other versions of the algorithm already implemented in the cuSOLVER library which may prove to decrease the time required for the SVD. However, investigating other implementations was out of the scope of this master thesis, due to the time it would require to investigate not only the execution times of the implementations, but also the accuracy of the results.

Looking at the data in Table 2.2, we see that with a data size of 360 elements, the SVD would require $6 \times 21 \times 360^3$ FLOPS. By using the SVD computation time in the third version using five threads each dispatching 100 CUDA streams, which was approximately 60 ms, and the peak TFLOPS promised for the GPU, we acquire an approximated utilization of the SVD function:

$$U_{SVD} = \frac{6 \times 21 \times 360^3}{0.06 \times 16.3 \times 10^{12}} = 0.006 = 0.6\%. \tag{5.1}$$

Therefore, to increase the utilization of the function further, we significantly need a faster SVD, or an increased data set size. An increased data size could be achieved by using a batched version of the algorithm, which currently exists. The batched version was tested with a dummy data set and showed great promise, but was not implemented in the 2D-ESPRIT algorithm. This was mainly due to the pipelining of the 2D data slices, which would require all computations to stall until sufficient threads reached the SVD stage of the algorithm to perform a large computation of several SVDs at once. This would essentially ruin the pipeline of the algorithm and was thus not investigated further. Perhaps the data set of this implementation does not map very well on the GPU, but would greatly benefit from having a larger covariance matrix size which would increase the utilization of the GPU.

Another improvement of this thesis work is to test the algorithm on different GPUs. GPUs are constructed to be portable, meaning that code written for older GPUs should also work on newer and more powerful GPUs with some small modifications. It would be interesting to see how the algorithm, and some functions, map on more powerful GPUs.

Another future work for this thesis is to test different algorithms other than ESPRIT. To further investigate GPU utilization, other algorithms can map better and yield better utilization results.

There are perhaps better tools, other than timers, for analyzing code on the GPU that will grant more insight into how to improve the algorithm further. It would be interesting to dive deeper and explore other ways to analyze the code and performance of the GPU.

When analyzing the scatter plots generated, we see that many functions have varying execution times. Another improvement would be to make the functions more predictable

to make the performance of the entire algorithm more stable.

Despite the various issues and areas of improvement discussed, the CUDA platform for interfacing the CPU and GPU with each other is highly convenient. It provides the engineer with a relatively easy syntax to use the GPU without a very steep learning curve and much cumbersome setup work.

5.5 Societal, ethical and ecological aspects

The work revolves around improving the computing efficiency of radar data. This radar technology is used in military technology, which might be ethically questionable. However, one might argue that state-of-the-art military technology is required for a country as a defense strategy to deter international conflicts. Also, radar is used for surveillance to keep track of what is happening in the air space and is not actively used to instigate conflicts. Radar is nonetheless used in weapon systems and is therefore ethically questionable.

6

Conclusion

This thesis project has focused on the implementation of the 2D-ESPRIT algorithm on the Quadro RTX 6000 to investigate if GPUs are suitable for handling the demanding digital signal processing chain in radar applications. It was found that the algorithm could be implemented on the GPU. By splitting the data and assigning it to different threads with corresponding CUDA streams, we decreased the wall-clock time of the algorithm, which further utilized the computing resources of the GPU. However, the spread of execution times increased with more threads, making the programs' execution times much more unpredictable.

The reliance on carefully placed timers in the code for analyzing the CUDA code made the development work difficult, due to the large function execution time spread. Further, the available functions required for the implementation of the 2D-ESPRIT algorithm are "black boxes" where the user does not have the option to analyze points of improvement or modify them to improve the utilization of the GPU. Despite this, CUDA simplifies the interface between the CPU and the GPU and can be an excellent tool to use in future projects to investigate the performance of the GPU.

Finally, the utilization could be increased by using threads and CUDA streams to increase parallelism. However, the SVD dominates the computation time and effort and must be improved upon to make significant improvements to the total execution time of the program which would also result in and increase in GPU utilization. This could potentially be done by implementing a batched version of the routine which would compute the SVD of several 2D data slices at once. Other versions of the SVD could also be investigated in future works to find one that perhaps work better with the covariance matrix size used in this work. For the remaining functions, they have low execution times which show promise for the GPU to be used in radar signal processing chains.

Bibliography

[1] Intel, "CPU vs. GPU: Powerful options for your computing needs.," Accessed: 24/01-2024 [Online], Available: https://www.intel.com/content/www/us/en/products/docs/processors/cpu-vs-gpu.html.

[2] R. Roy and T. Kailath, "ESPRIT-Estimation of Signal Parameters via Rotational Invariance Techniques," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 7, pp. 984–995, 1989. DOI: 10.1109/29.32276.

[3] F. Oh, *What Is CUDA?* Accessed: 12/04-2024 [Online], Available: https://blogs.nvidia.com/blog/what-is-cuda-2/, 2012.

[4] J. Petterson and I. Wainwright, *Radar Signal Processing with Graphics Processors (GPUs)*, Master's thesis, Available: https://www.diva-portal.org/smash/get/diva2:292558/FULLTEXT01.pdf, Uppsala, Jan. 2010.

[5] S. Friberg and P. Pålsson, *Space-Time Adaptive Processing in FPGA*, Master's thesis, Available: https://publications.lib.chalmers.se/records/fulltext/173689/173689.pdf, Gothenburg, Nov. 2012.

[6] E. Buber and B. Diri, "Performance Analysis and CPU vs GPU Comparison for Deep Learning," in *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, 2018, pp. 1–6. DOI: 10.1109/CEIT.2018.8751930.

[7] A. Paulraj, R. Roy, and T. Kailath, "Estimation of Signal Parameters Via Rotational Invariance Techniques- ESPRIT," in *Nineteeth Asilomar Conference on Circuits, Systems and Computers, 1985.*, 1985, pp. 83–89. DOI: 10.1109/ACSSC.1985.671426.

[8] D. Wen, H. Yi, W. Zhang, and H. Xu, "2D-Unitary ESPRIT Based Multi-Target Joint Range and Velocity Estimation Algorithm for FMCW Radar," *Applied Sciences*, vol. 13, p. 10 448, Sep. 2023. DOI: 10.3390/app131810448.

[9] Y. Wu, C. Li, Y. T. Hou, and W. Lou, "Real-time DoA Estimation for Automotive Radar," in *2021 18th European Radar Conference (EuRAD)*, 2022, pp. 437–440. DOI: 10.23919/EuRAD50154.2022.9784501.

[10] R. Mark A., *Fundamentals of Radar Signal Processing. 3rd ed.* McGraw Hill, 2022.

[11] S. George W., *Introduction to Airborne Radar, 2nd ed.* SciTech Publishing, Inc., 1998.

[12] Intel, "What is a GPU?" Accessed: 23/01-2024 [Online], Available: https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html.

[13] *NVIDIA Turing GPU Architecture: Graphics Reinvented*, Accessed: 03/04-2024 [Online], Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/

design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, NVIDIA, 2018.

[14] NVIDIA, "CUDA Zone," Accessed: 12/04-2024 [Online], Available: https://developer.nvidia.com/cuda-zone.

[15] G. H. Golub and C. F. V. Loan, *Matrix Computations*, Third Edition. The Johns Hopkins University Press, 1996.

[16] NVIDIA, *NVIDIA NSight Systems User Guide*, https://docs.nvidia.com/nsight-systems/UserGuide/index.html, [Online; accessed 13-October-2024], 2024.

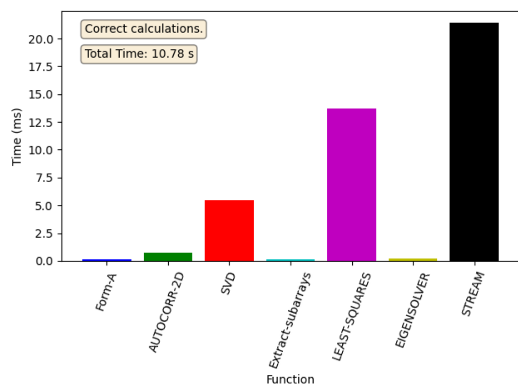
A

Appendix

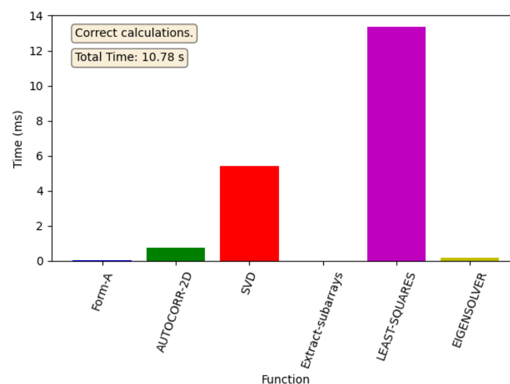
The appendix contains the generated graphs from all other runs in this work that were not included in the results section. The plots were generated from different threads and CUDA stream ratios for versions two and three.

A.1 Version 2, 1 threads, 500 streams

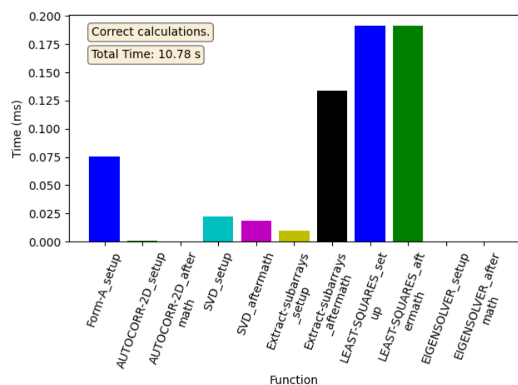
A.1.1 FP64, size 64



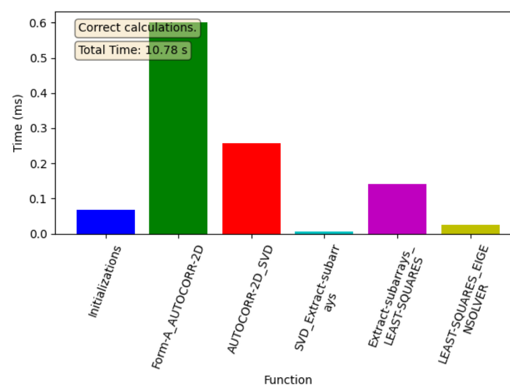
(a) Function mean time



(b) Compute mean time

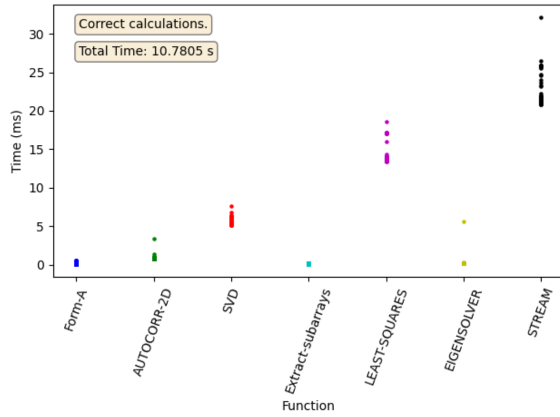


(c) Memory mean time

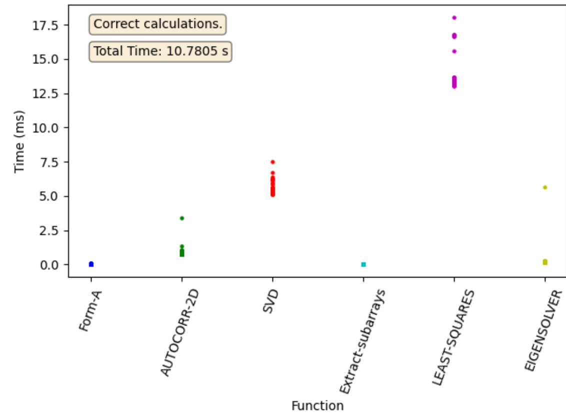


(d) Inter-function mean time

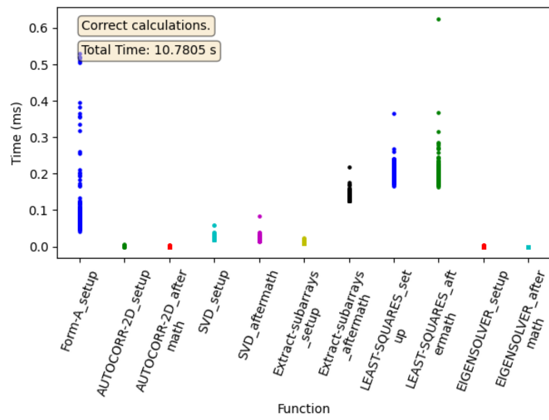
Figure A.1: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is one thread and 500 CUDA streams.



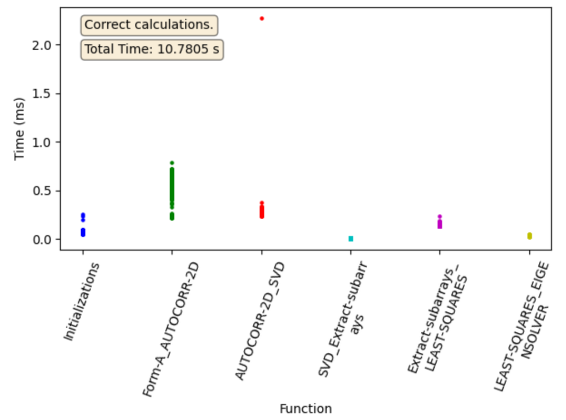
(a) Function mean time



(b) Compute mean time



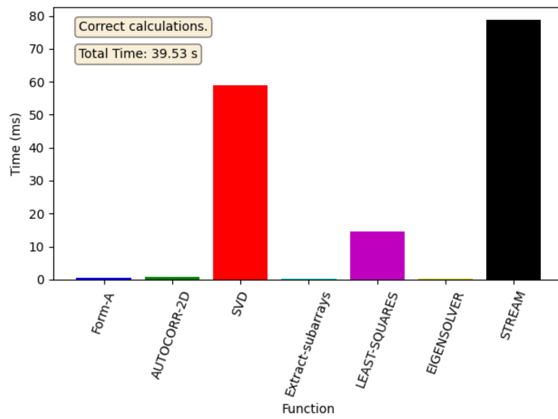
(c) Memory mean time



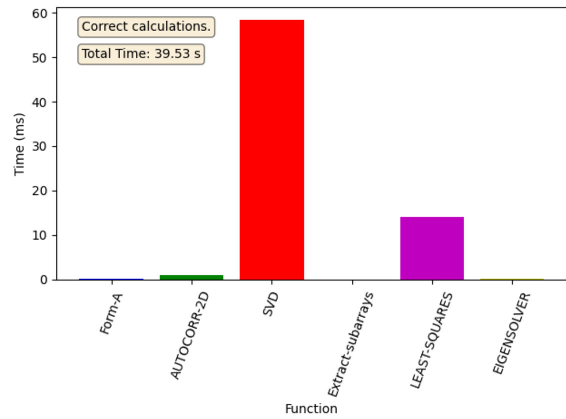
(d) Inter-function mean time

Figure A.2: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is one thread and 500 CUDA streams.

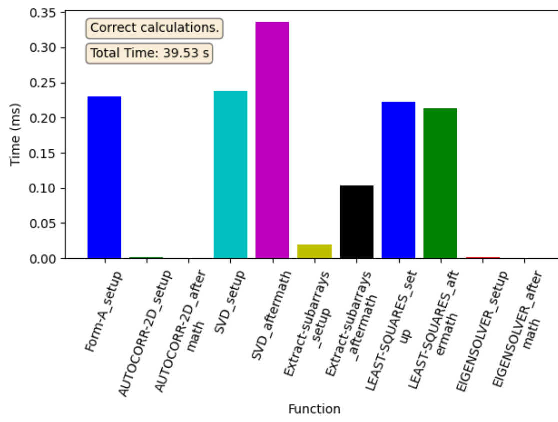
A.1.2 FP64, size 360



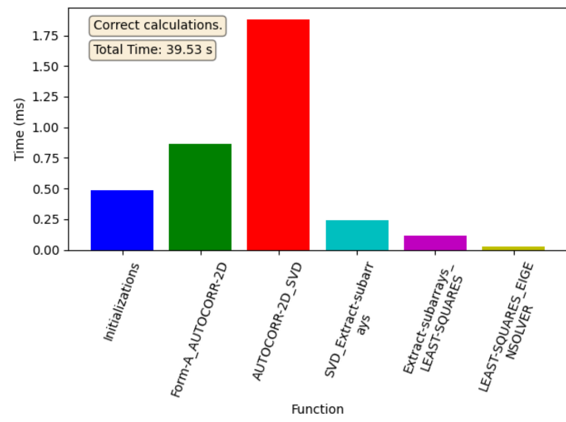
(a) Function mean time



(b) Compute mean time

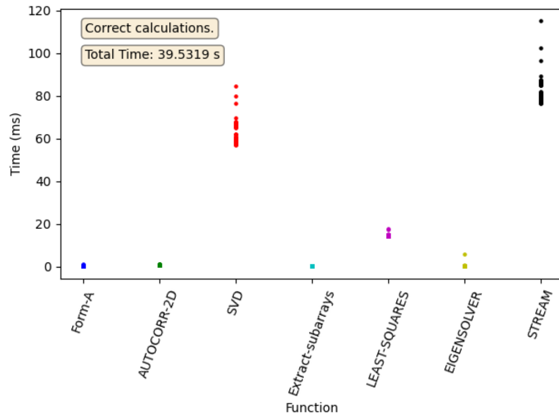


(c) Memory mean time

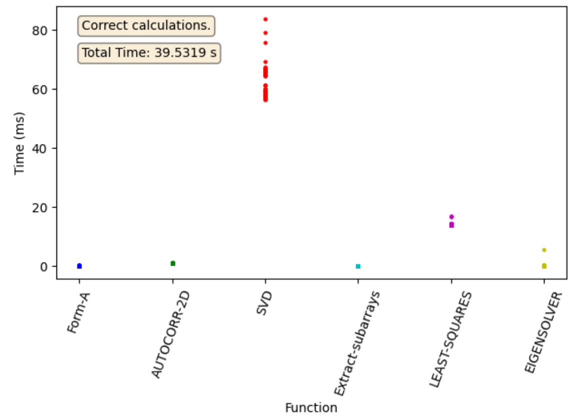


(d) Inter-function mean time

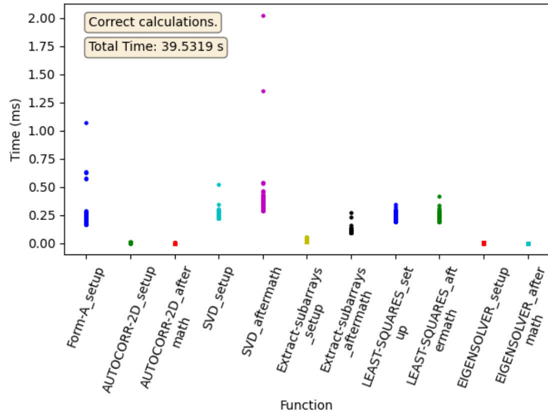
Figure A.3: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is one thread and 500 CUDA streams.



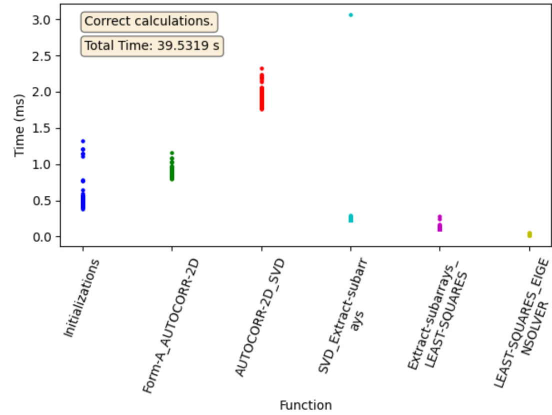
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

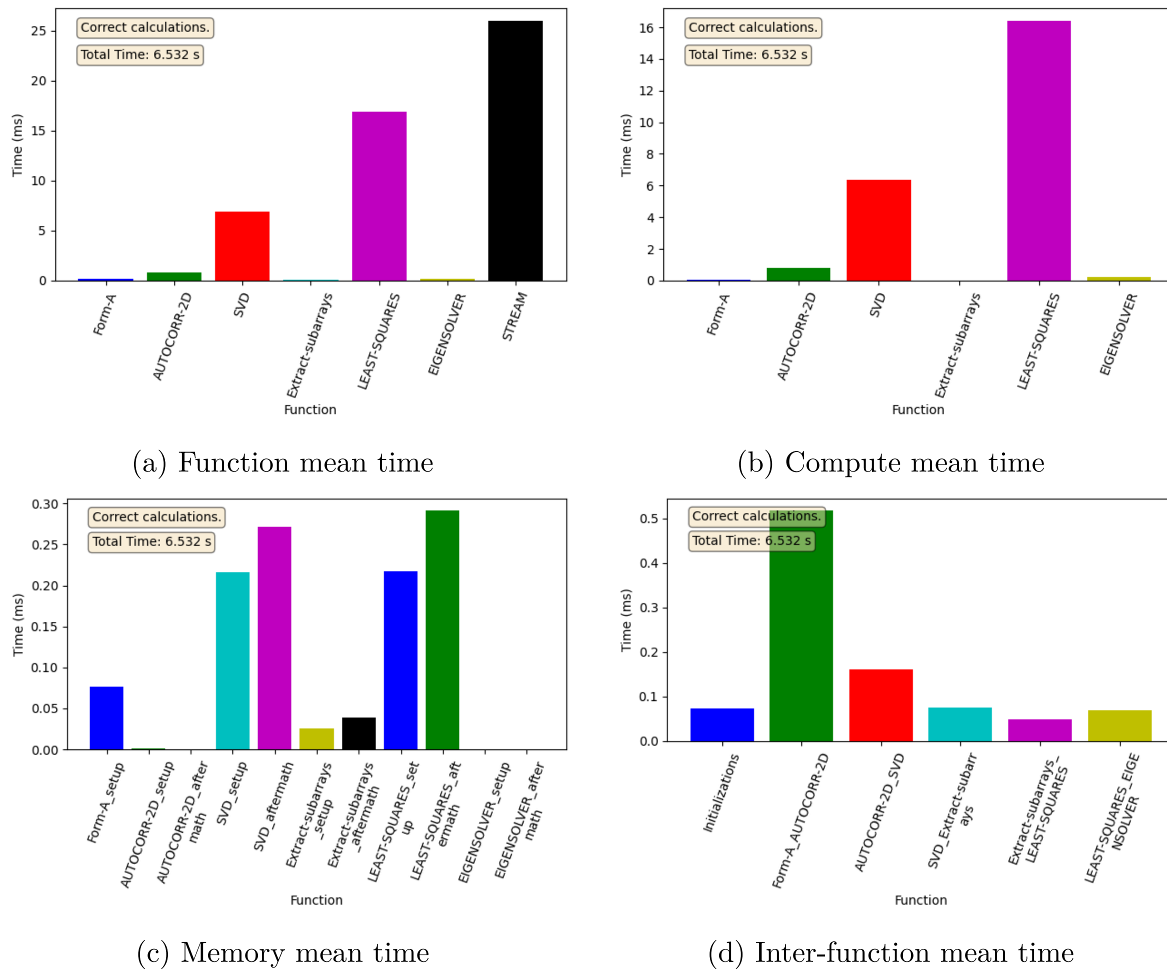


(d) Inter-function mean time

Figure A.4: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is one thread and 500 CUDA streams.

A.2 Version 2, 2 threads, 250 streams

A.2.1 FP64, size 64



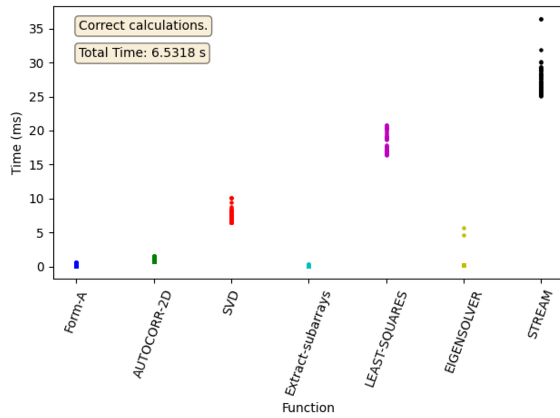
(a) Function mean time

(b) Compute mean time

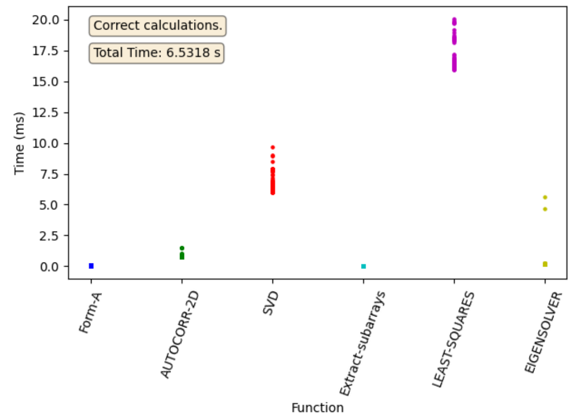
(c) Memory mean time

(d) Inter-function mean time

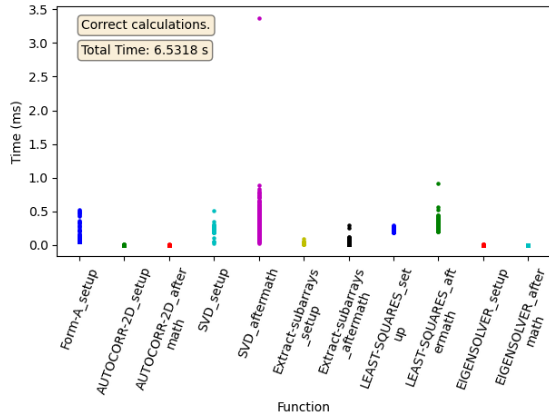
Figure A.5: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is two threads with 250 CUDA streams per thread.



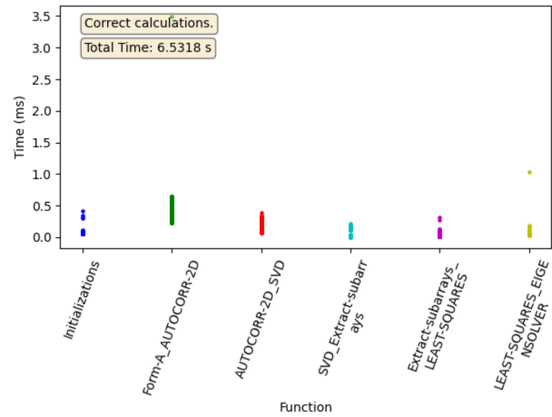
(a) Function mean time



(b) Compute mean time



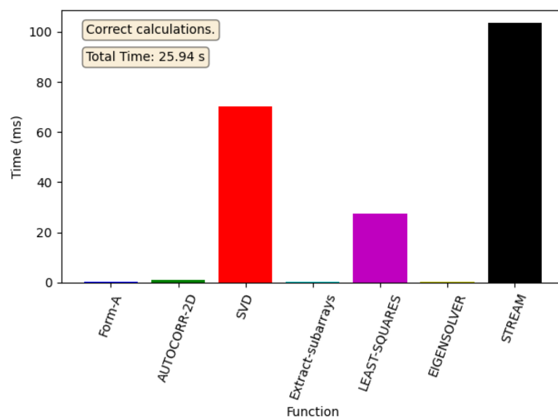
(c) Memory mean time



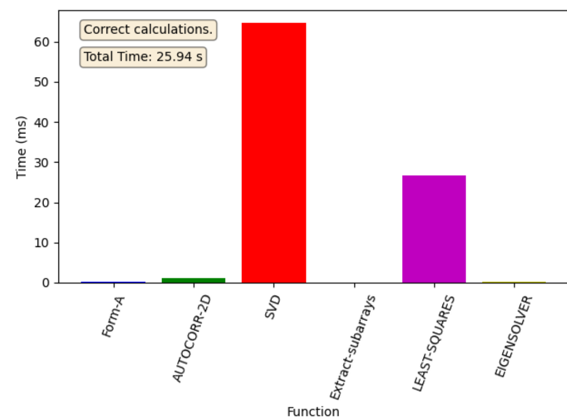
(d) Inter-function mean time

Figure A.6: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is two threads with 250 CUDA streams per thread.

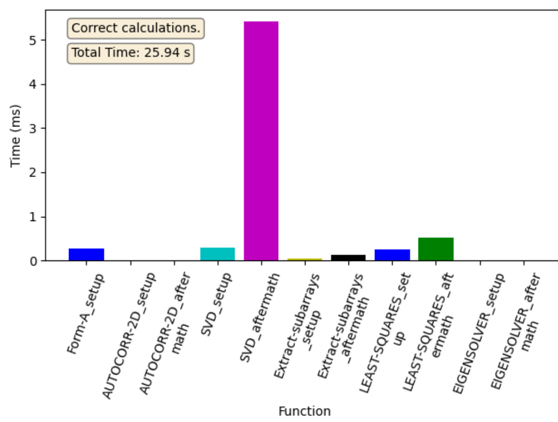
A.2.2 FP64, size 360



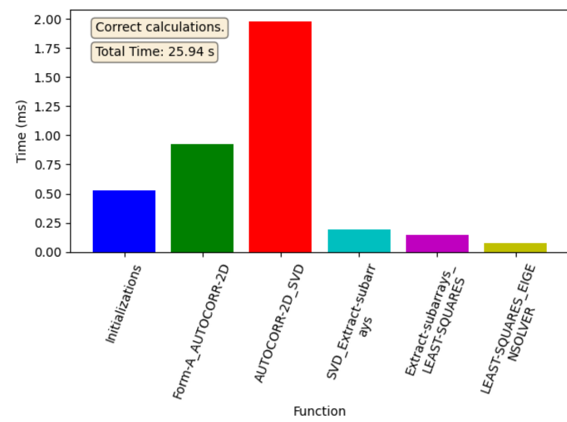
(a) Function mean time



(b) Compute mean time

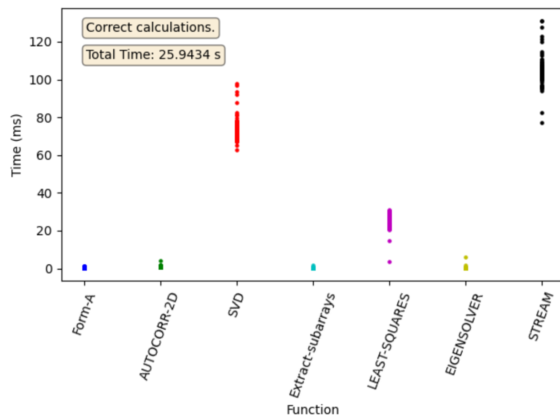


(c) Memory mean time

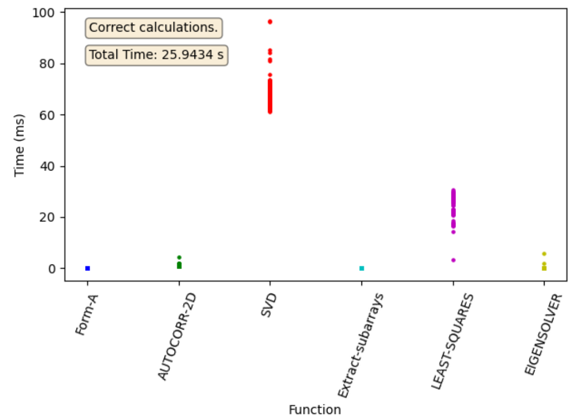


(d) Inter-function mean time

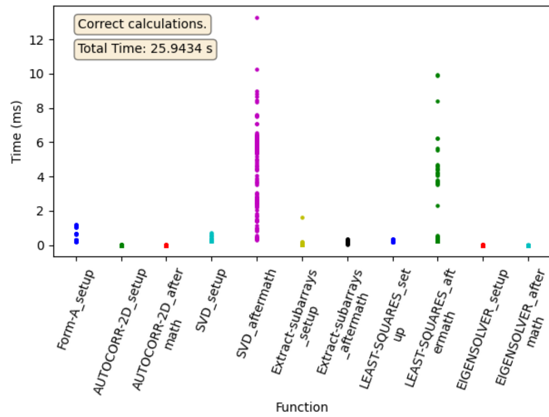
Figure A.7: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is two threads with 250 CUDA streams per thread.



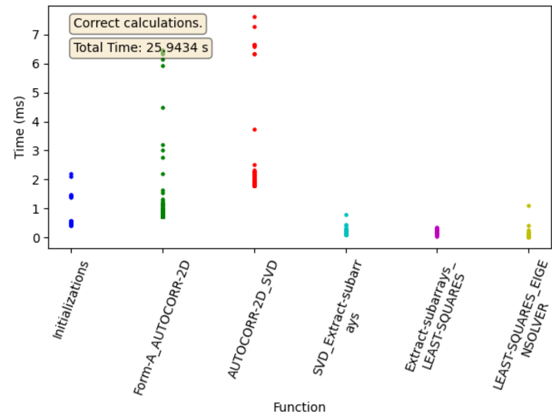
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

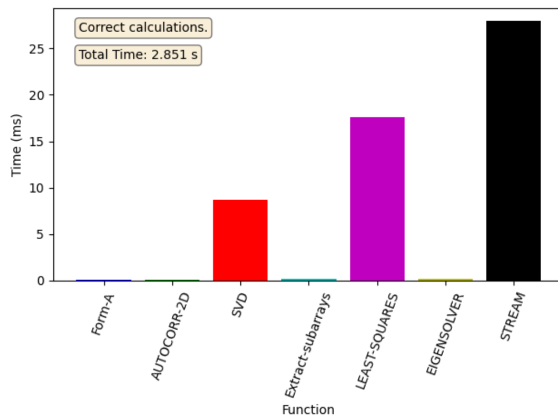


(d) Inter-function mean time

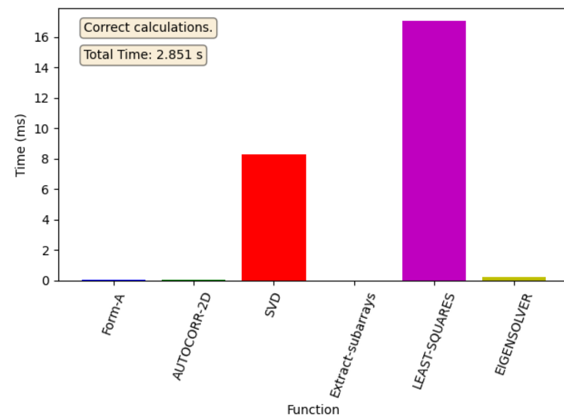
Figure A.8: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is two threads with 250 CUDA streams per thread.

A.3 Version 2, 5 threads, 100 streams

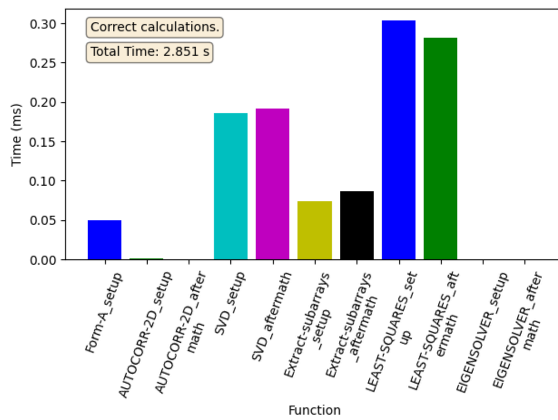
A.3.1 FP32, size 64



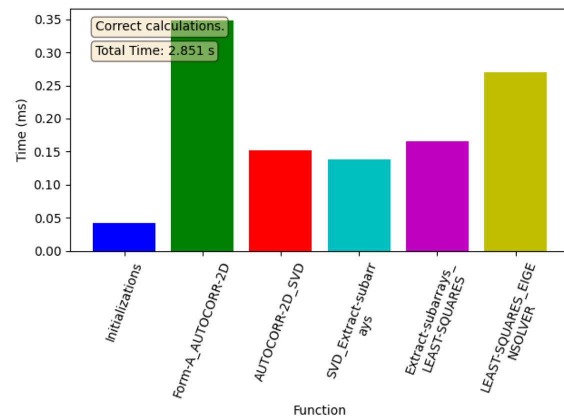
(a) Function mean time



(b) Compute mean time

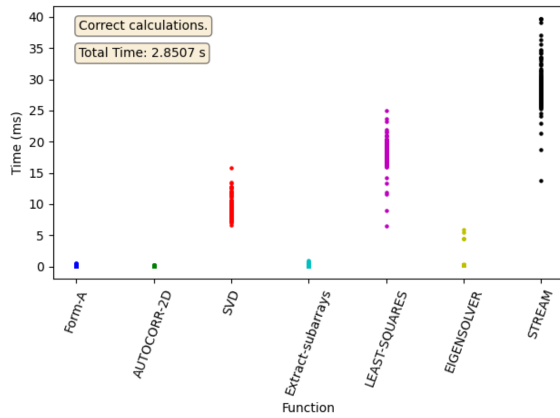


(c) Memory mean time

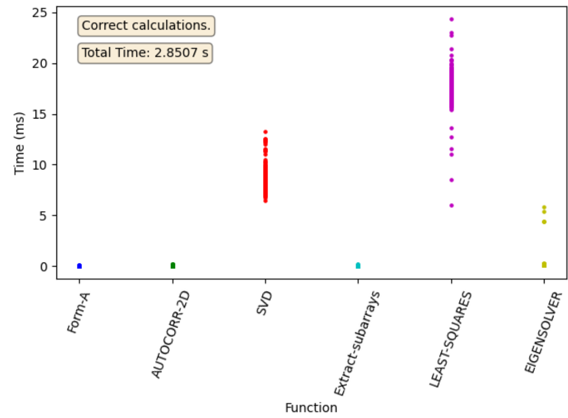


(d) Inter-function mean time

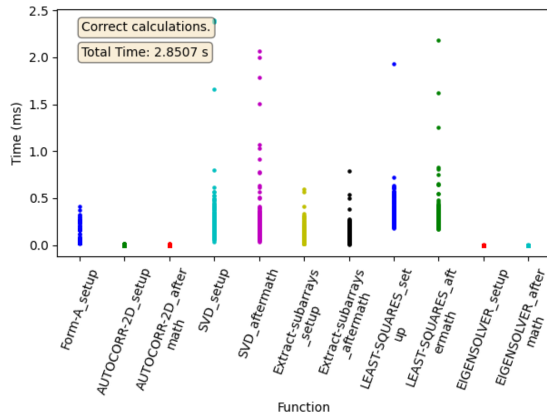
Figure A.9: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 5 threads with 100 CUDA streams per thread.



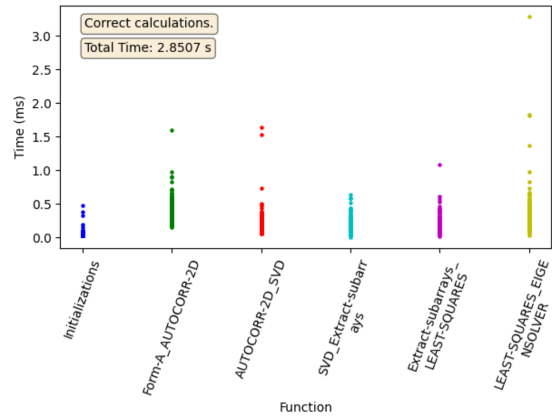
(a) Function mean time



(b) Compute mean time



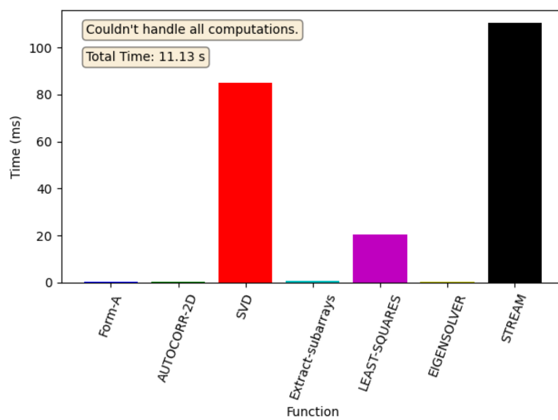
(c) Memory mean time



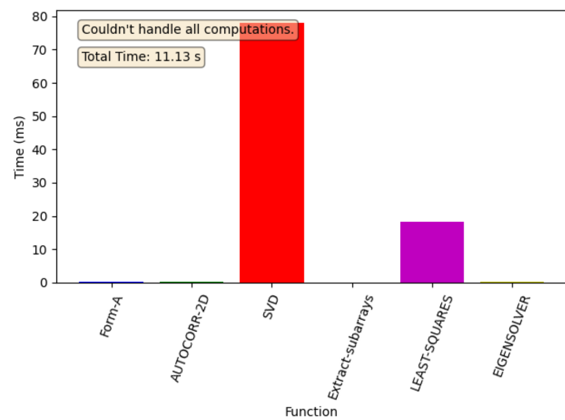
(d) Inter-function mean time

Figure A.10: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 5 threads with 100 CUDA streams per thread.

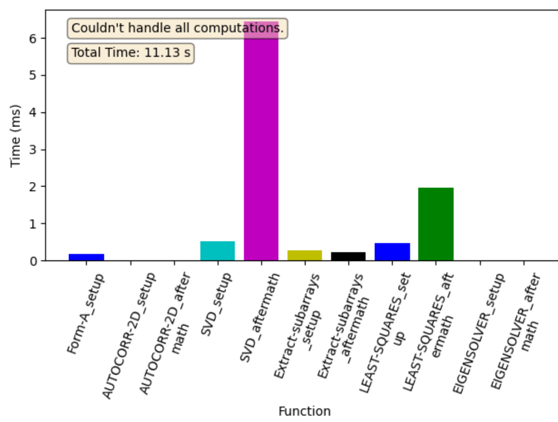
A.3.2 FP32, size 360



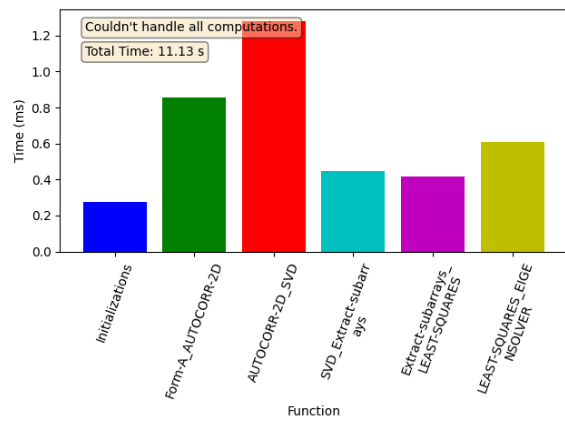
(a) Function mean time



(b) Compute mean time

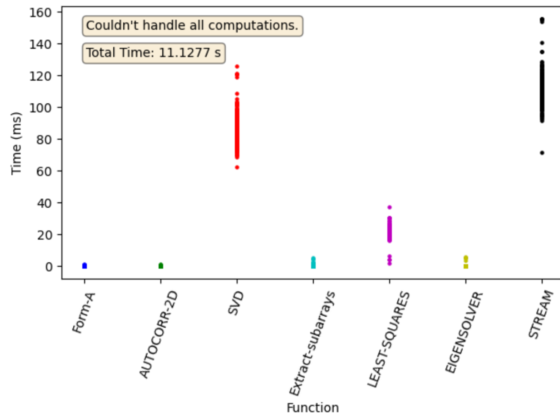


(c) Memory mean time

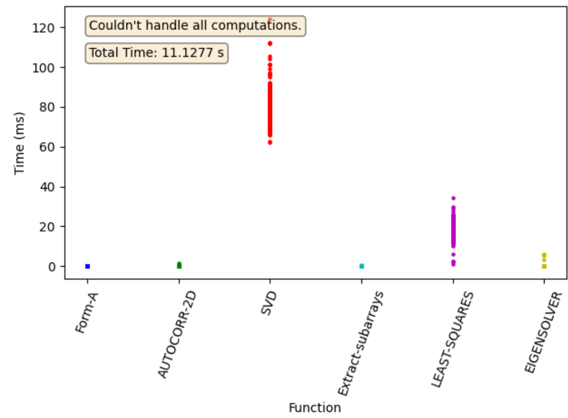


(d) Inter-function mean time

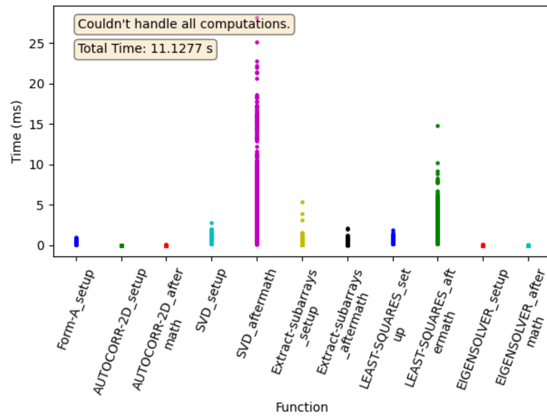
Figure A.11: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 5 threads with 100 CUDA streams per thread.



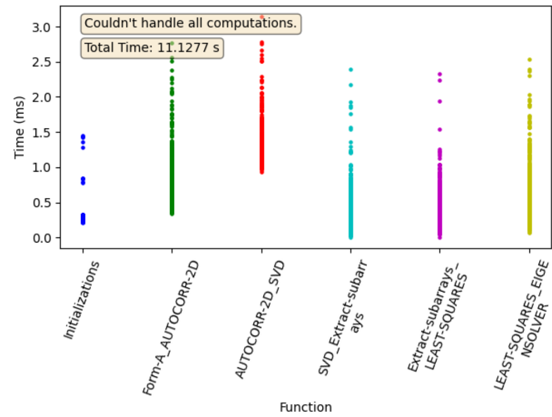
(a) Function mean time



(b) Compute mean time



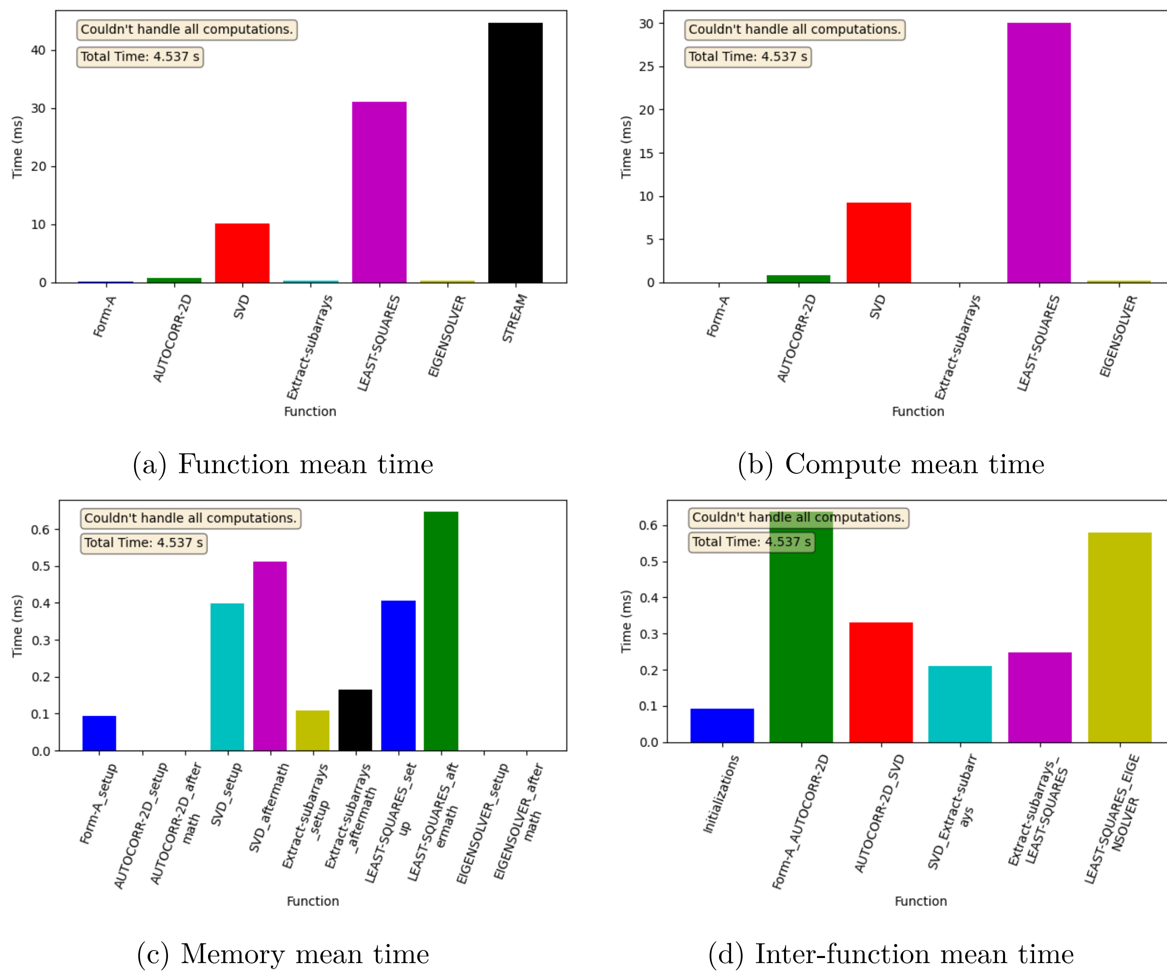
(c) Memory mean time



(d) Inter-function mean time

Figure A.12: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 5 threads with 100 CUDA streams per thread.

A.3.3 FP64, size 64



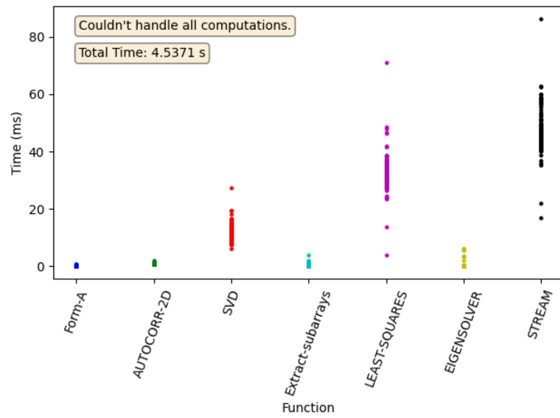
(a) Function mean time

(b) Compute mean time

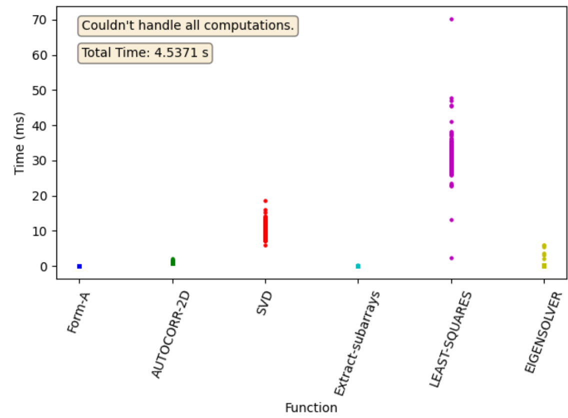
(c) Memory mean time

(d) Inter-function mean time

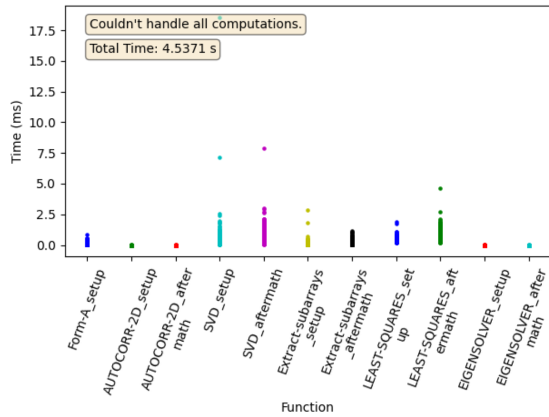
Figure A.13: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 5 threads with 100 CUDA streams per thread.



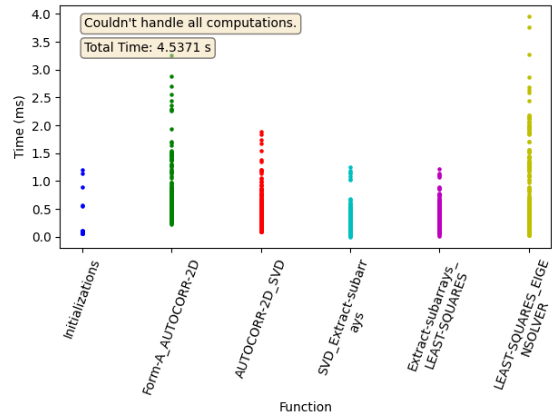
(a) Function mean time



(b) Compute mean time



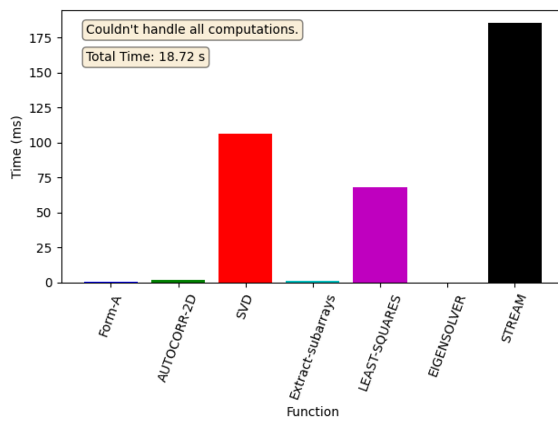
(c) Memory mean time



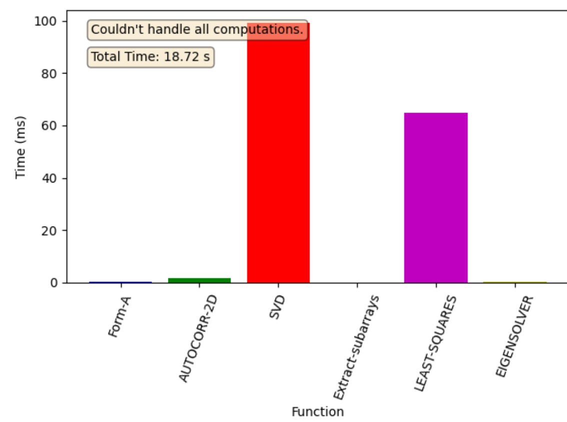
(d) Inter-function mean time

Figure A.14: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 5 threads with 100 CUDA streams per thread.

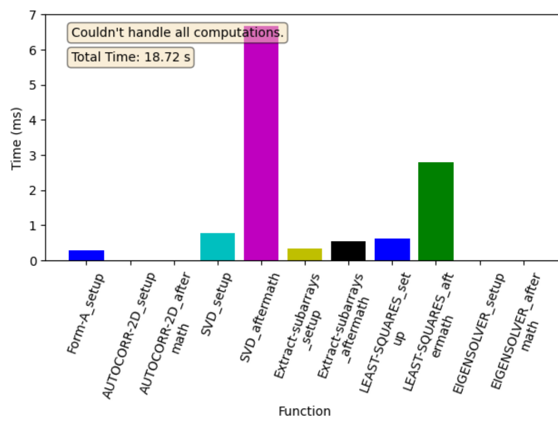
A.3.4 FP64, size 360



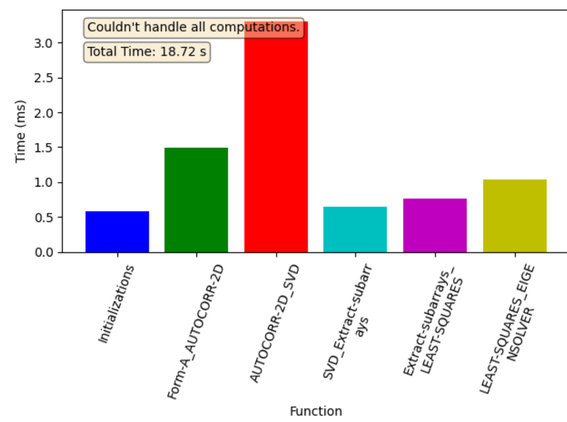
(a) Function mean time



(b) Compute mean time

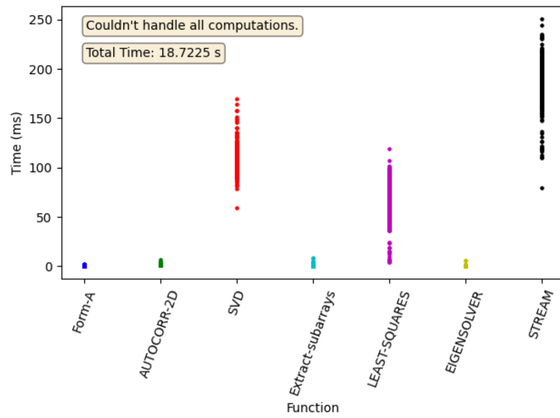


(c) Memory mean time

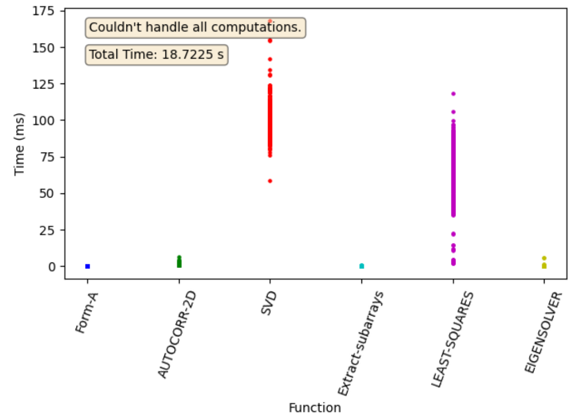


(d) Inter-function mean time

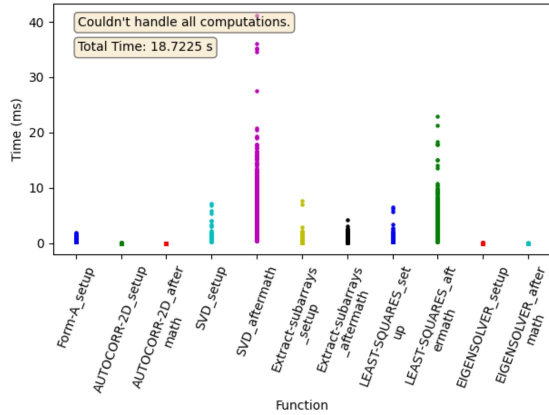
Figure A.15: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 5 threads with 100 CUDA streams per thread.



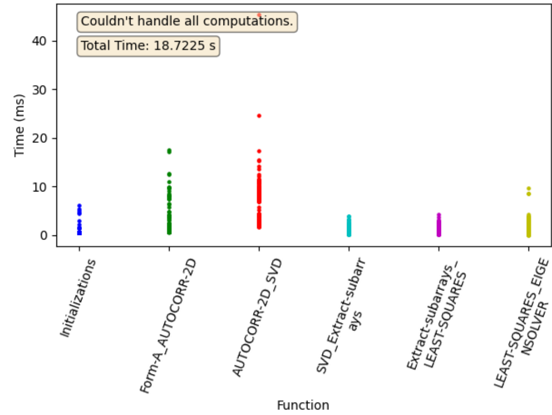
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

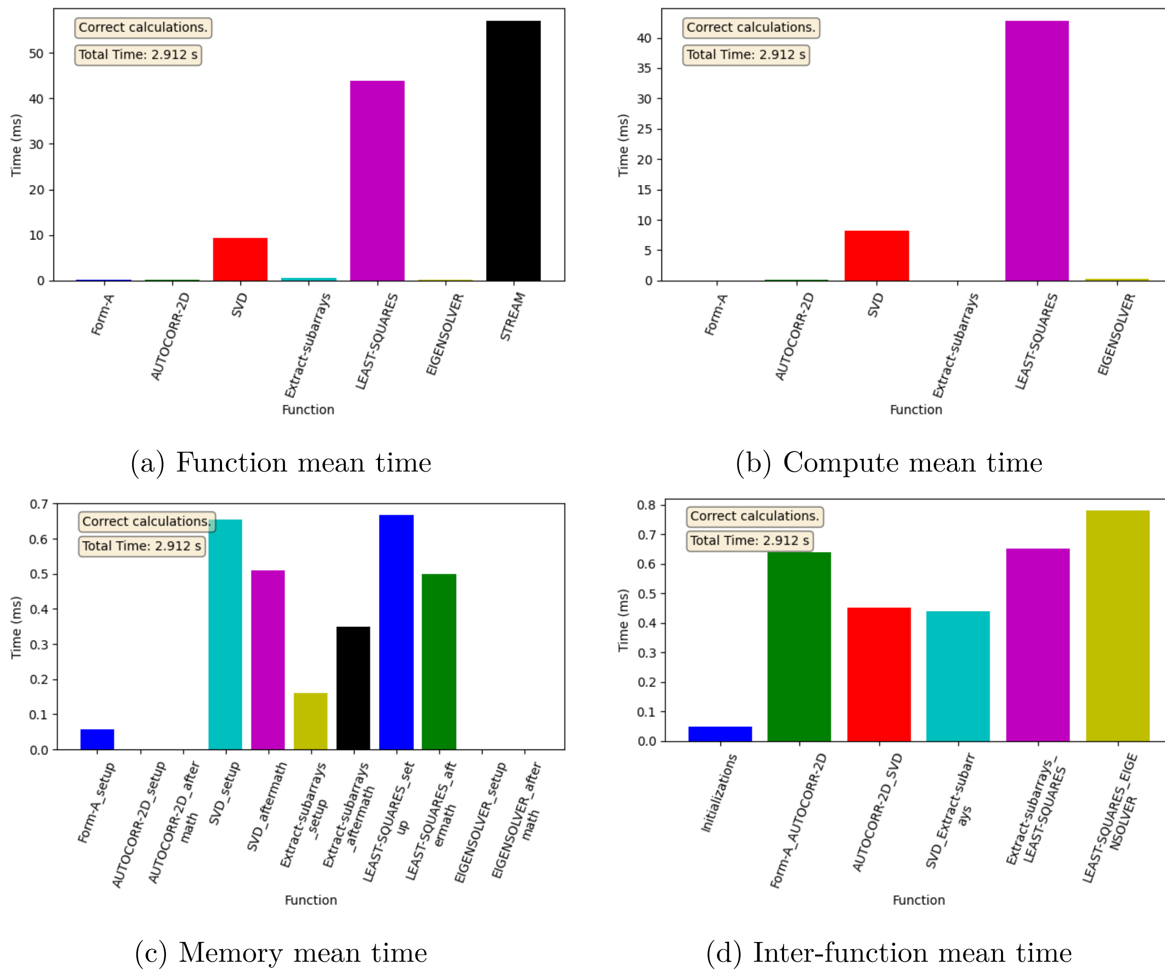


(d) Inter-function mean time

Figure A.16: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 5 threads with 100 CUDA streams per thread.

A.4 Version 2, 10 threads, 50 streams

A.4.1 FP32, size 64



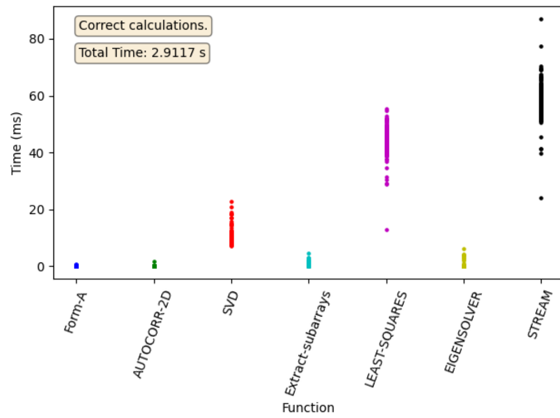
(a) Function mean time

(b) Compute mean time

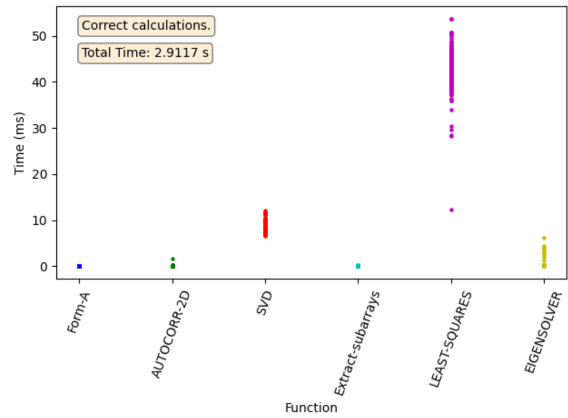
(c) Memory mean time

(d) Inter-function mean time

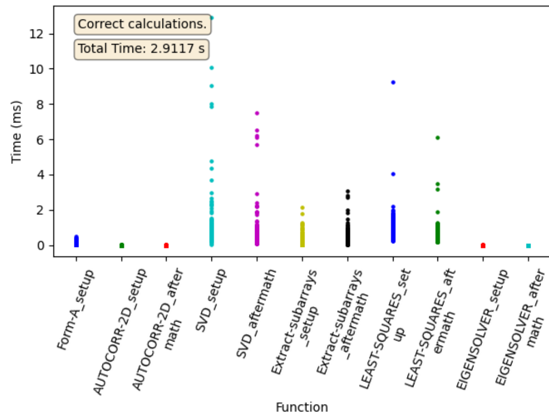
Figure A.17: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 10 threads with 50 CUDA streams per thread.



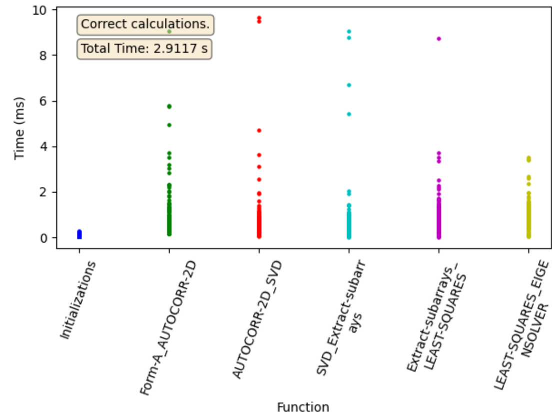
(a) Function mean time



(b) Compute mean time



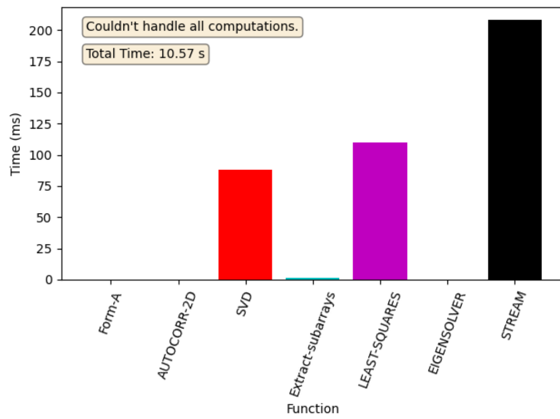
(c) Memory mean time



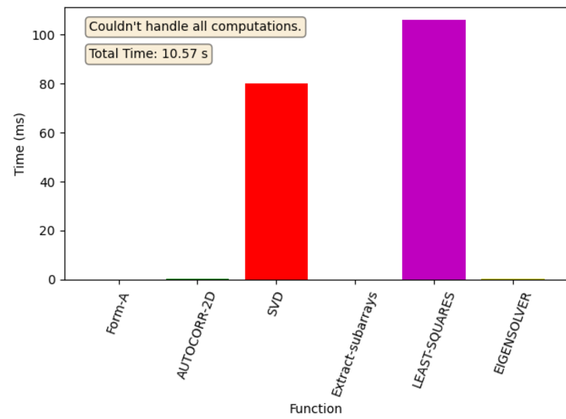
(d) Inter-function mean time

Figure A.18: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 10 threads with 50 CUDA streams per thread.

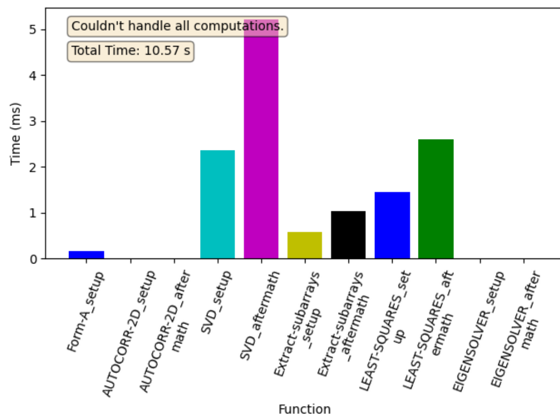
A.4.2 FP32, size 360



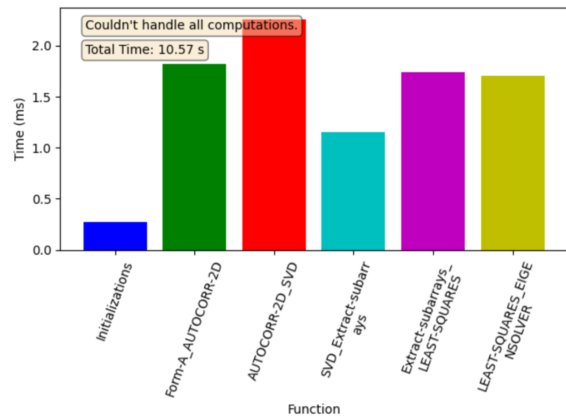
(a) Function mean time



(b) Compute mean time

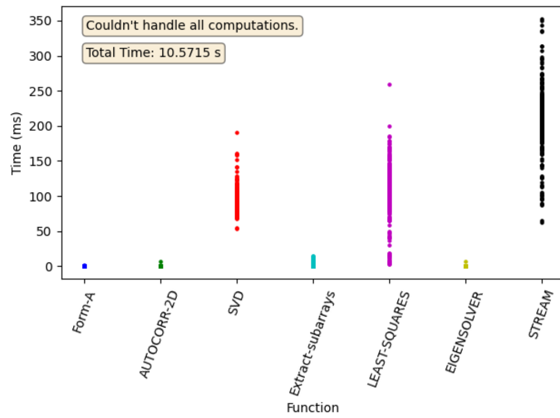


(c) Memory mean time

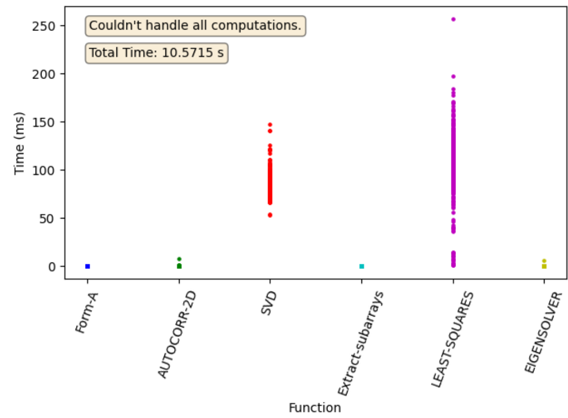


(d) Inter-function mean time

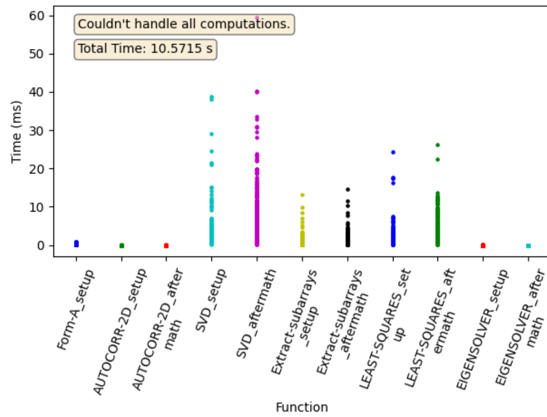
Figure A.19: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 10 threads with 50 CUDA streams per thread.



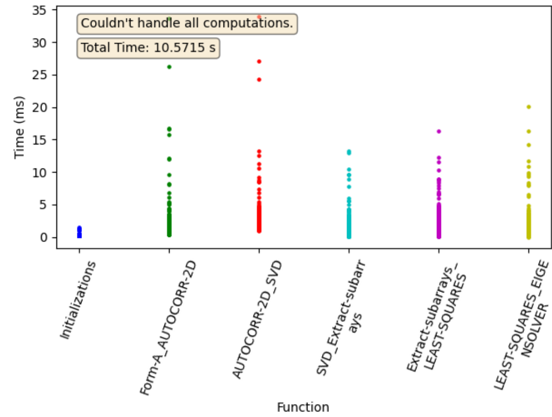
(a) Function mean time



(b) Compute mean time



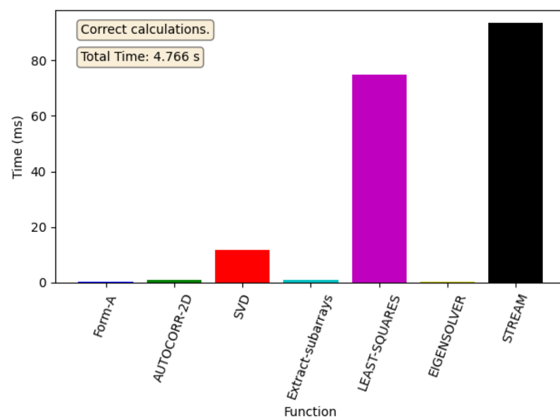
(c) Memory mean time



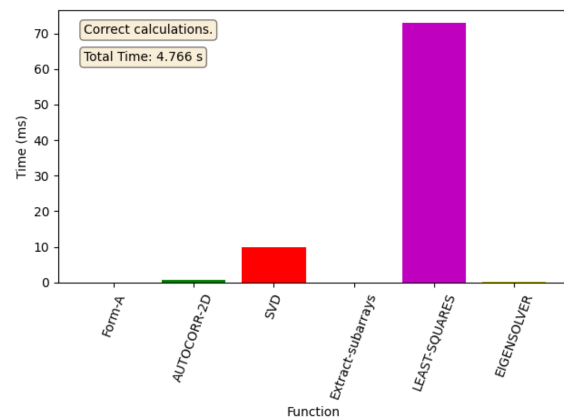
(d) Inter-function mean time

Figure A.20: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 10 threads with 50 CUDA streams per thread.

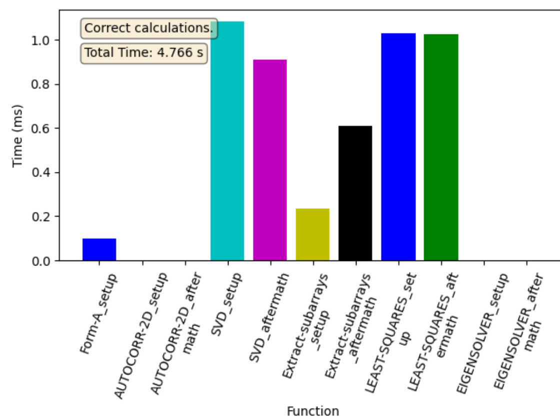
A.4.3 FP64, size 64



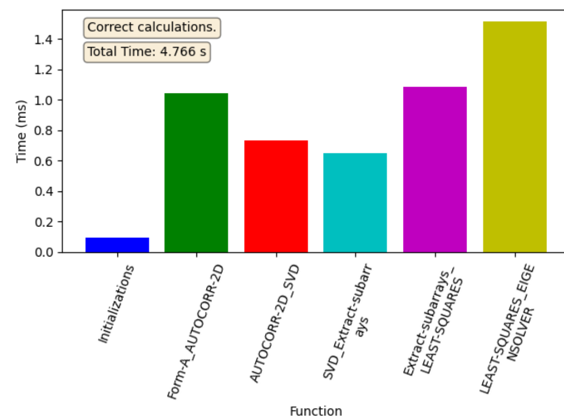
(a) Function mean time



(b) Compute mean time

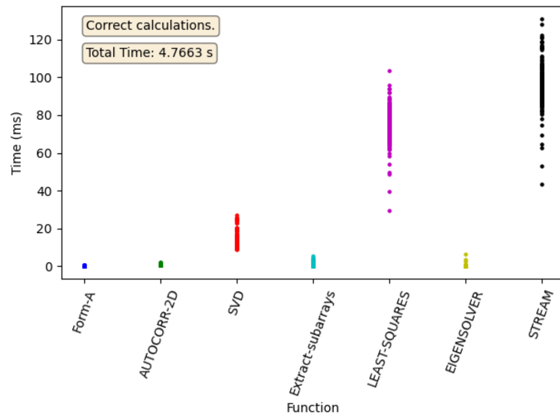


(c) Memory mean time

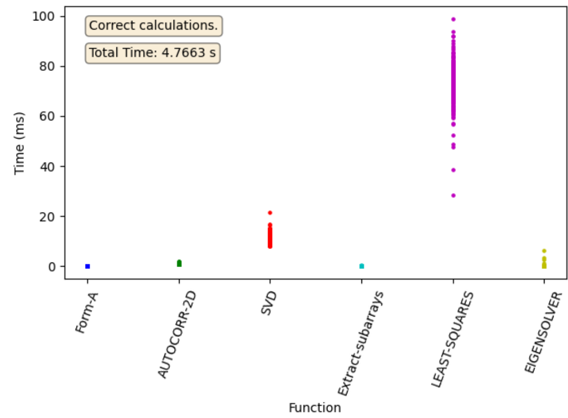


(d) Inter-function mean time

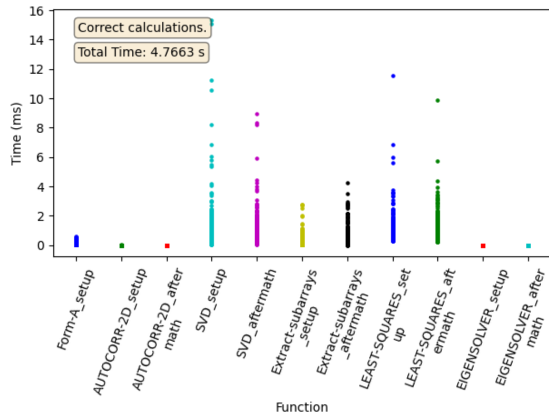
Figure A.21: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 10 threads with 50 CUDA streams per thread.



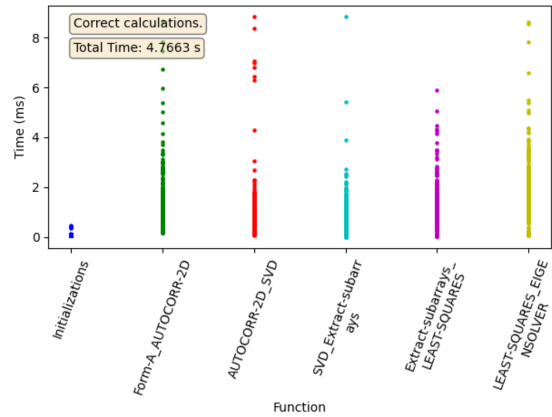
(a) Function mean time



(b) Compute mean time



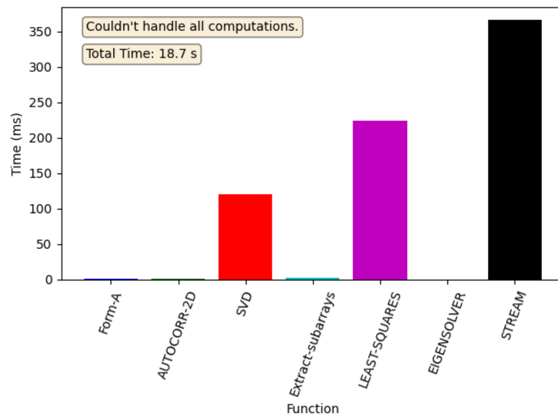
(c) Memory mean time



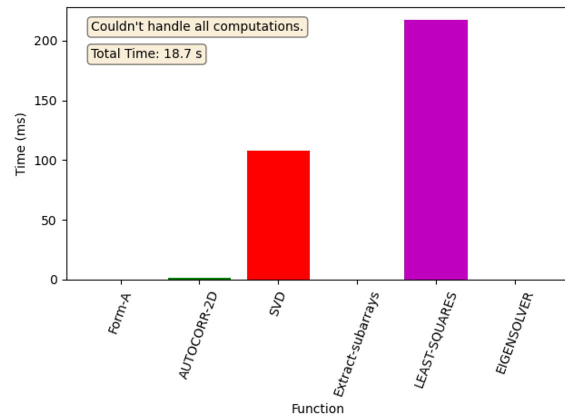
(d) Inter-function mean time

Figure A.22: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 10 threads with 50 CUDA streams per thread.

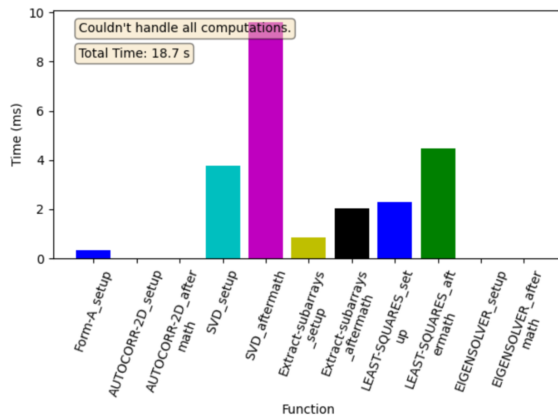
A.4.4 FP64, size 360



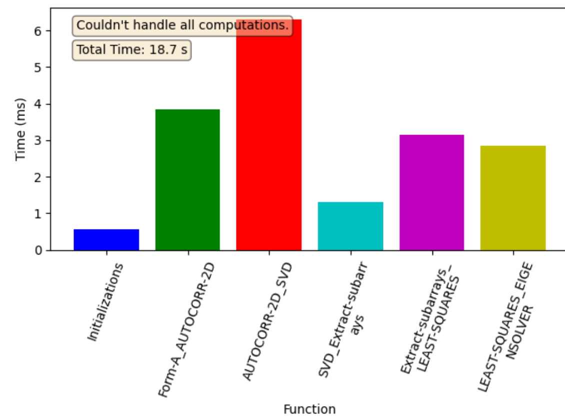
(a) Function mean time



(b) Compute mean time

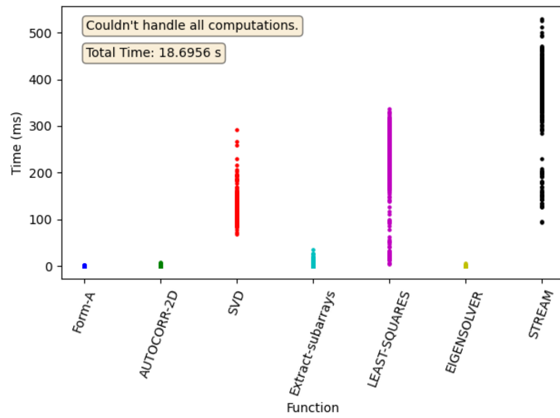


(c) Memory mean time

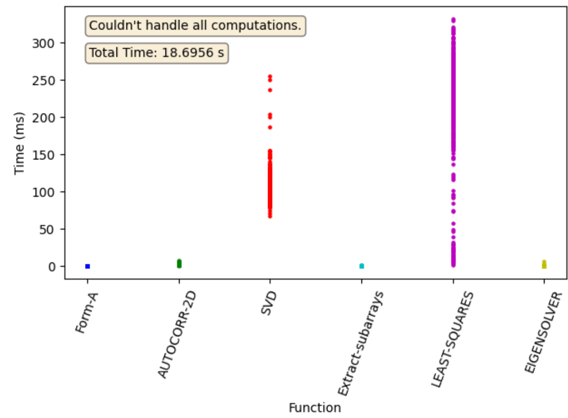


(d) Inter-function mean time

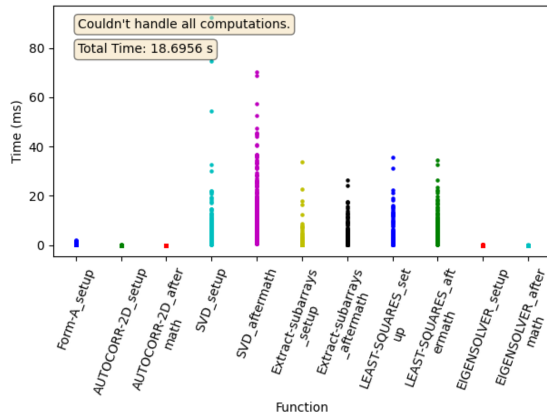
Figure A.23: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 10 threads with 50 CUDA streams per thread.



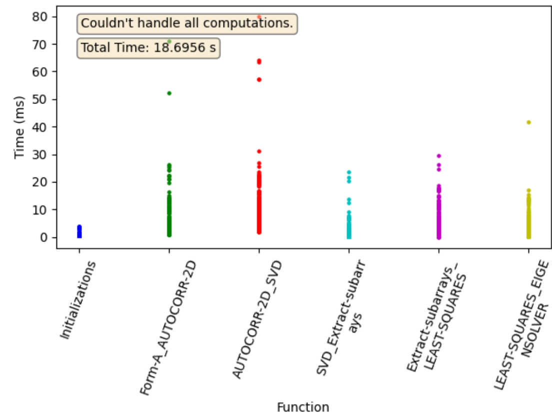
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

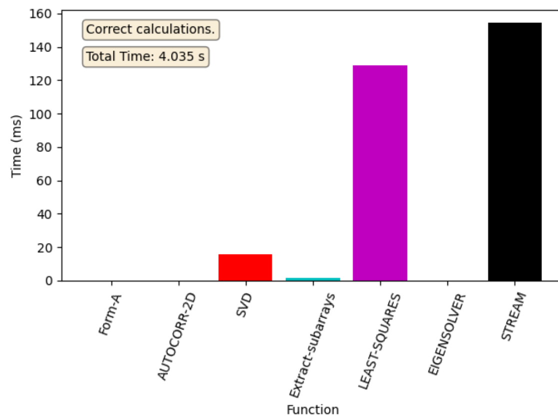


(d) Inter-function mean time

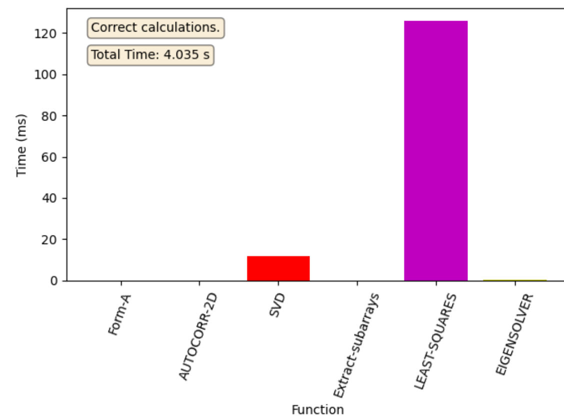
Figure A.24: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 10 threads with 50 CUDA streams per thread.

A.5 Version 2, 20 threads, 25 streams

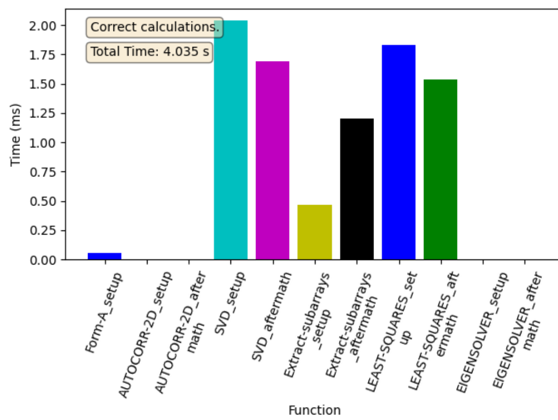
A.5.1 FP32, size 64



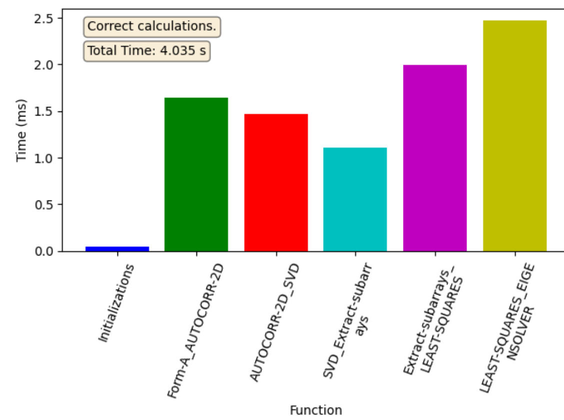
(a) Function mean time



(b) Compute mean time

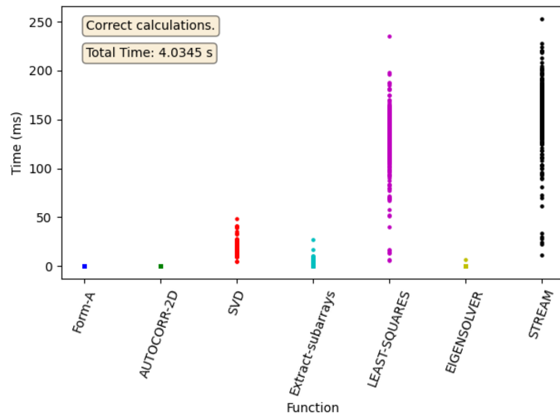


(c) Memory mean time

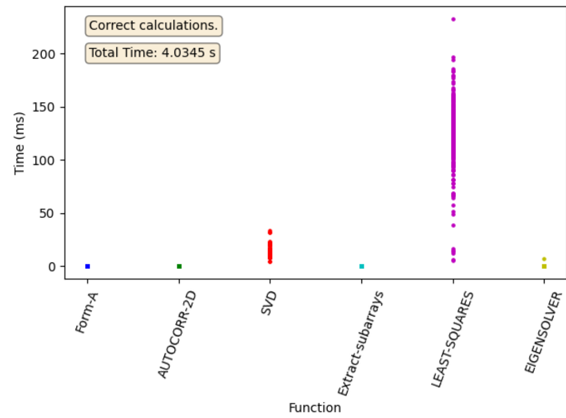


(d) Inter-function mean time

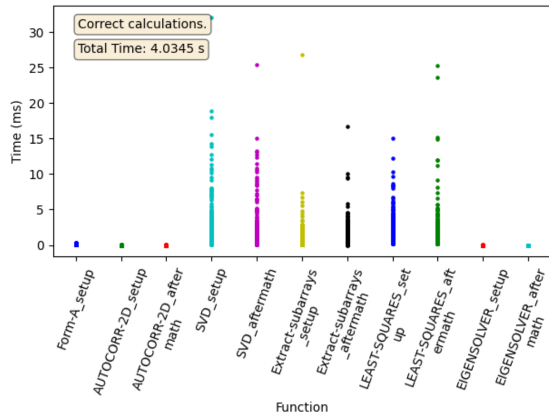
Figure A.25: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 20 threads with 25 CUDA streams per thread.



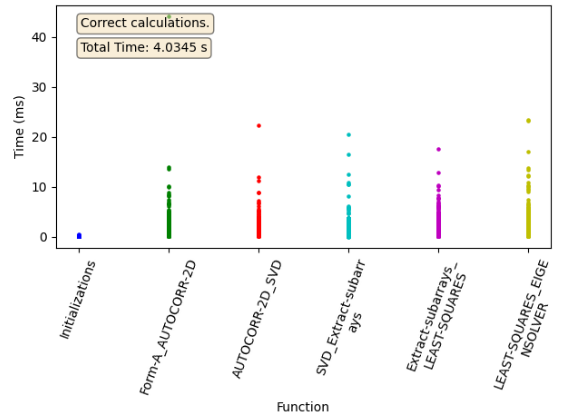
(a) Function mean time



(b) Compute mean time



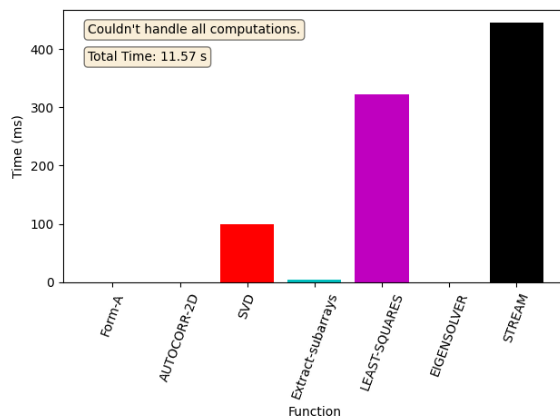
(c) Memory mean time



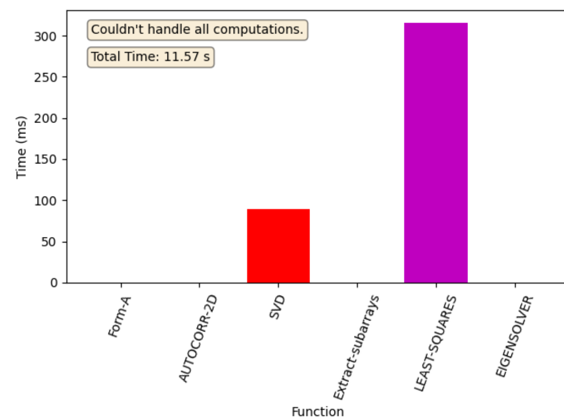
(d) Inter-function mean time

Figure A.26: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 20 threads with 25 CUDA streams per thread.

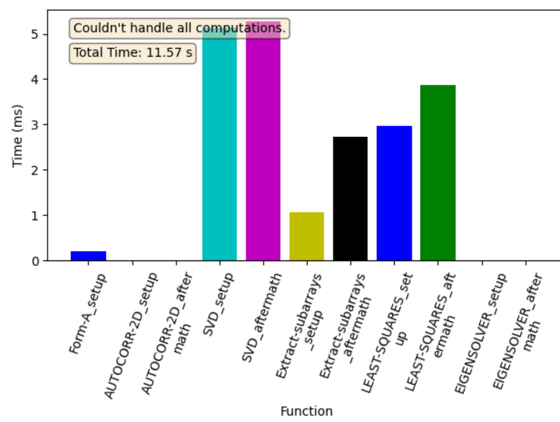
A.5.2 FP32, size 360



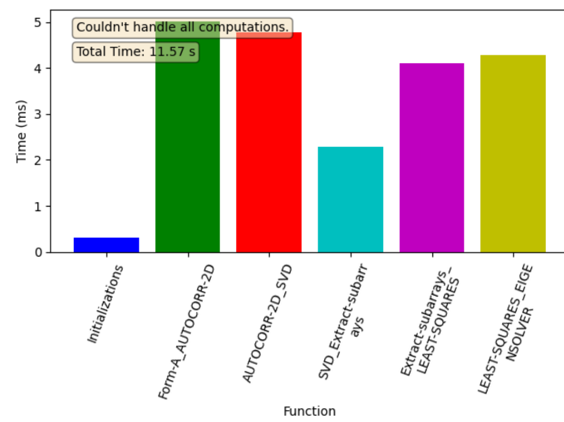
(a) Function mean time



(b) Compute mean time

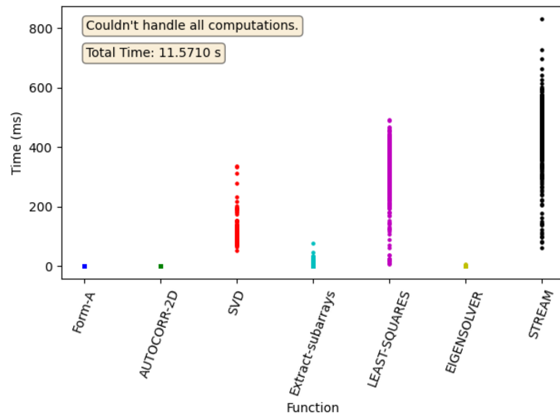


(c) Memory mean time

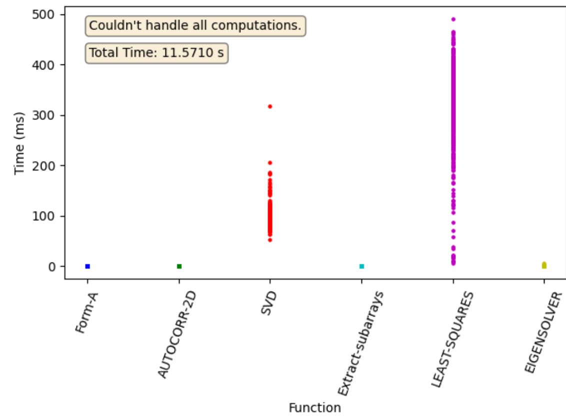


(d) Inter-function mean time

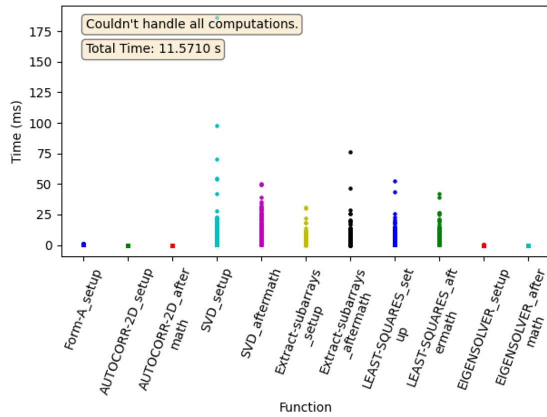
Figure A.27: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 20 threads with 25 CUDA streams per thread.



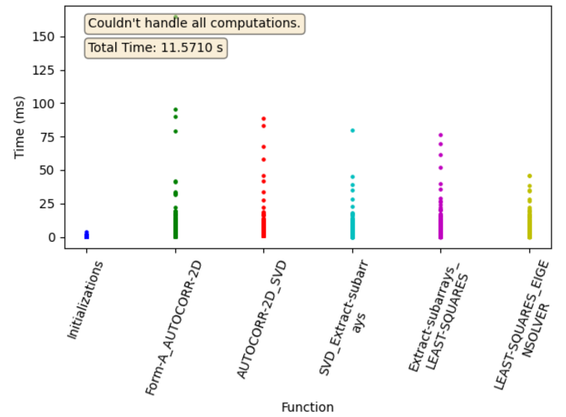
(a) Function mean time



(b) Compute mean time



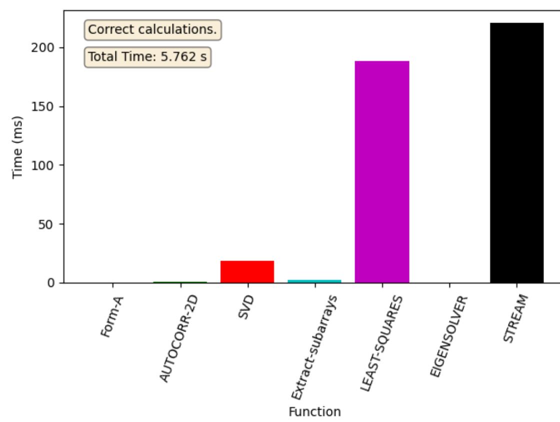
(c) Memory mean time



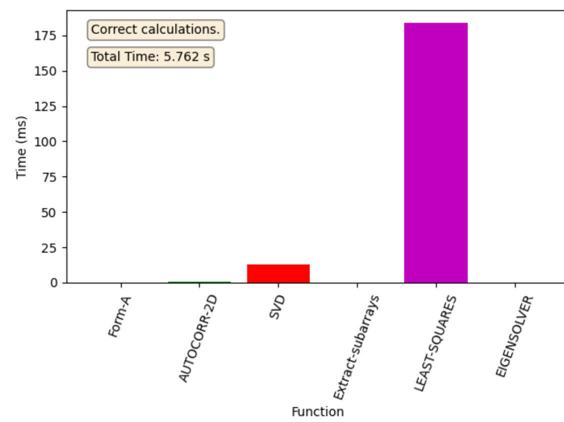
(d) Inter-function mean time

Figure A.28: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 20 threads with 25 CUDA streams per thread.

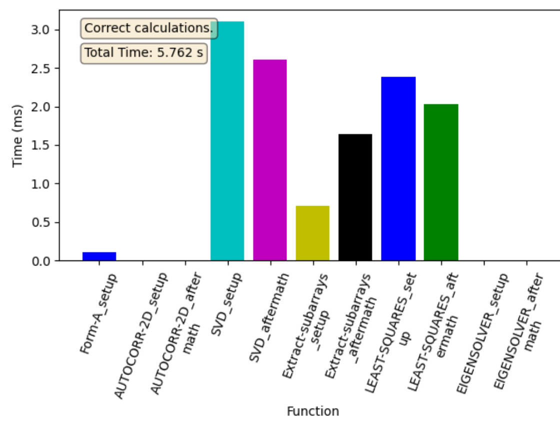
A.5.3 FP64, size 64



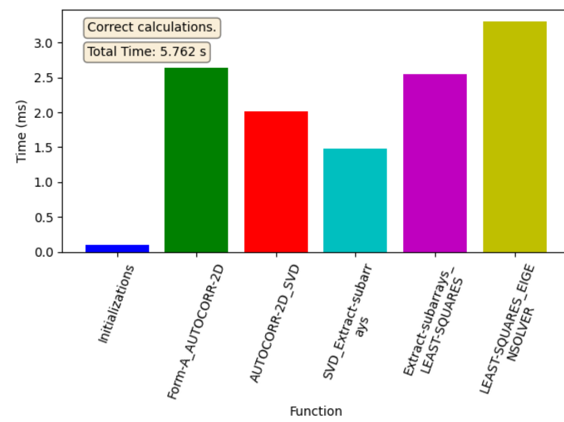
(a) Function mean time



(b) Compute mean time

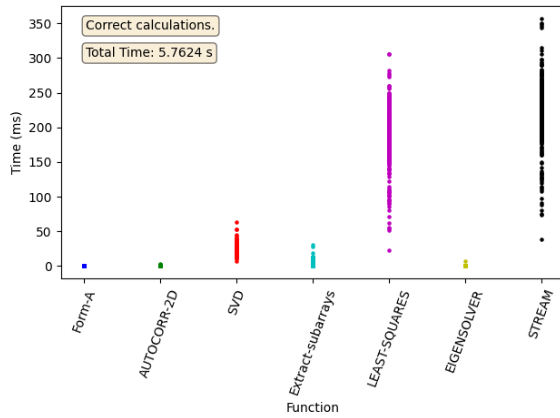


(c) Memory mean time

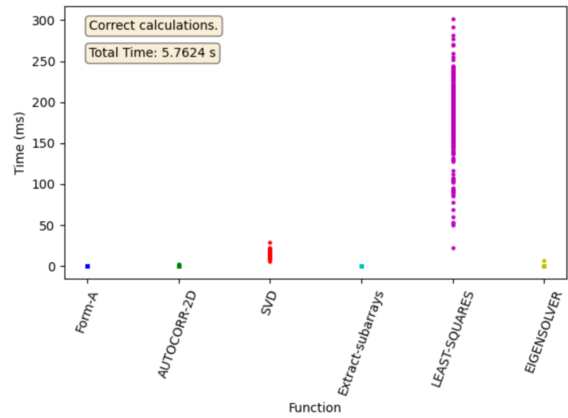


(d) Inter-function mean time

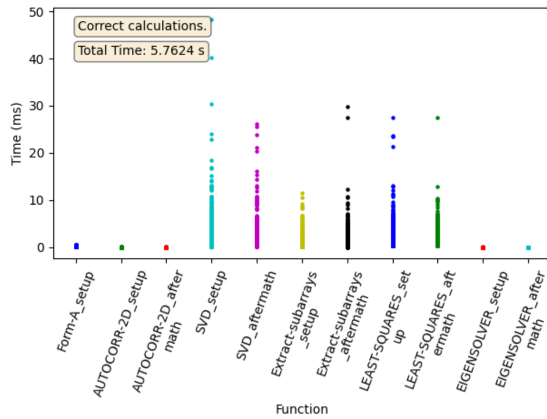
Figure A.29: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 20 threads with 25 CUDA streams per thread.



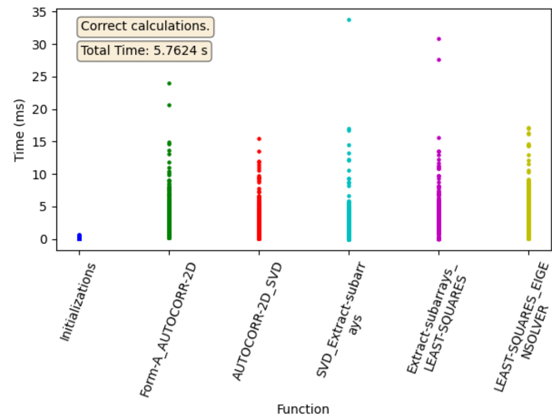
(a) Function mean time



(b) Compute mean time



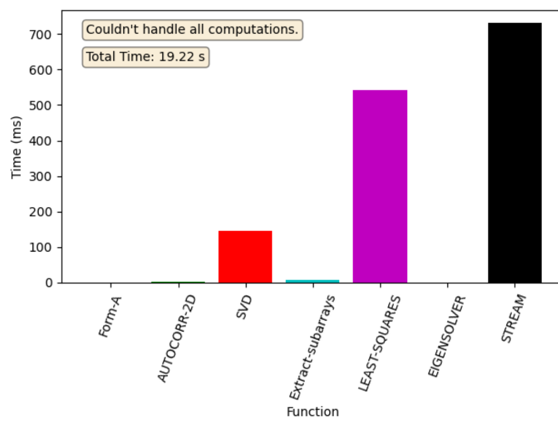
(c) Memory mean time



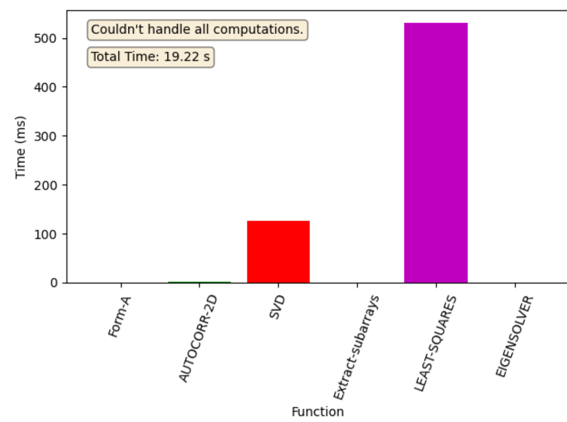
(d) Inter-function mean time

Figure A.30: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 20 threads with 25 CUDA streams per thread.

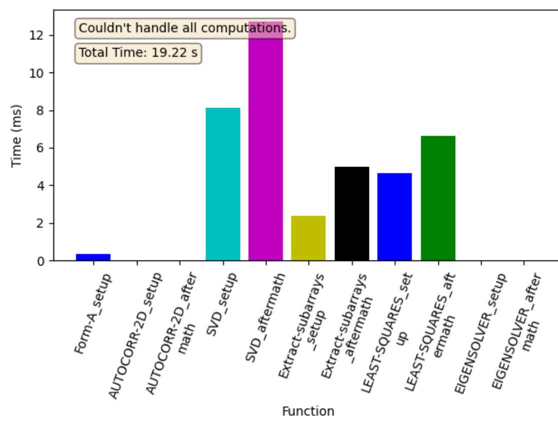
A.5.4 FP64, size 360



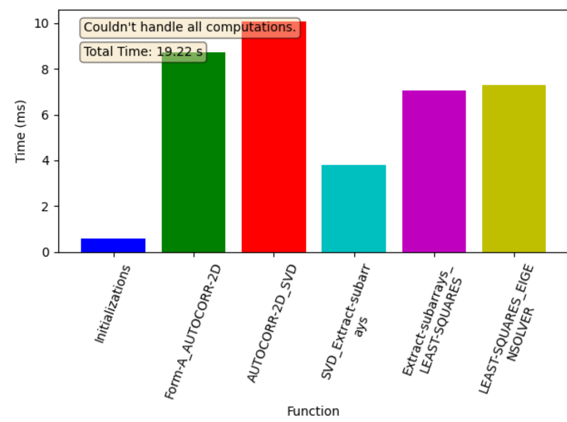
(a) Function mean time



(b) Compute mean time

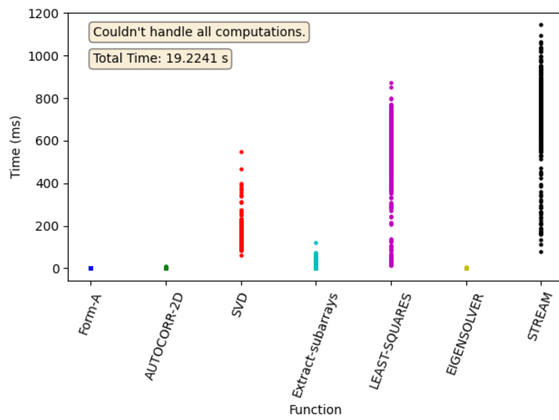


(c) Memory mean time

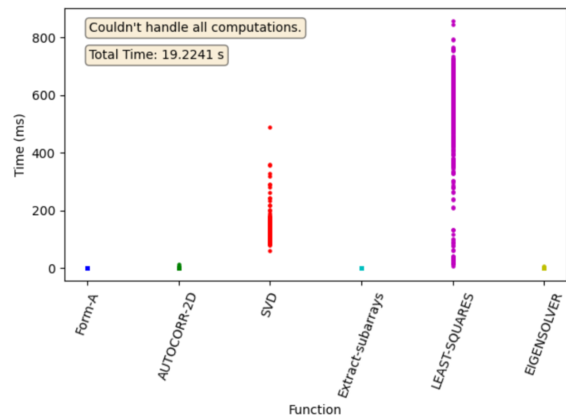


(d) Inter-function mean time

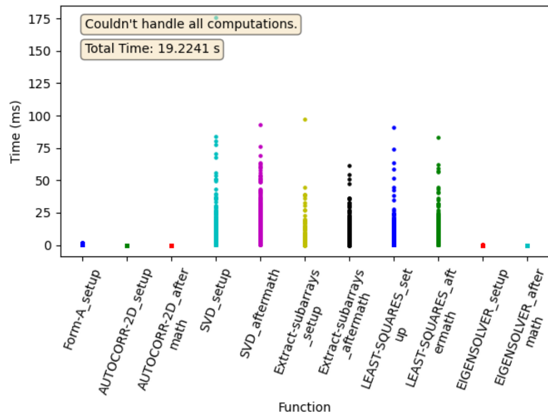
Figure A.31: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 20 threads with 25 CUDA streams per thread.



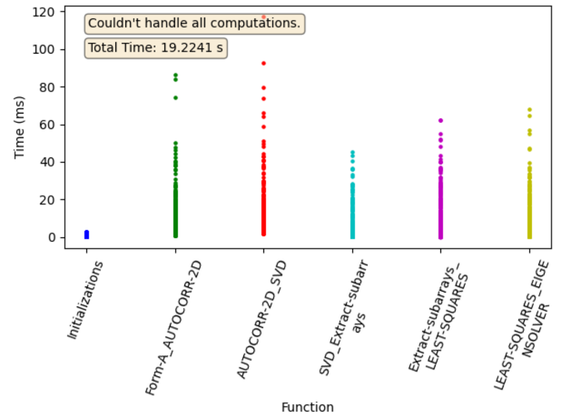
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

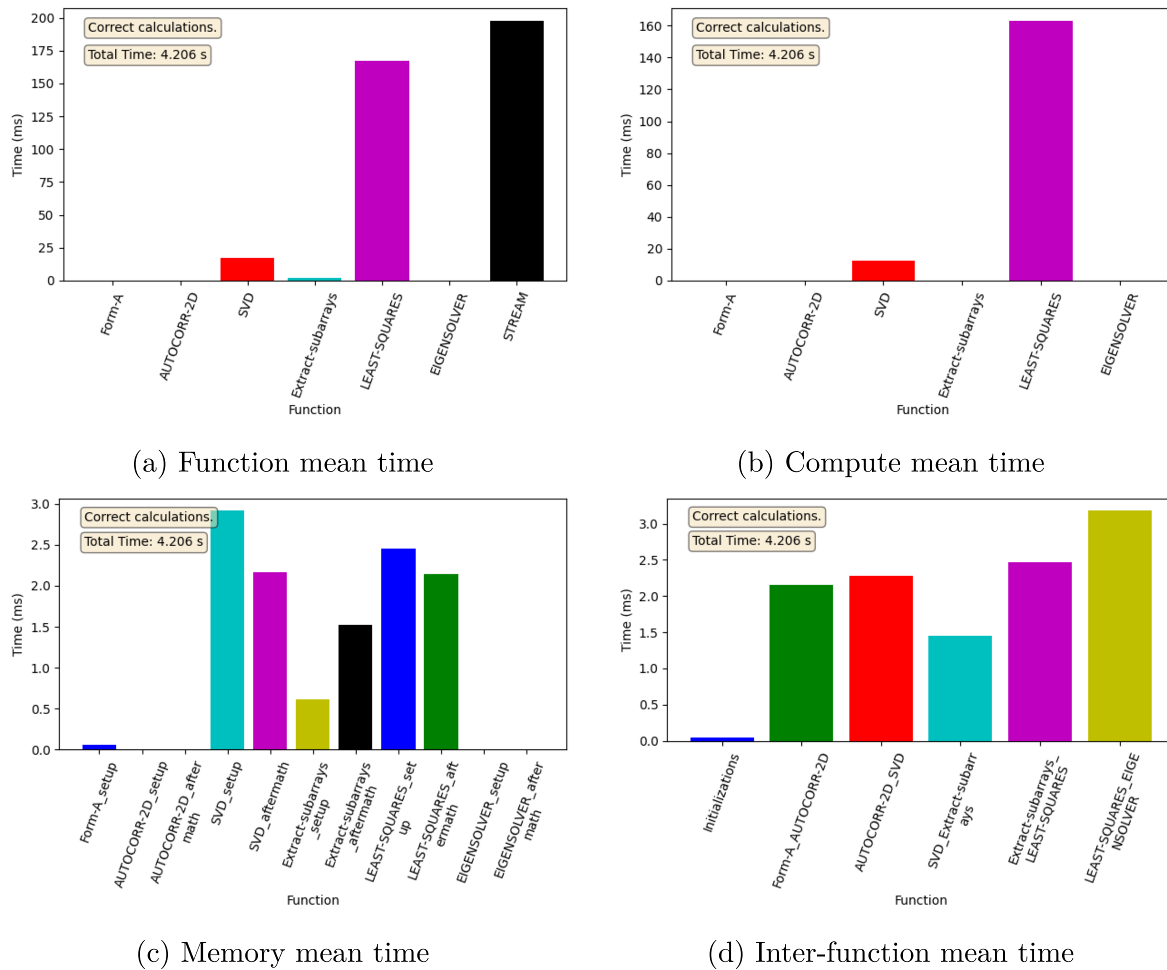


(d) Inter-function mean time

Figure A.32: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 20 threads with 25 CUDA streams per thread.

A.6 Version 2, 25 threads, 20 streams

A.6.1 FP32, size 64



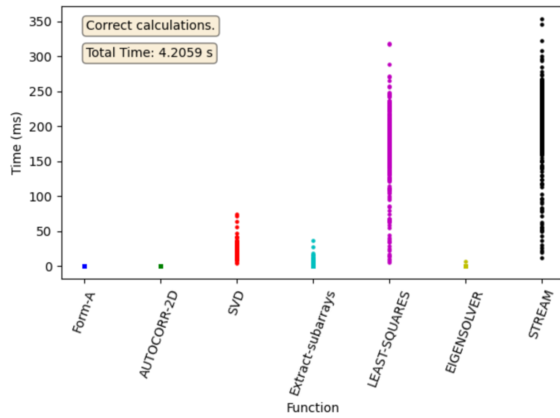
(a) Function mean time

(b) Compute mean time

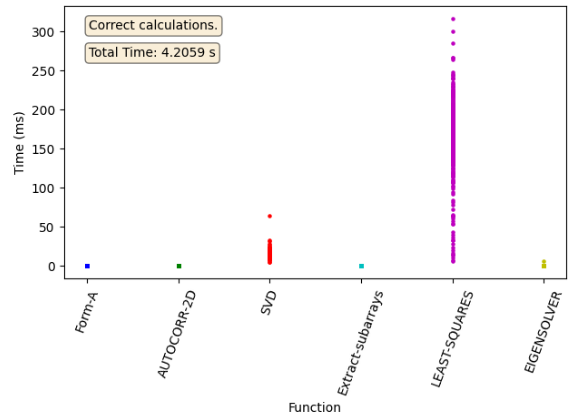
(c) Memory mean time

(d) Inter-function mean time

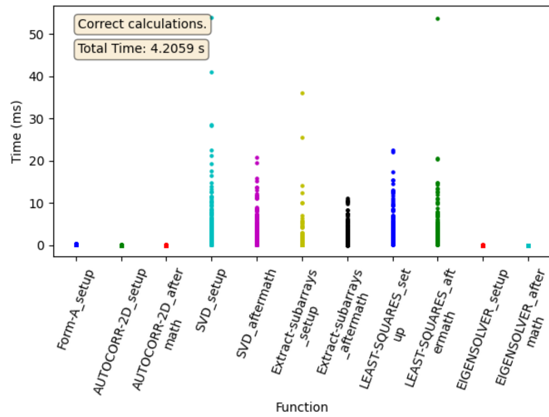
Figure A.33: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 25 threads with 20 CUDA streams per thread.



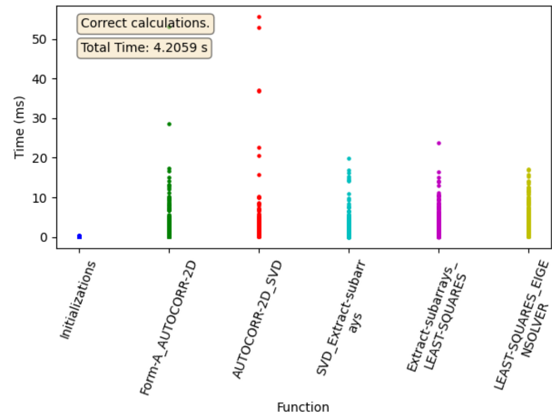
(a) Function mean time



(b) Compute mean time



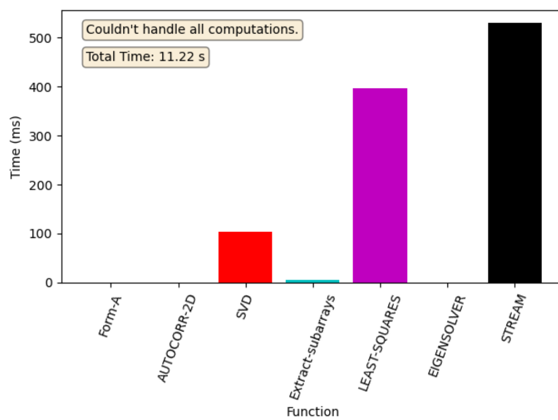
(c) Memory mean time



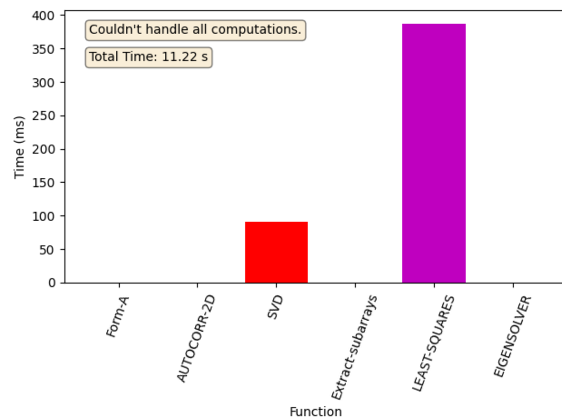
(d) Inter-function mean time

Figure A.34: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 25 threads with 20 CUDA streams per thread.

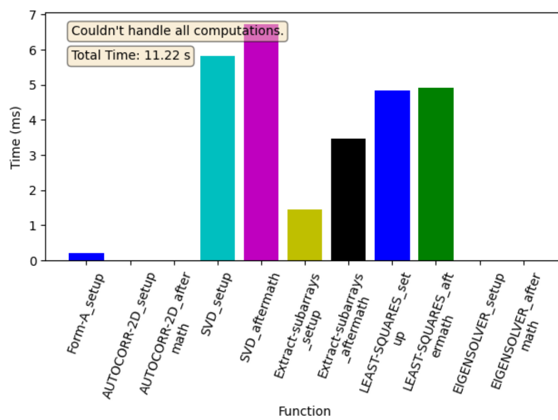
A.6.2 FP32, size 360



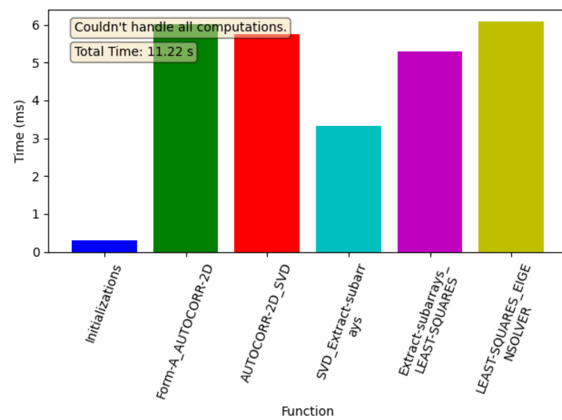
(a) Function mean time



(b) Compute mean time

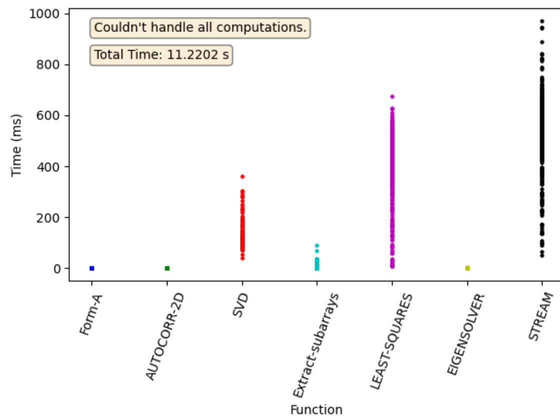


(c) Memory mean time

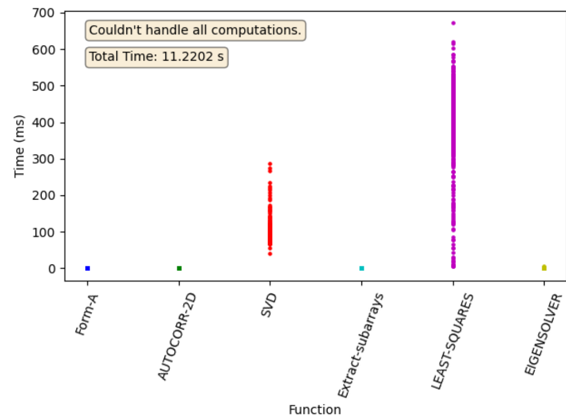


(d) Inter-function mean time

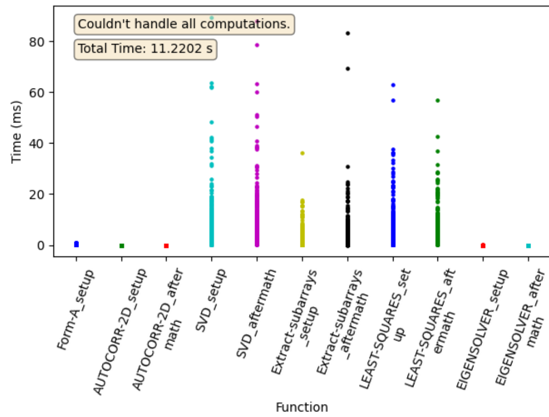
Figure A.35: Mean times for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 25 threads with 20 CUDA streams per thread.



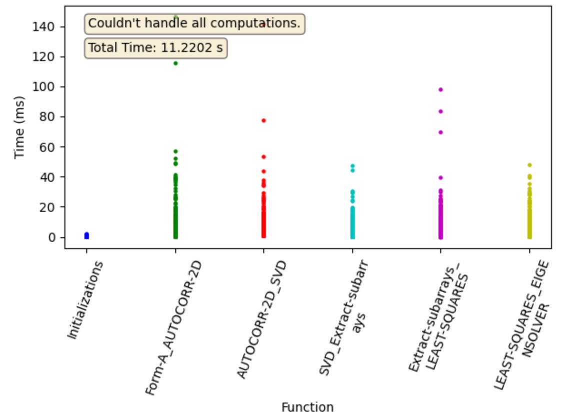
(a) Function mean time



(b) Compute mean time



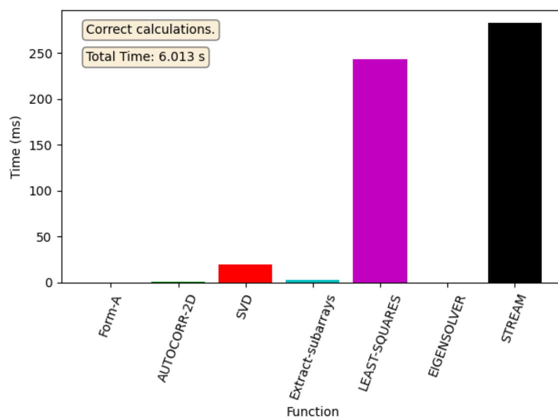
(c) Memory mean time



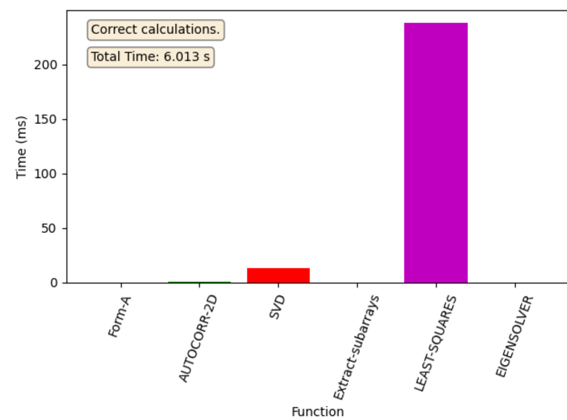
(d) Inter-function mean time

Figure A.36: Scatter plots for FP32 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 25 threads with 20 CUDA streams per thread.

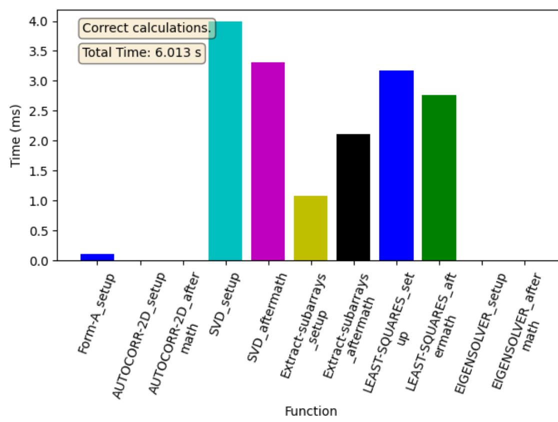
A.6.3 FP64, size 64



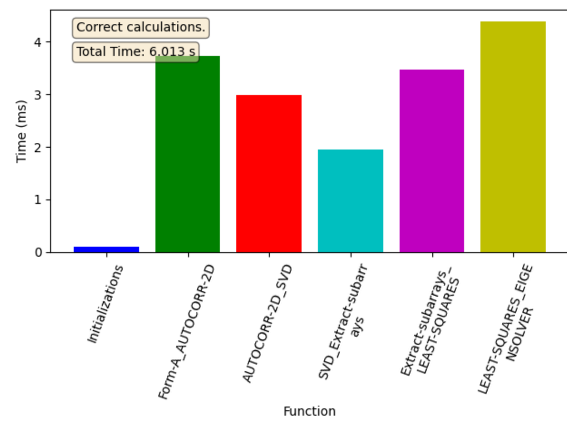
(a) Function mean time



(b) Compute mean time

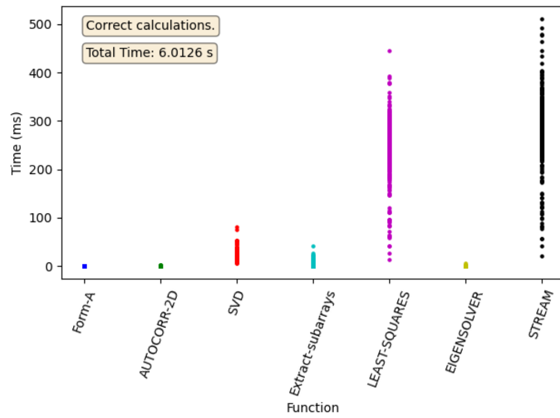


(c) Memory mean time

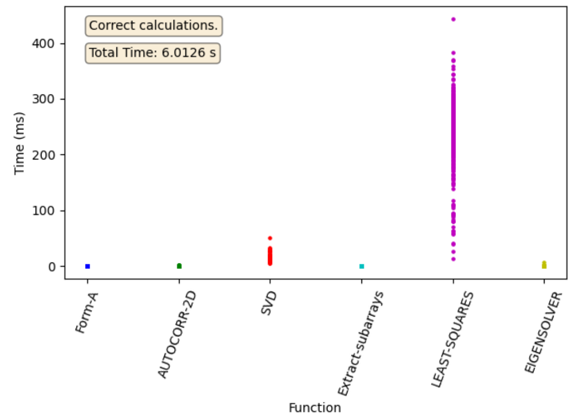


(d) Inter-function mean time

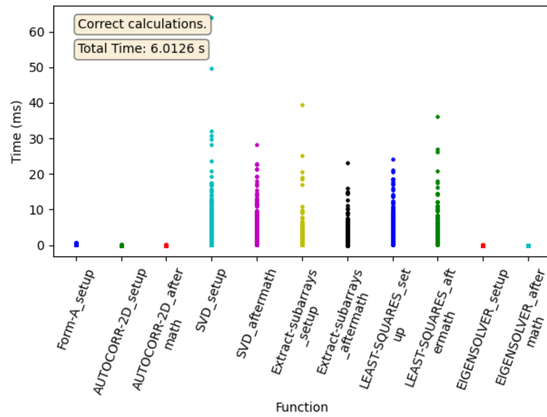
Figure A.37: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 25 threads with 20 CUDA streams per thread.



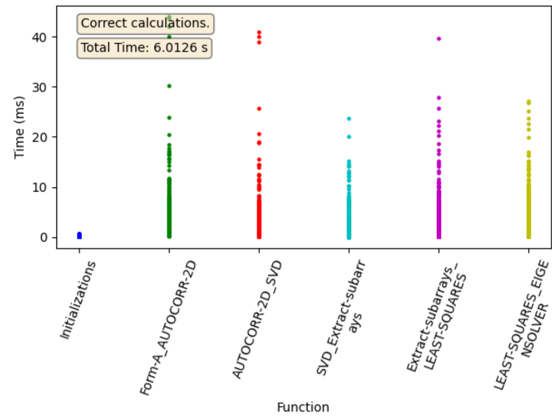
(a) Function mean time



(b) Compute mean time



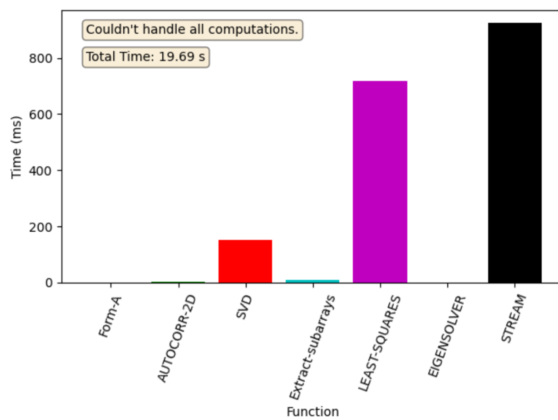
(c) Memory mean time



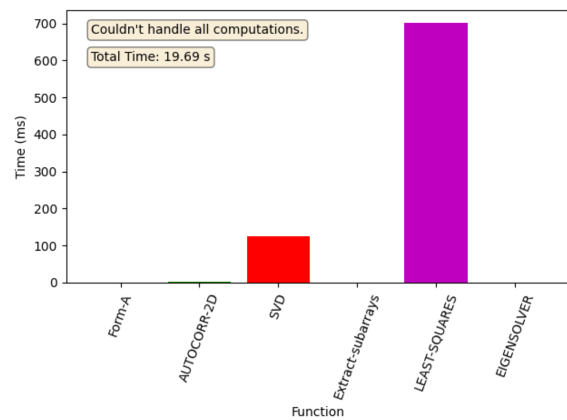
(d) Inter-function mean time

Figure A.38: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 64 elements. Configuration is 25 threads with 20 CUDA streams per thread.

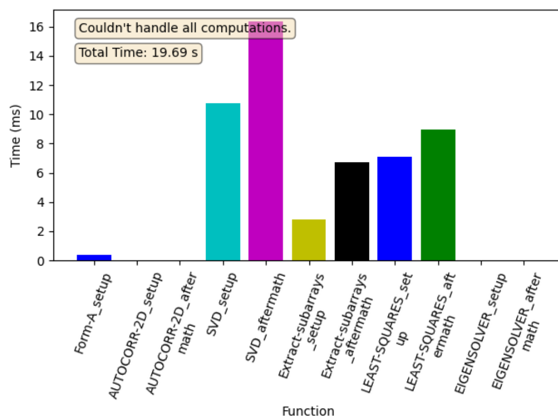
A.6.4 FP64, size 360



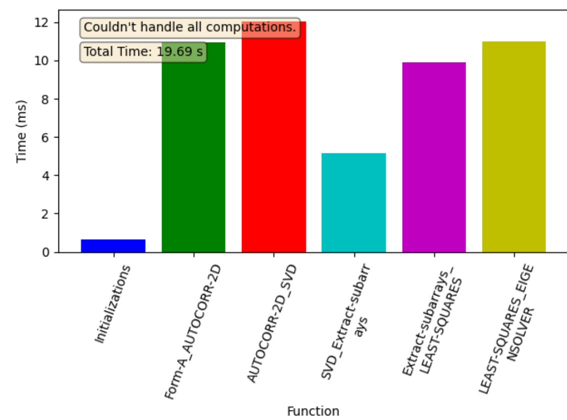
(a) Function mean time



(b) Compute mean time

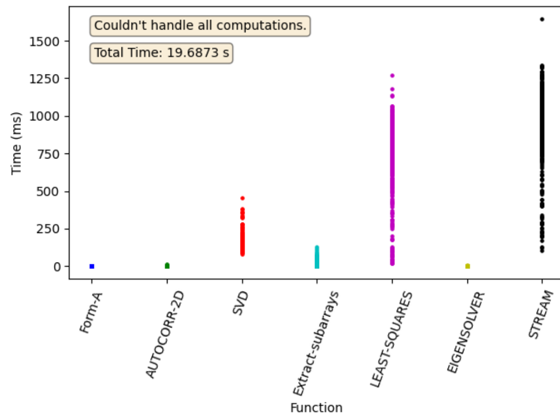


(c) Memory mean time

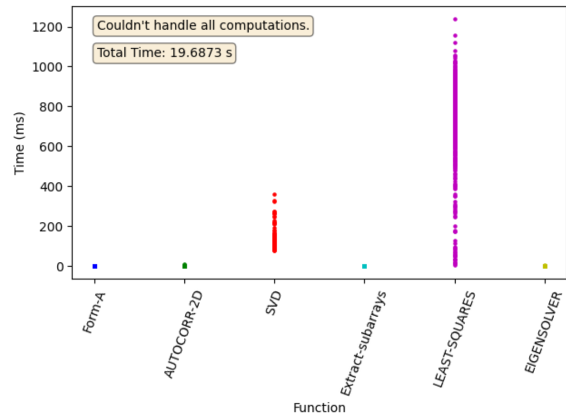


(d) Inter-function mean time

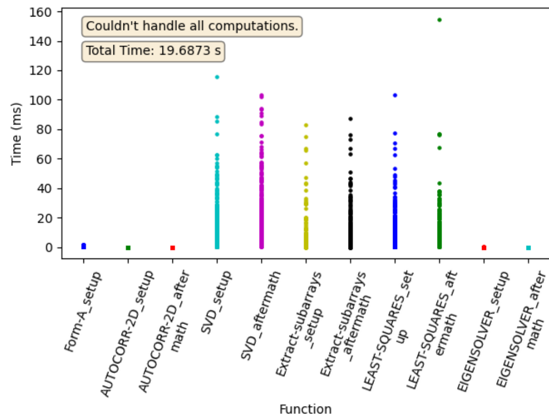
Figure A.39: Mean times for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 25 threads with 20 CUDA streams per thread.



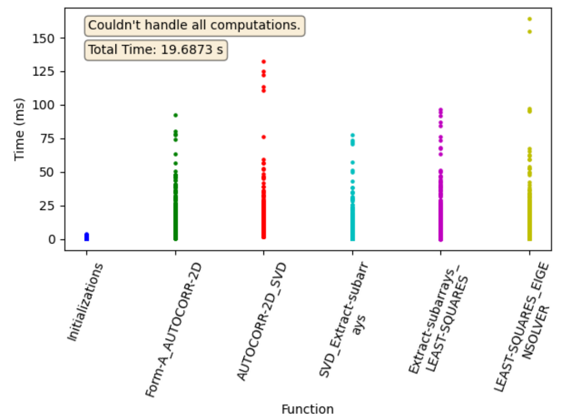
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

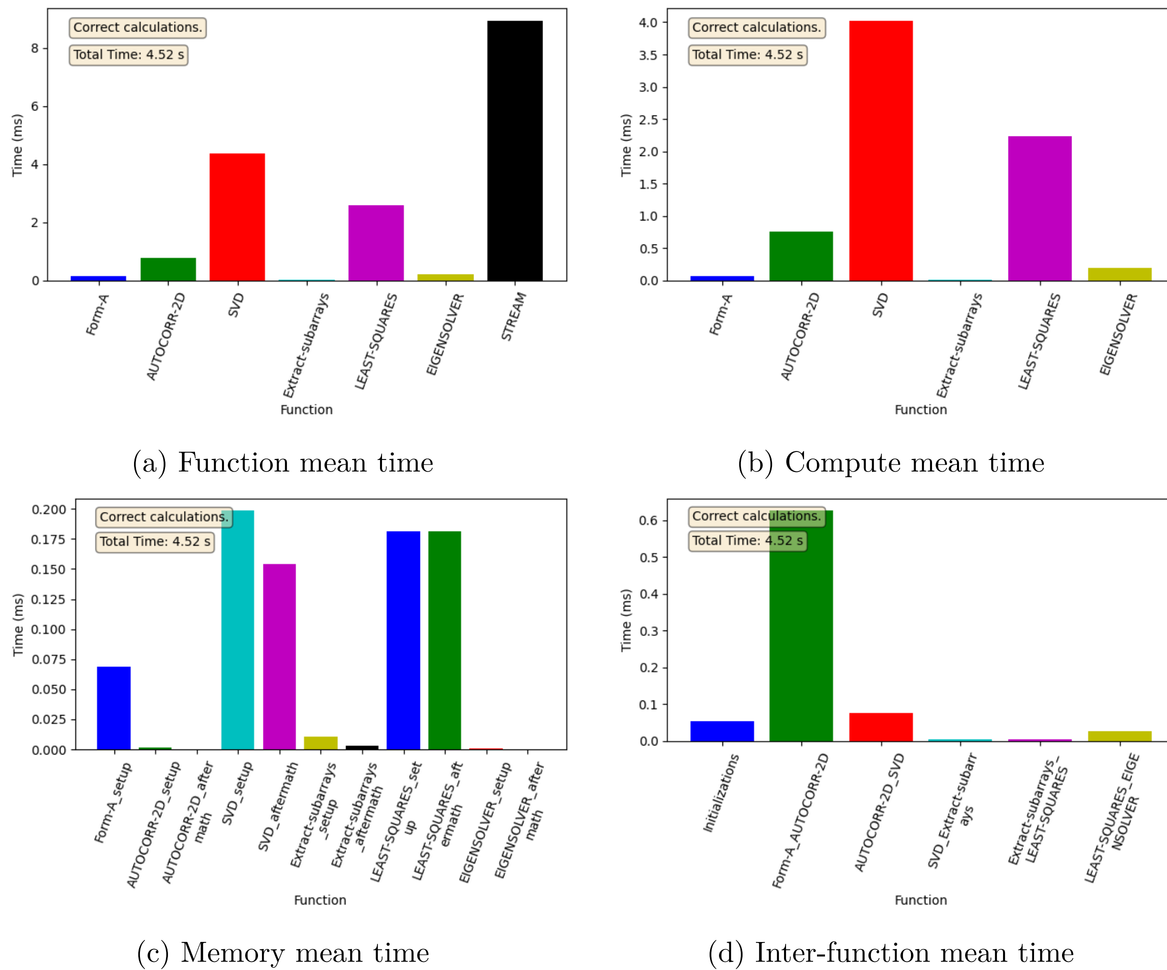


(d) Inter-function mean time

Figure A.40: Scatter plots for FP64 using the second version of the algorithm with a covariance matrix size of 360 elements. Configuration is 25 threads with 20 CUDA streams per thread.

A.7 Version 3, 1 thread, 500 streams

A.7.1 FP64, size 64



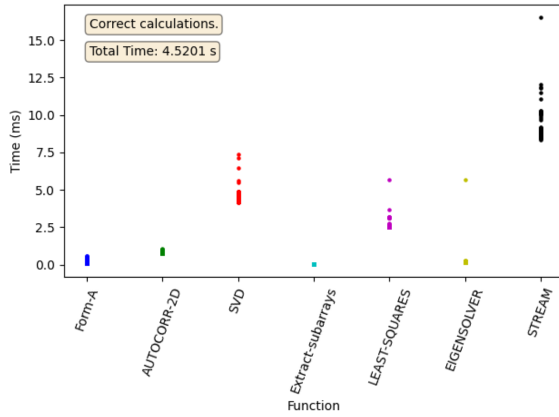
(a) Function mean time

(b) Compute mean time

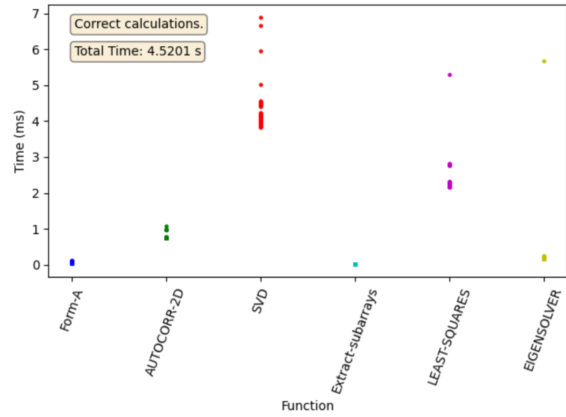
(c) Memory mean time

(d) Inter-function mean time

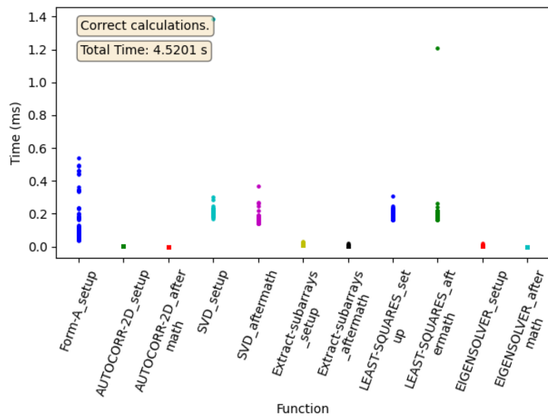
Figure A.41: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is one thread and 500 CUDA streams.



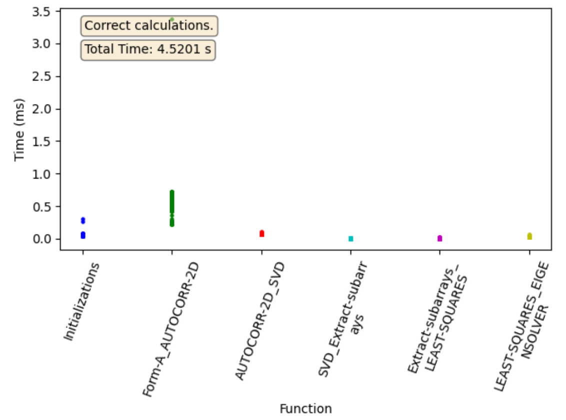
(a) Function mean time



(b) Compute mean time



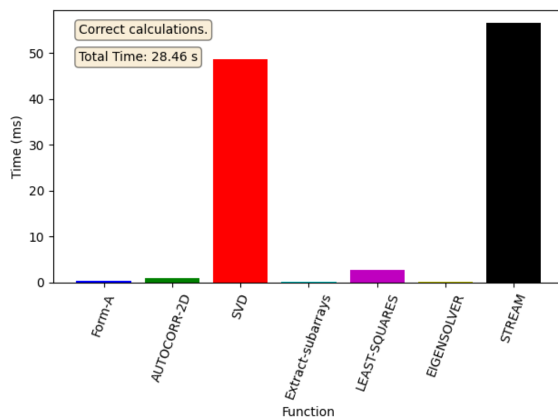
(c) Memory mean time



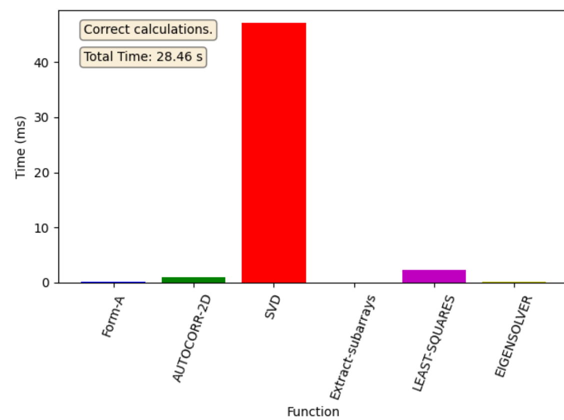
(d) Inter-function mean time

Figure A.42: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is one thread and 500 CUDA streams.

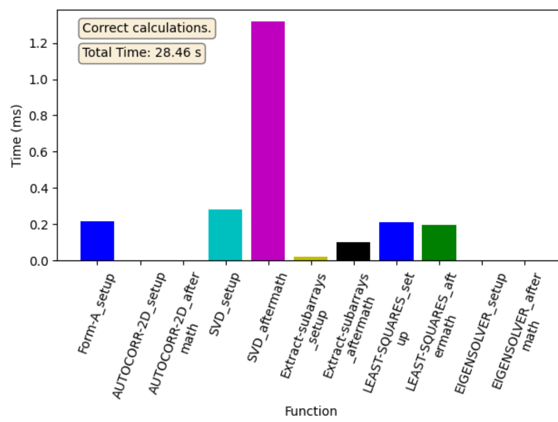
A.7.2 FP64, size 360



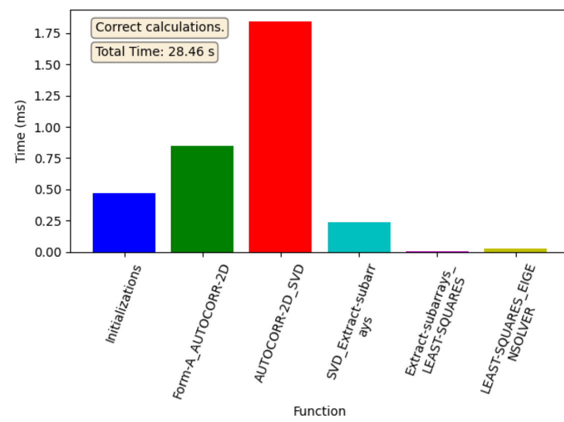
(a) Function mean time



(b) Compute mean time

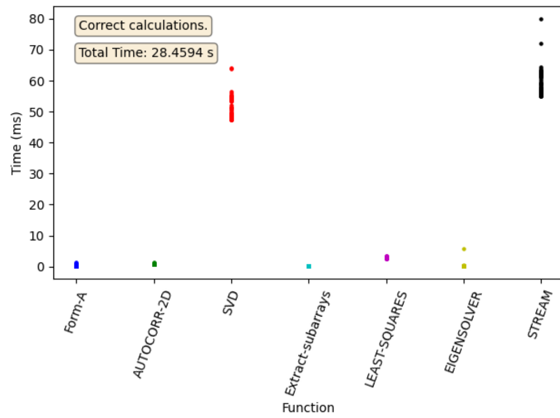


(c) Memory mean time

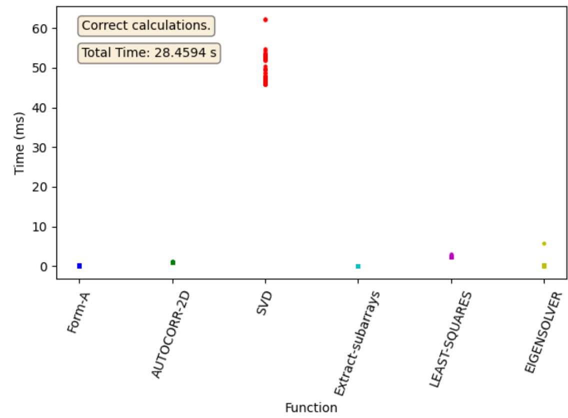


(d) Inter-function mean time

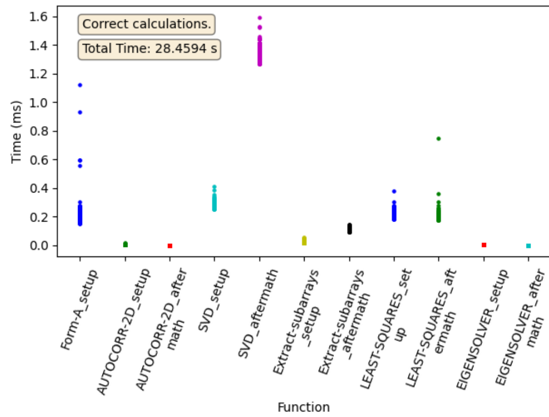
Figure A.43: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is one thread and 500 CUDA streams.



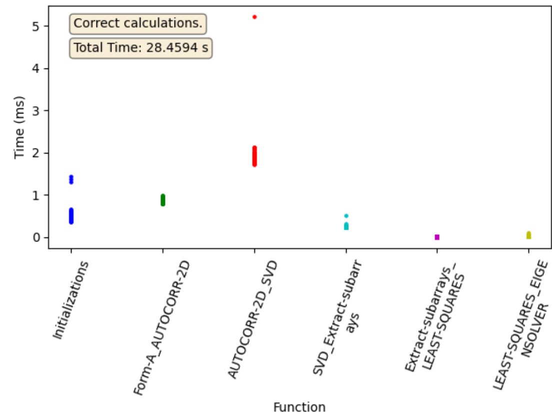
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

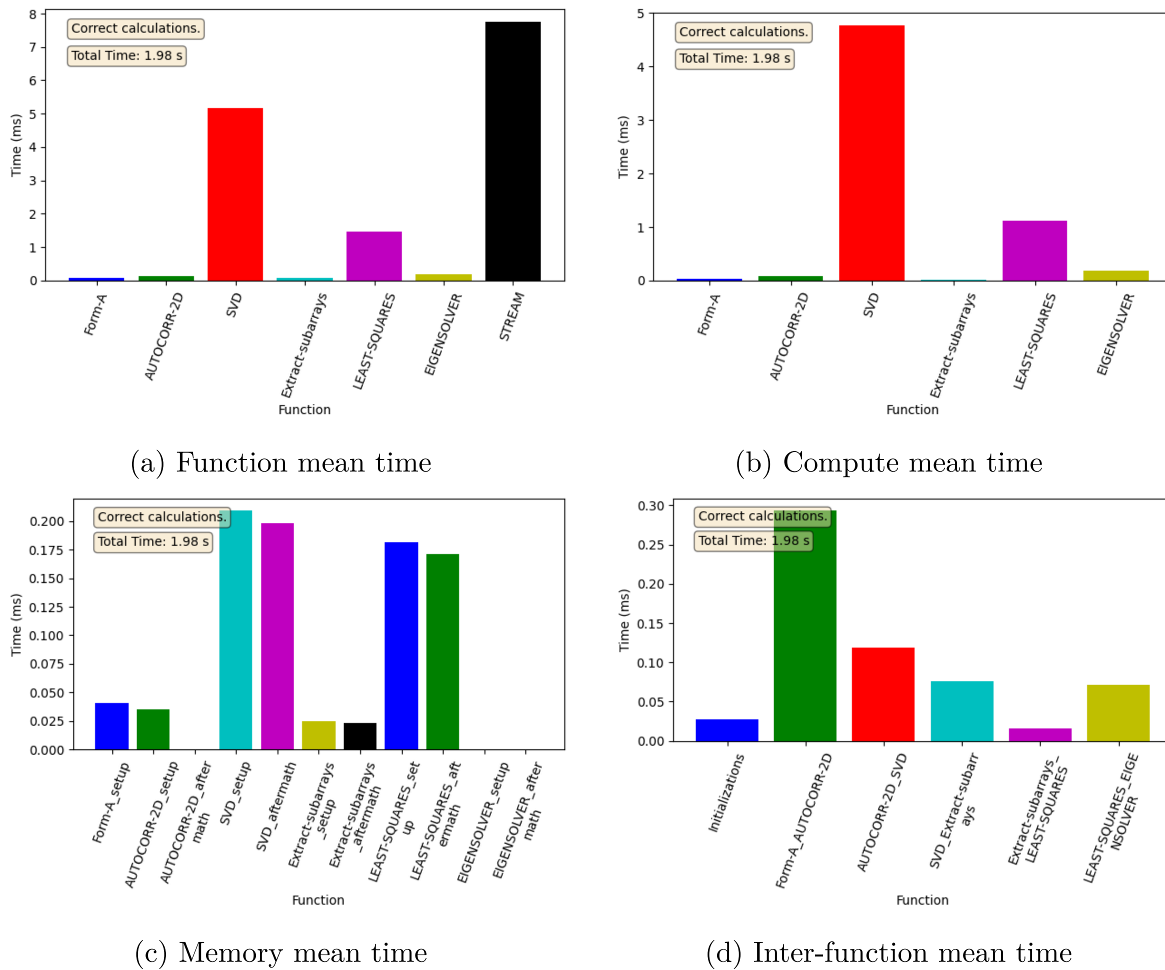


(d) Inter-function mean time

Figure A.44: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is one thread and 500 CUDA streams.

A.8 Version 3, 2 threads, 250 streams

A.8.1 FP32, size 64



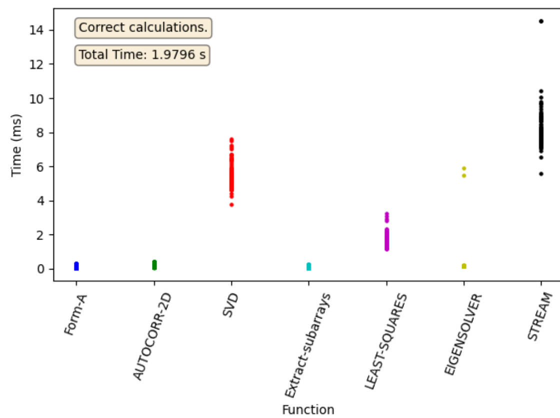
(a) Function mean time

(b) Compute mean time

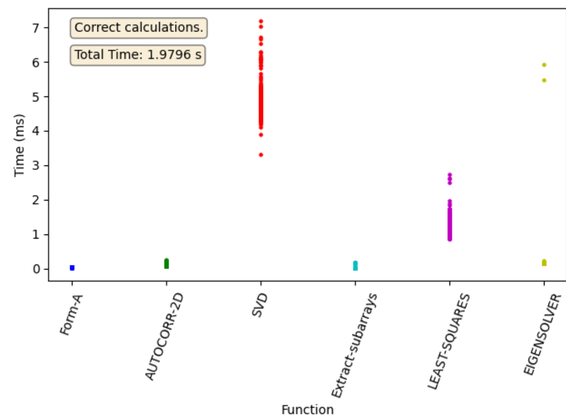
(c) Memory mean time

(d) Inter-function mean time

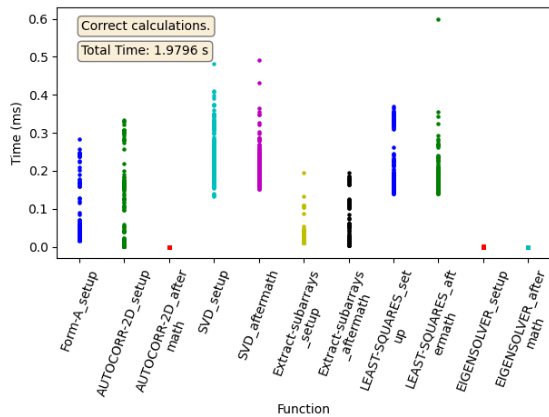
Figure A.45: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is two threads with 250 CUDA streams per thread.



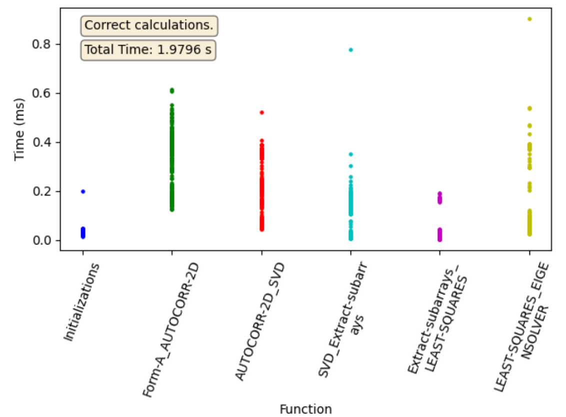
(a) Function mean time



(b) Compute mean time



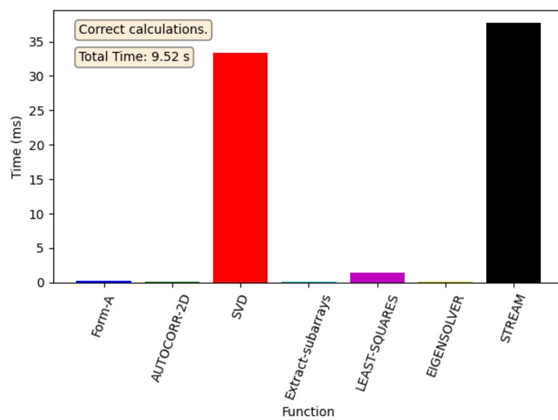
(c) Memory mean time



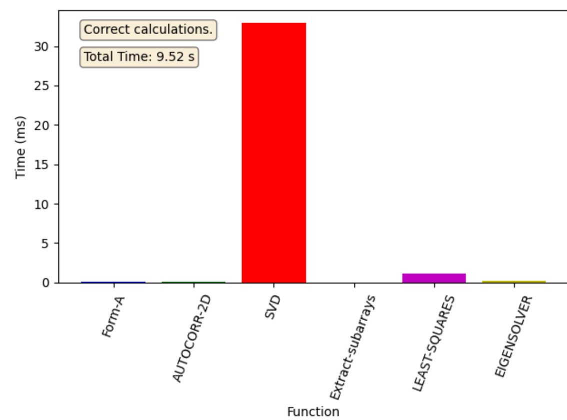
(d) Inter-function mean time

Figure A.46: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is two threads with 250 CUDA streams per thread.

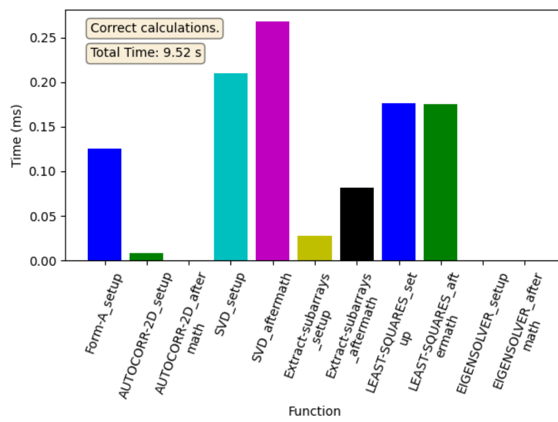
A.8.2 FP32, size 360



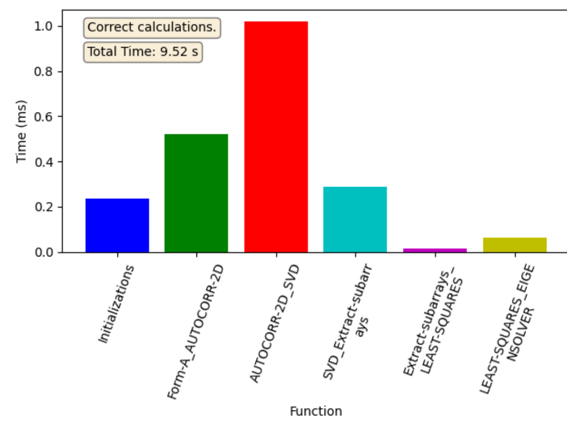
(a) Function mean time



(b) Compute mean time

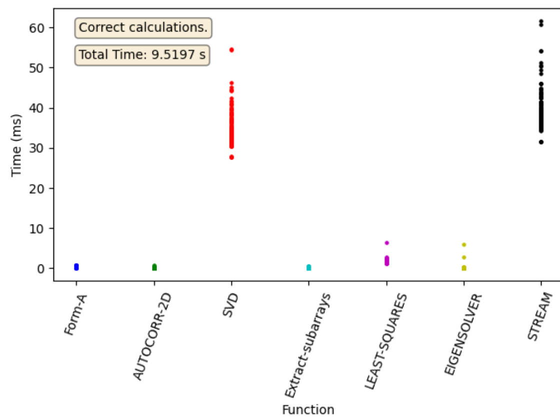


(c) Memory mean time

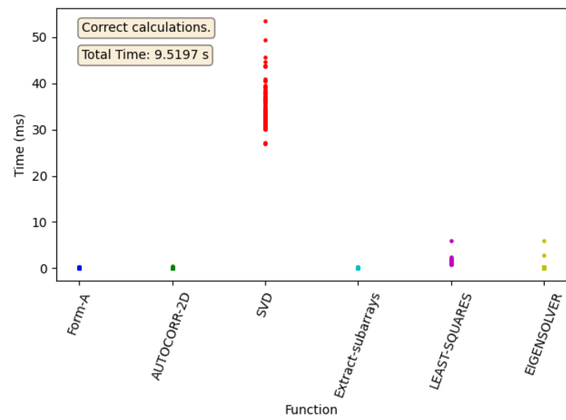


(d) Inter-function mean time

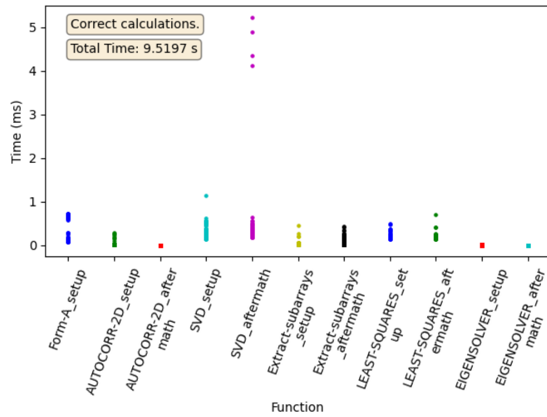
Figure A.47: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is two threads with 250 CUDA streams per thread.



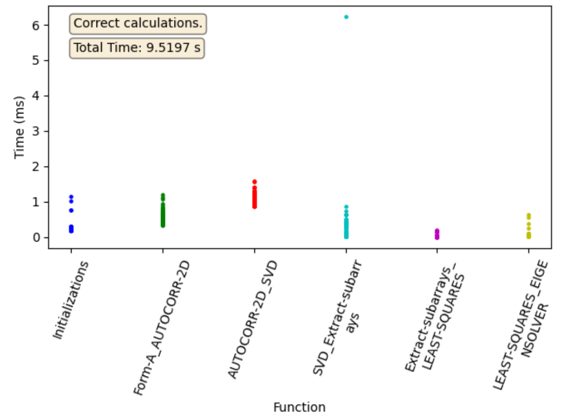
(a) Function mean time



(b) Compute mean time



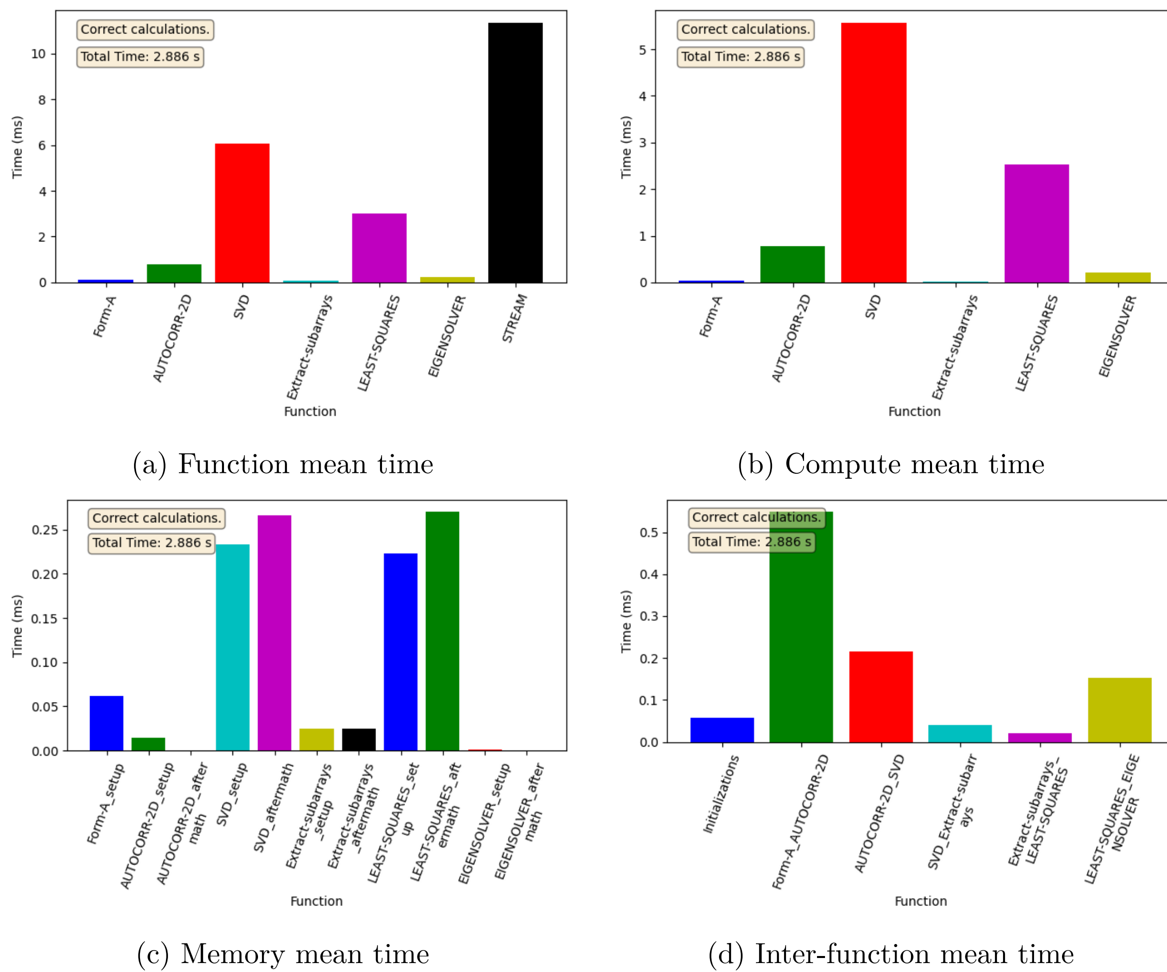
(c) Memory mean time



(d) Inter-function mean time

Figure A.48: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is two threads with 250 CUDA streams per thread.

A.8.3 FP64, size 64



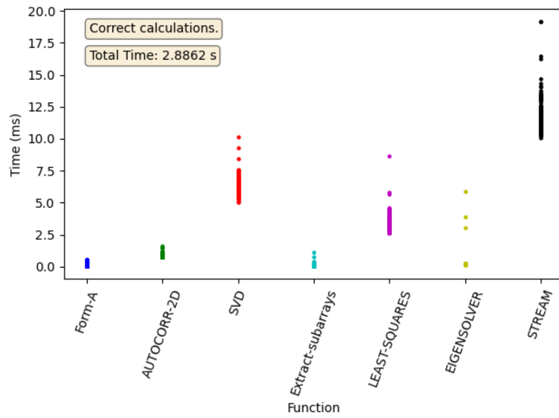
(a) Function mean time

(b) Compute mean time

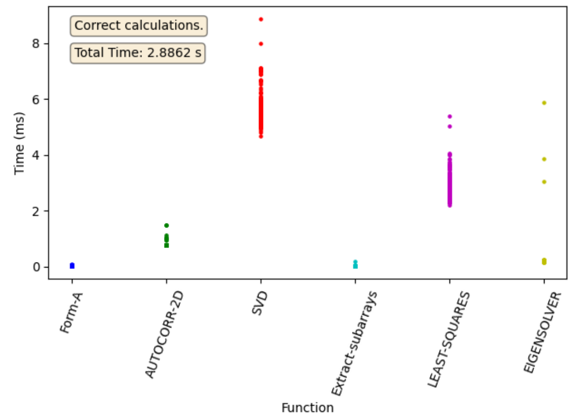
(c) Memory mean time

(d) Inter-function mean time

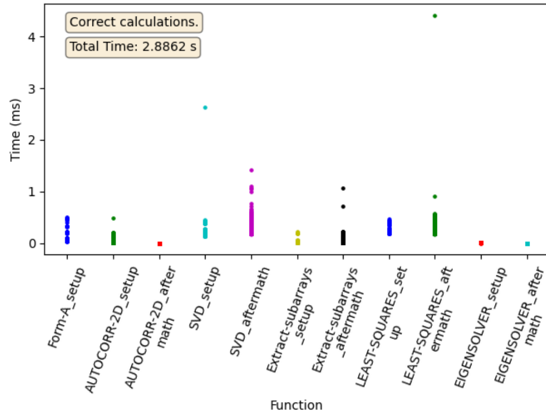
Figure A.49: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is two threads with 250 CUDA streams per thread.



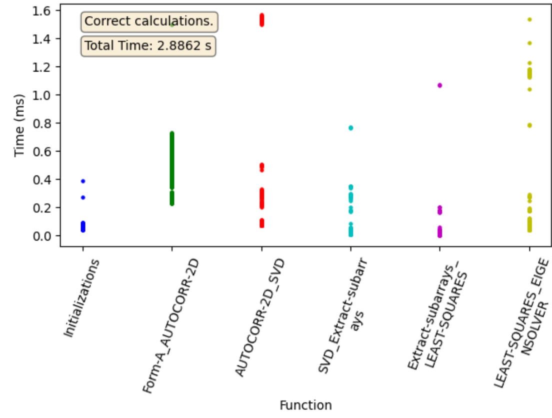
(a) Function mean time



(b) Compute mean time



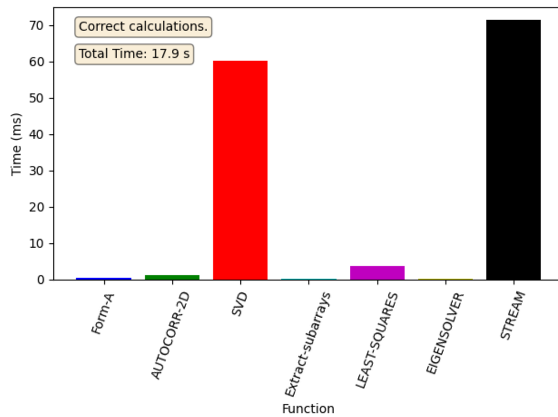
(c) Memory mean time



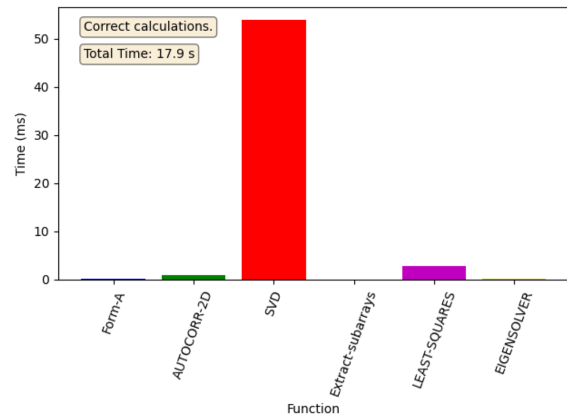
(d) Inter-function mean time

Figure A.50: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is two threads with 250 CUDA streams per thread.

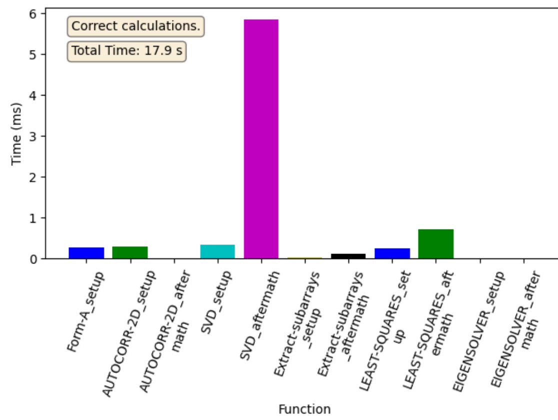
A.8.4 FP64, size 360



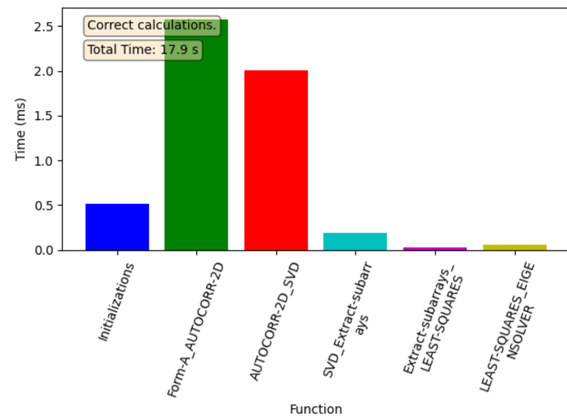
(a) Function mean time



(b) Compute mean time

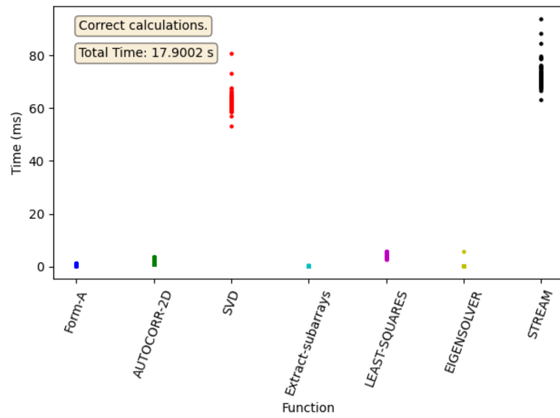


(c) Memory mean time

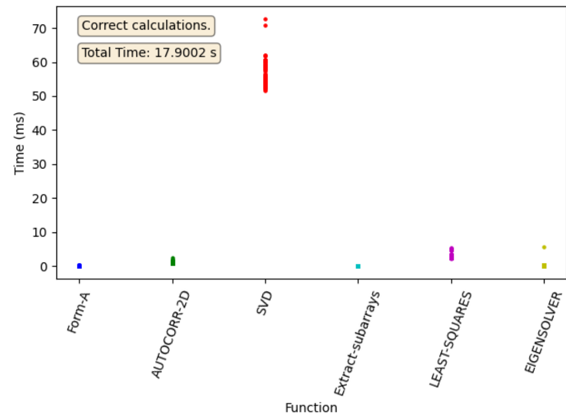


(d) Inter-function mean time

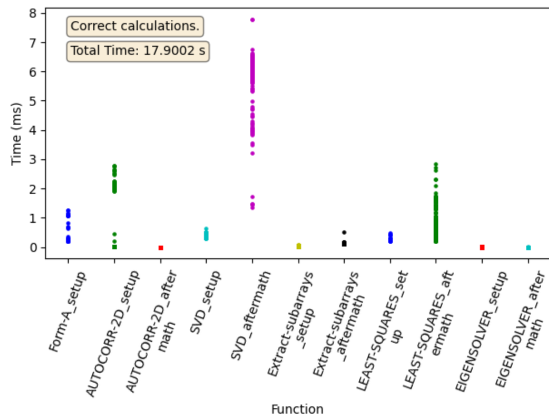
Figure A.51: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is two threads with 250 CUDA streams per thread.



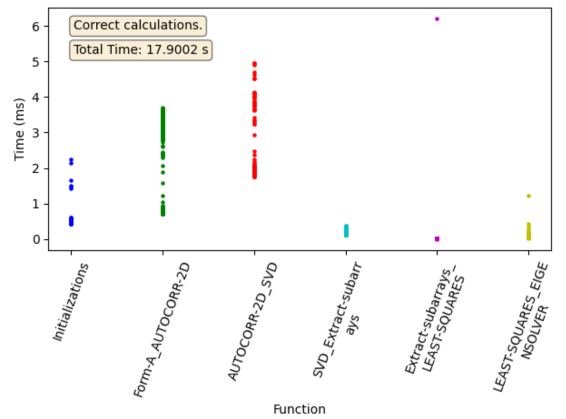
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

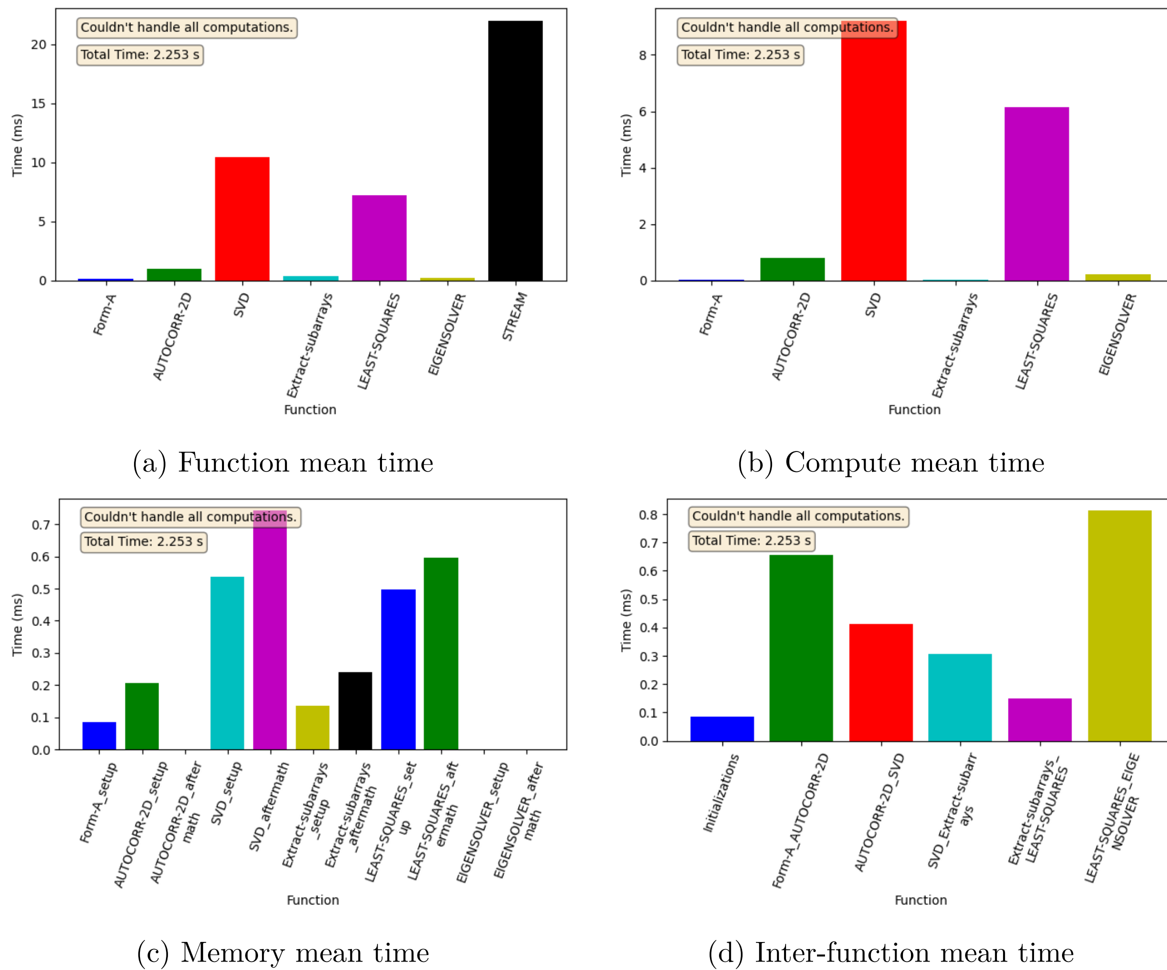


(d) Inter-function mean time

Figure A.52: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is two threads with 250 CUDA streams per thread.

A.9 Version 3, 5 threads, 100 streams

A.9.1 FP64, size 64



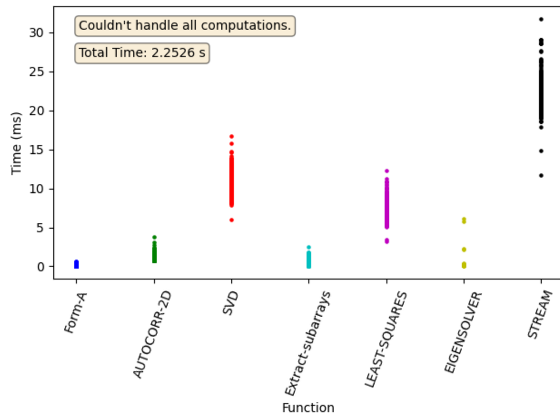
(a) Function mean time

(b) Compute mean time

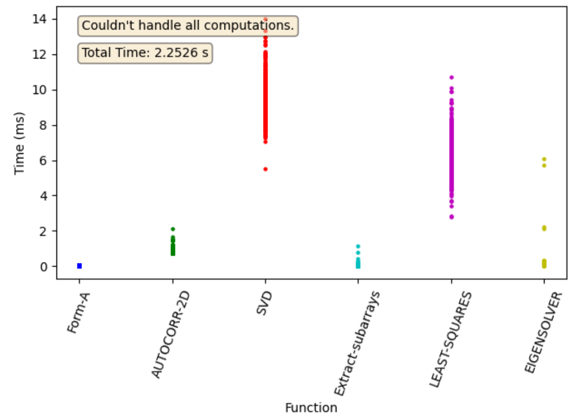
(c) Memory mean time

(d) Inter-function mean time

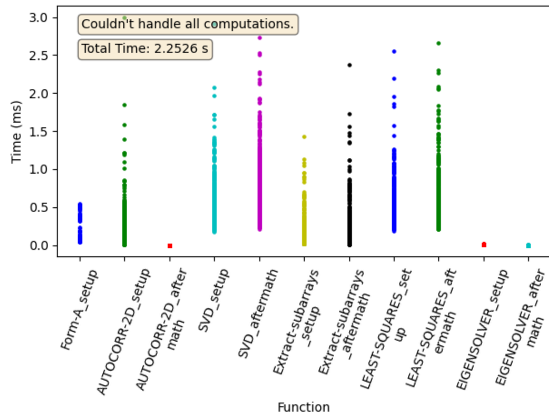
Figure A.53: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 5 threads with 100 CUDA streams per thread.



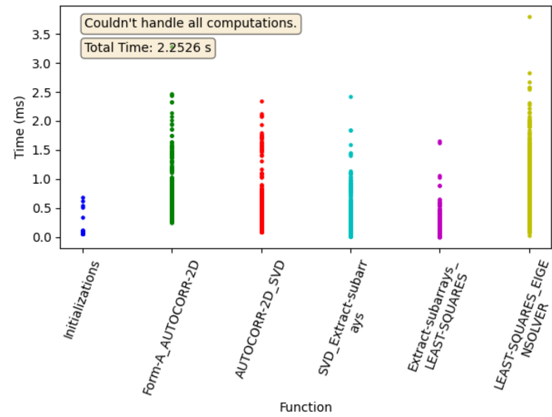
(a) Function mean time



(b) Compute mean time



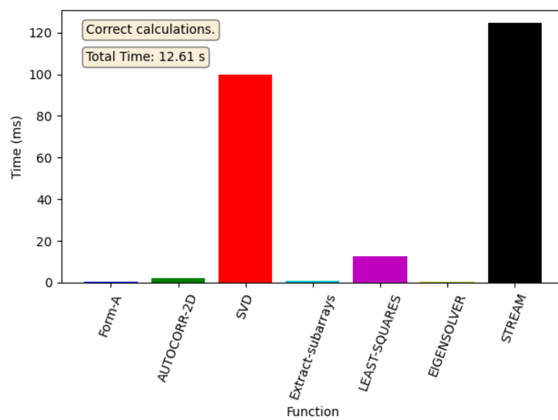
(c) Memory mean time



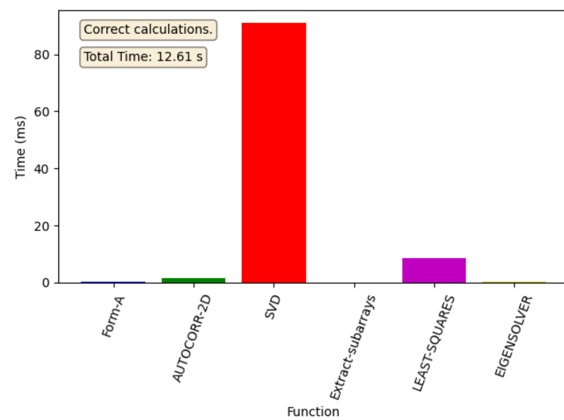
(d) Inter-function mean time

Figure A.54: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 5 threads with 100 CUDA streams per thread.

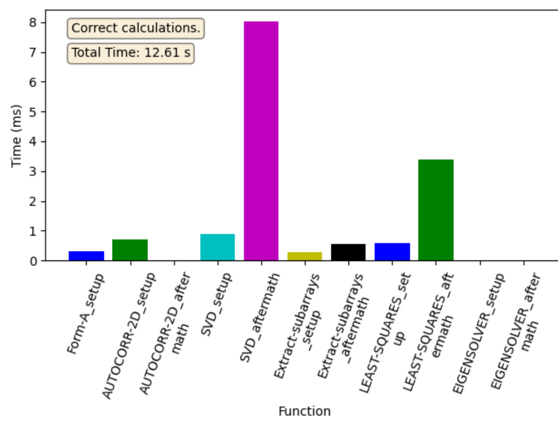
A.9.2 FP64, size 360



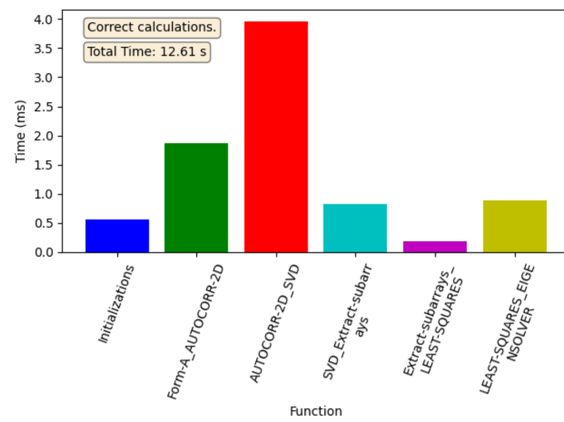
(a) Function mean time



(b) Compute mean time

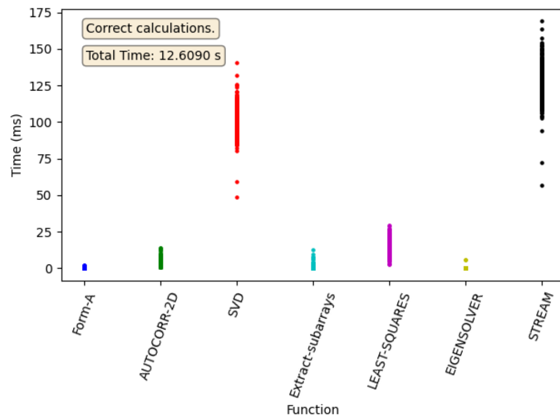


(c) Memory mean time

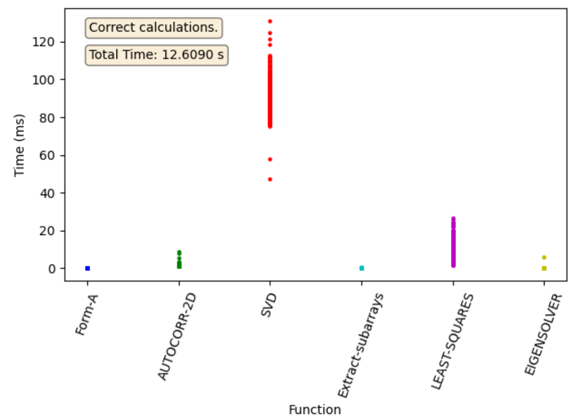


(d) Inter-function mean time

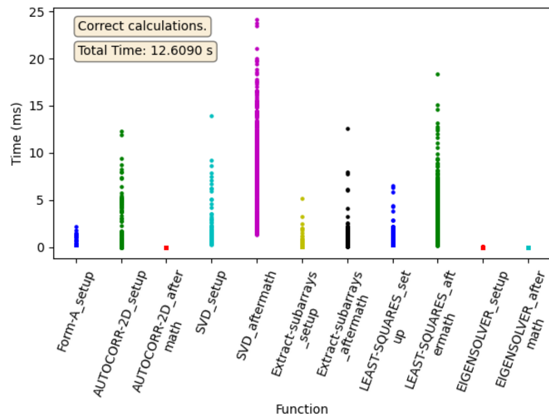
Figure A.55: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 5 threads with 100 CUDA streams per thread.



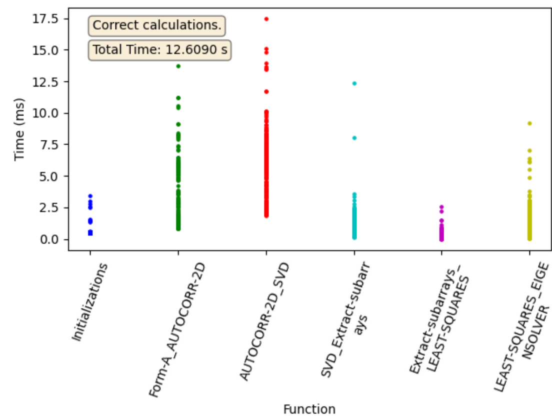
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

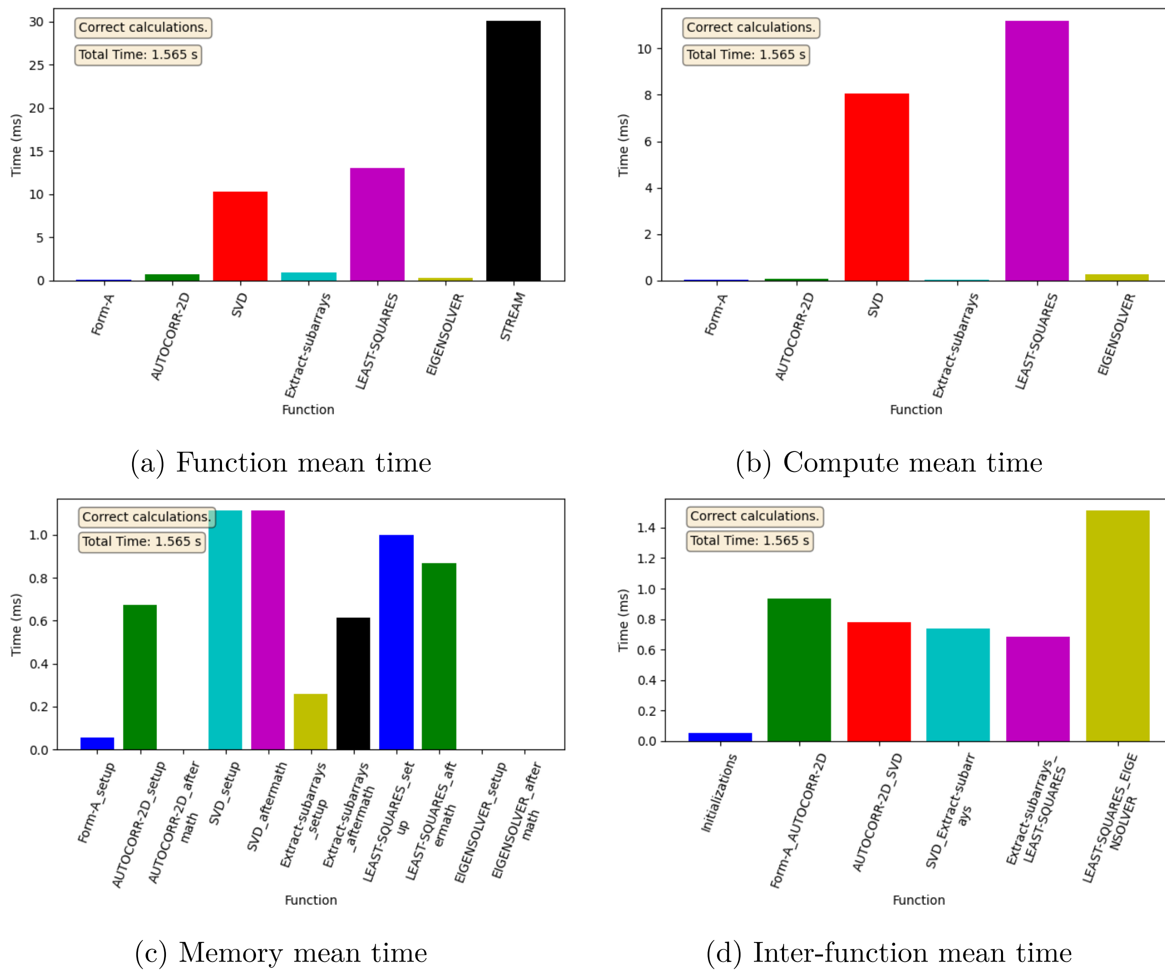


(d) Inter-function mean time

Figure A.56: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 5 threads with 100 CUDA streams per thread.

A.10 Version 3, 10 threads, 50 streams

A.10.1 FP32, size 64



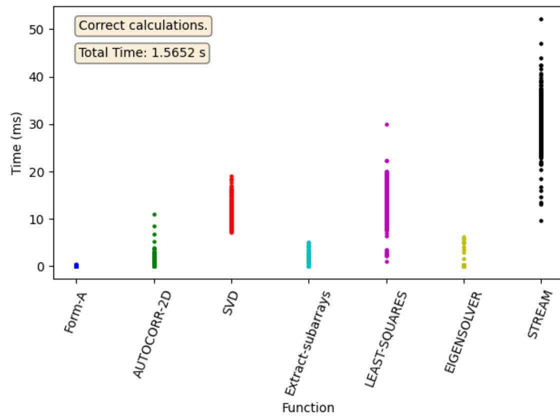
(a) Function mean time

(b) Compute mean time

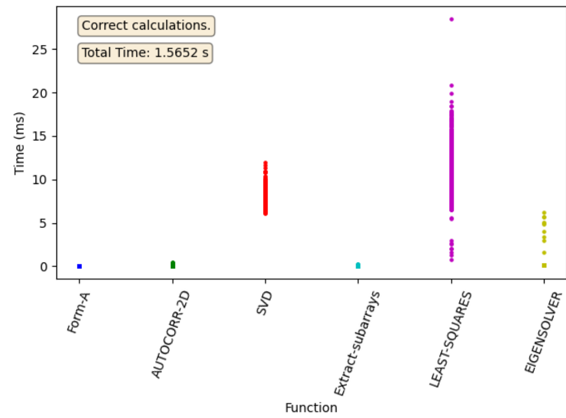
(c) Memory mean time

(d) Inter-function mean time

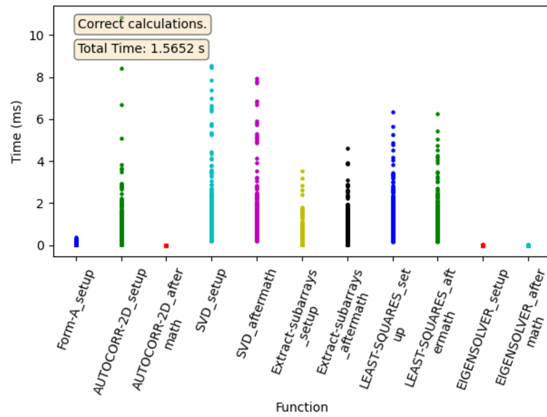
Figure A.57: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 10 threads with 50 CUDA streams per thread.



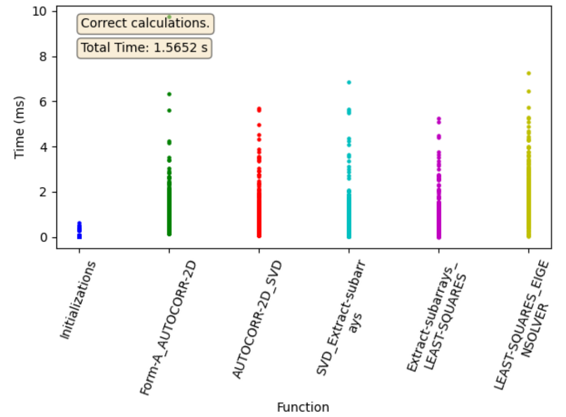
(a) Function mean time



(b) Compute mean time



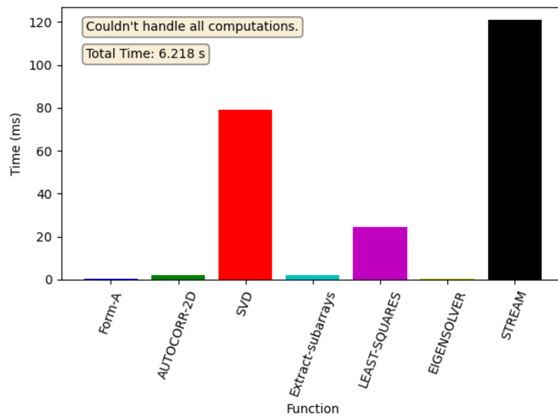
(c) Memory mean time



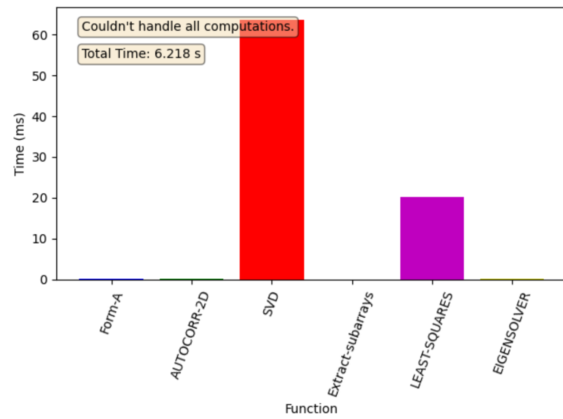
(d) Inter-function mean time

Figure A.58: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 10 threads with 50 CUDA streams per thread.

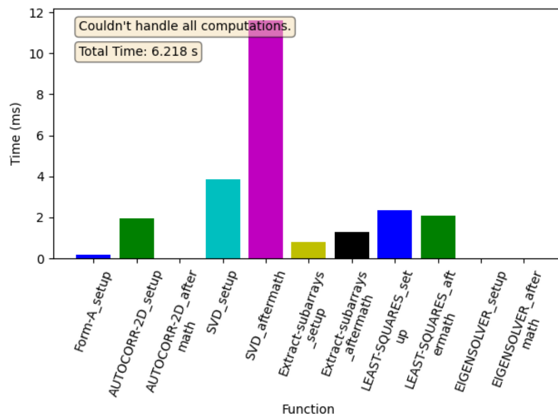
A.10.2 FP32, size 360



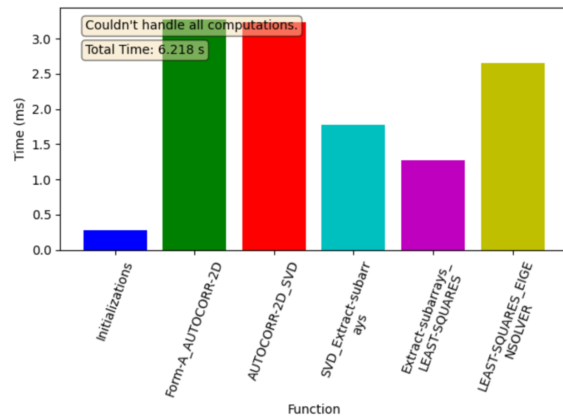
(a) Function mean time



(b) Compute mean time

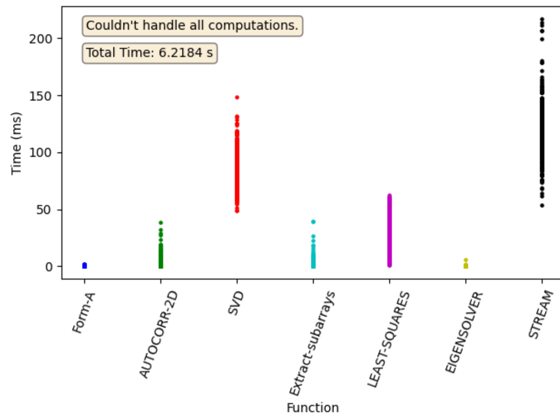


(c) Memory mean time

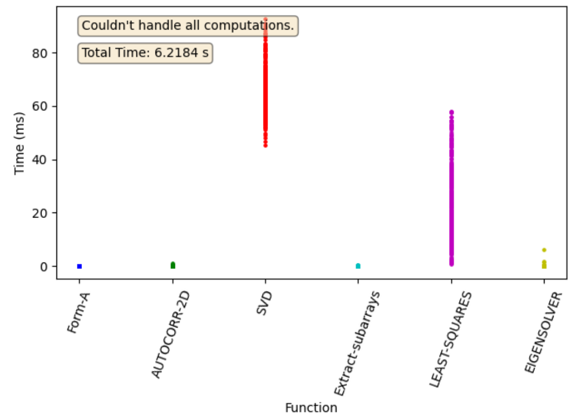


(d) Inter-function mean time

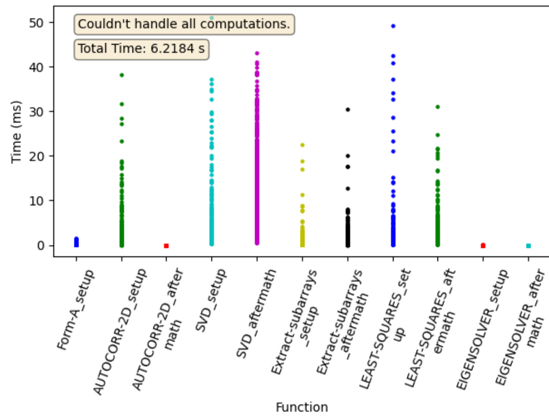
Figure A.59: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 10 threads with 50 CUDA streams per thread.



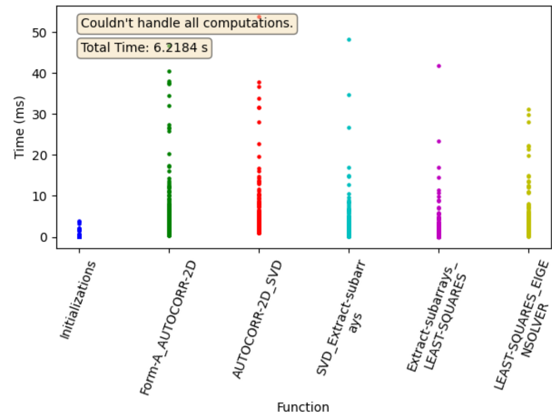
(a) Function mean time



(b) Compute mean time



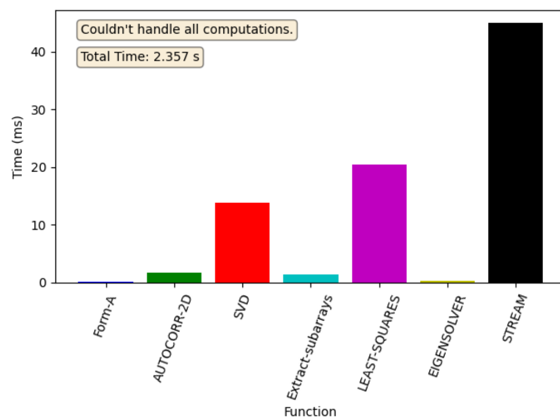
(c) Memory mean time



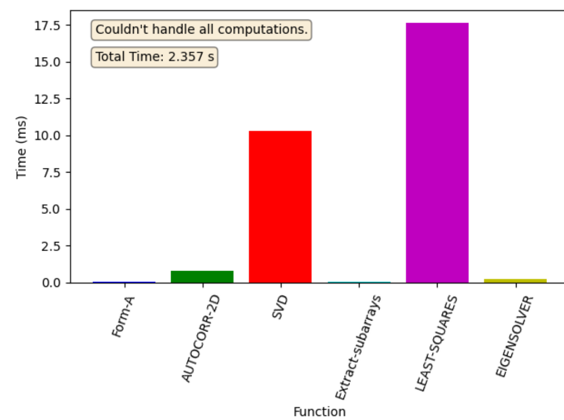
(d) Inter-function mean time

Figure A.60: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 10 threads with 50 CUDA streams per thread.

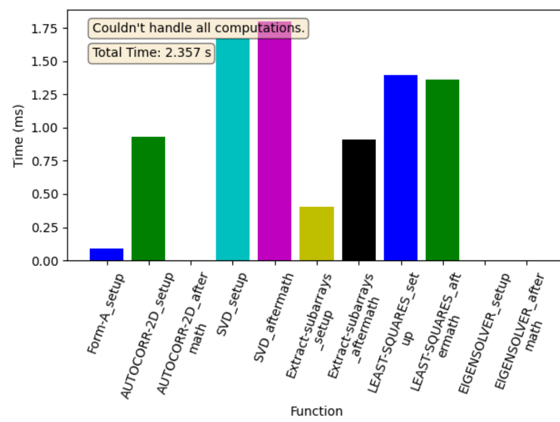
A.10.3 FP64, size 64



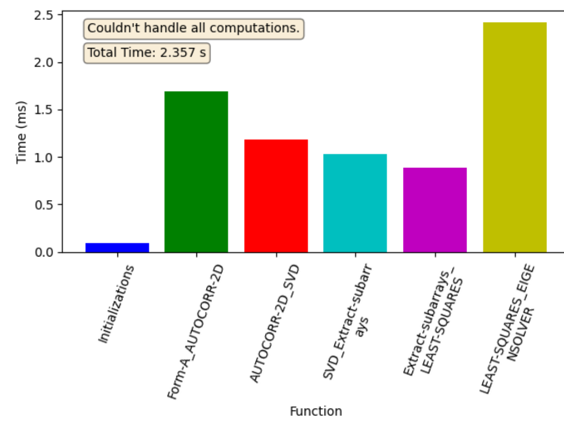
(a) Function mean time



(b) Compute mean time

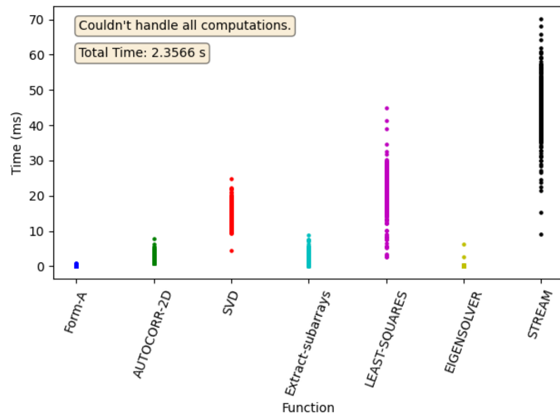


(c) Memory mean time

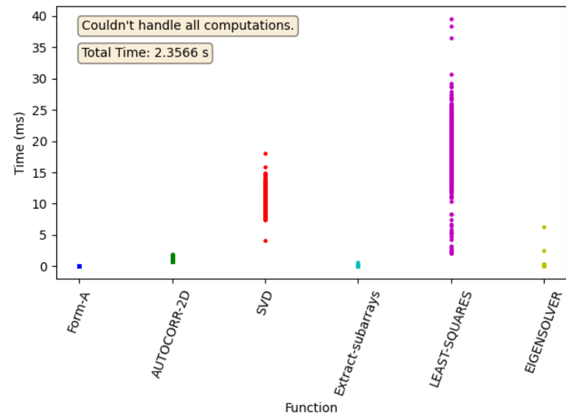


(d) Inter-function mean time

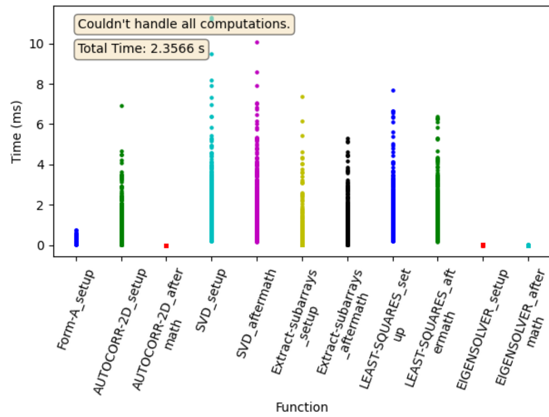
Figure A.61: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 10 threads with 50 CUDA streams per thread.



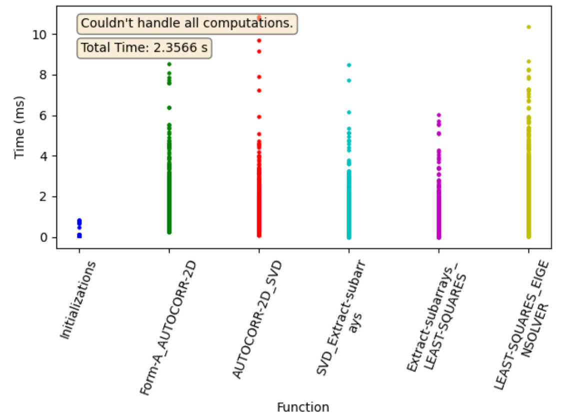
(a) Function mean time



(b) Compute mean time



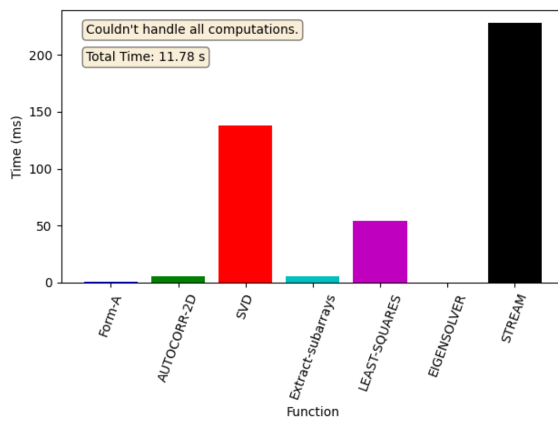
(c) Memory mean time



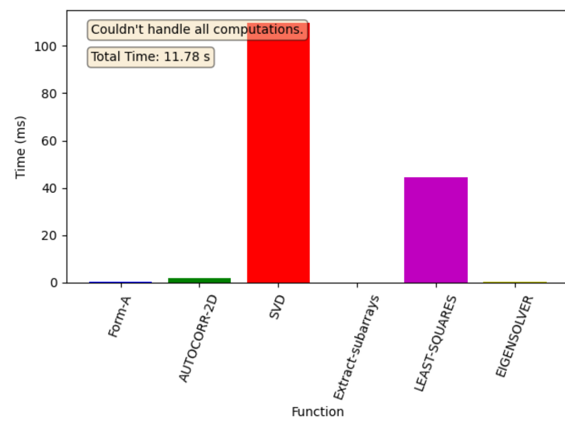
(d) Inter-function mean time

Figure A.62: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 10 threads with 50 CUDA streams per thread.

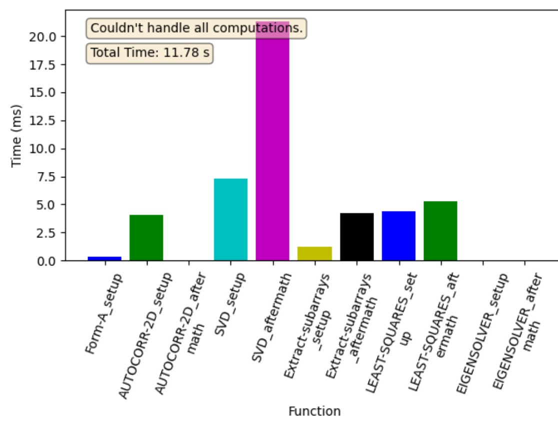
A.10.4 FP64, size 360



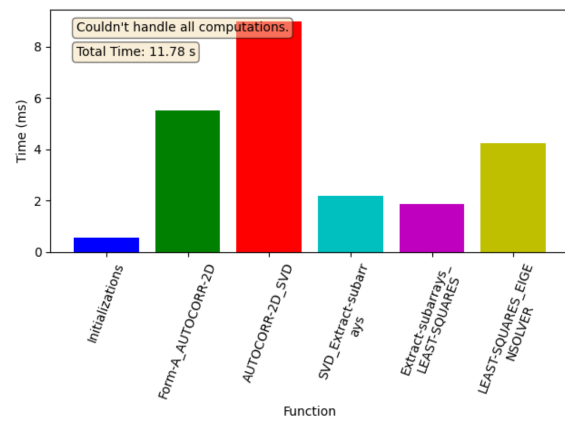
(a) Function mean time



(b) Compute mean time

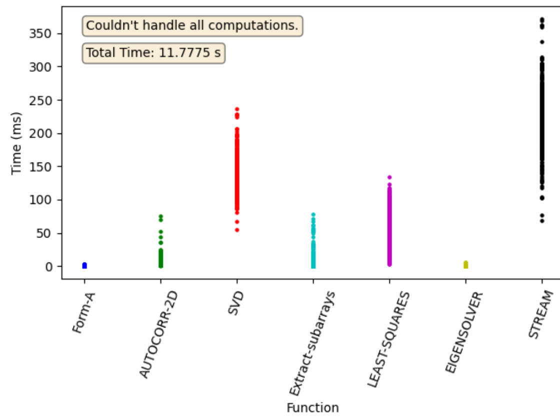


(c) Memory mean time

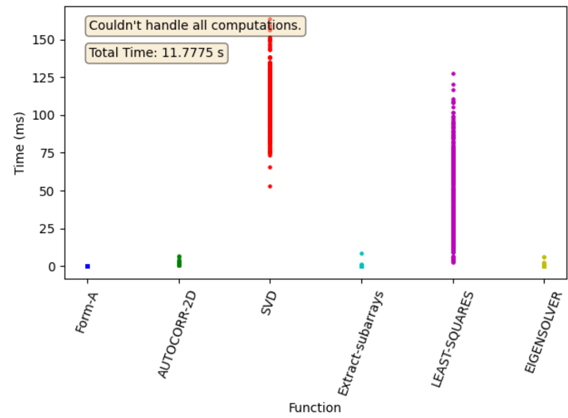


(d) Inter-function mean time

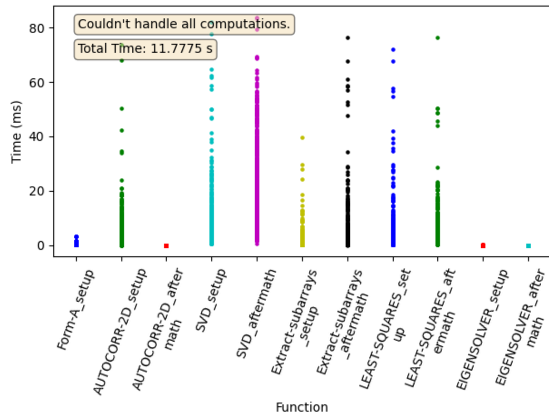
Figure A.63: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 10 threads with 50 CUDA streams per thread.



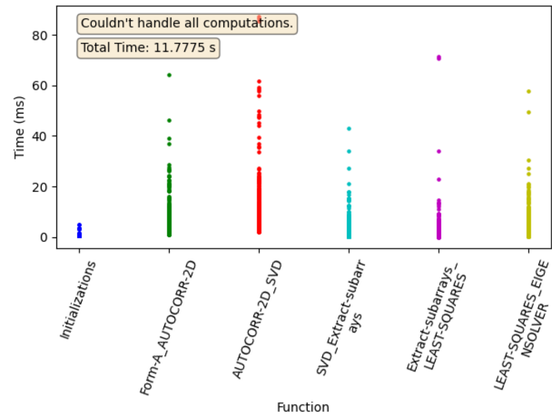
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

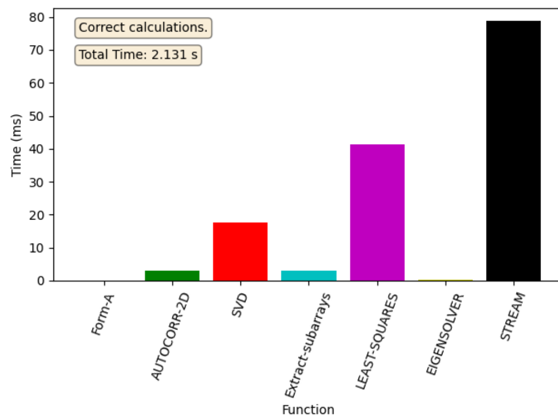


(d) Inter-function mean time

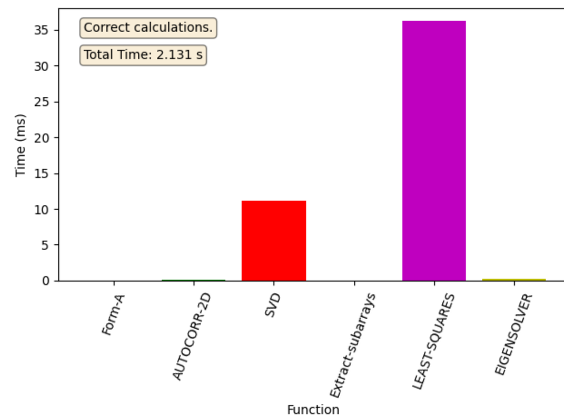
Figure A.64: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 10 threads with 50 CUDA streams per thread.

A.11 Version 3, 20 threads, 25 streams

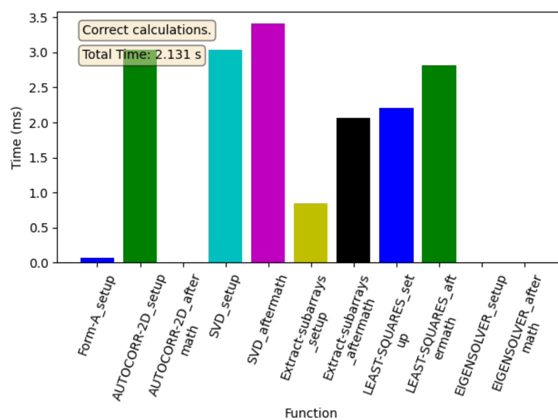
A.11.1 FP32, size 64



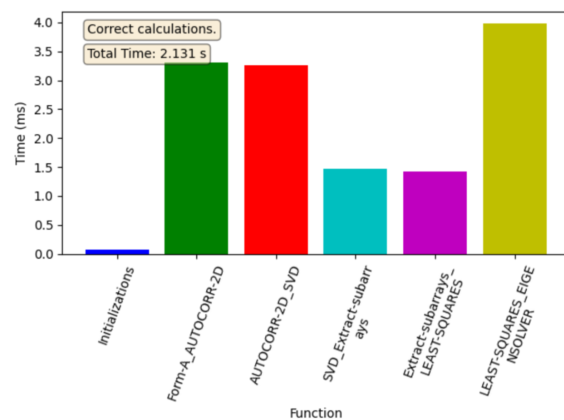
(a) Function mean time



(b) Compute mean time

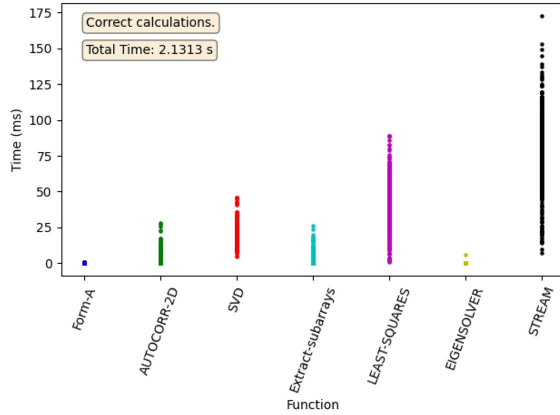


(c) Memory mean time

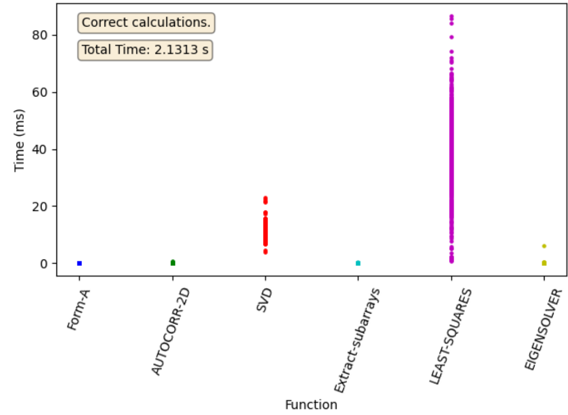


(d) Inter-function mean time

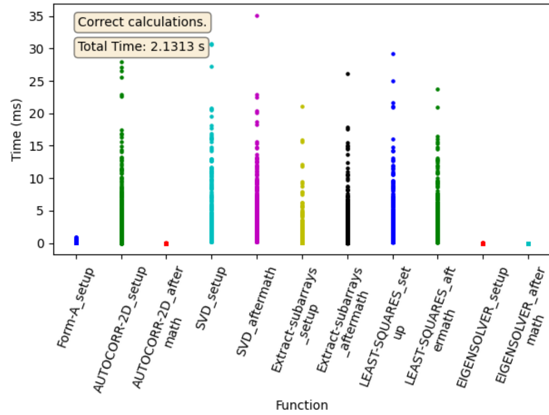
Figure A.65: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 20 threads with 25 CUDA streams per thread.



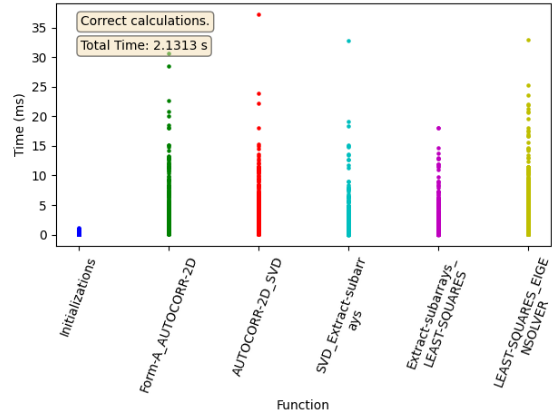
(a) Function mean time



(b) Compute mean time



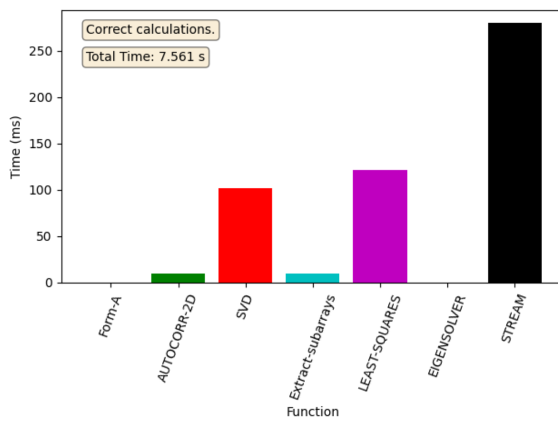
(c) Memory mean time



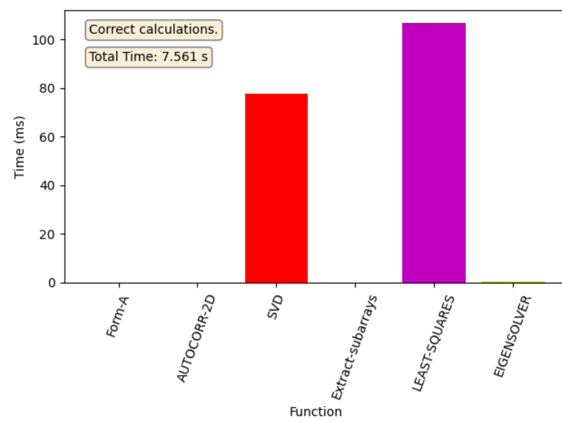
(d) Inter-function mean time

Figure A.66: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 20 threads with 25 CUDA streams per thread.

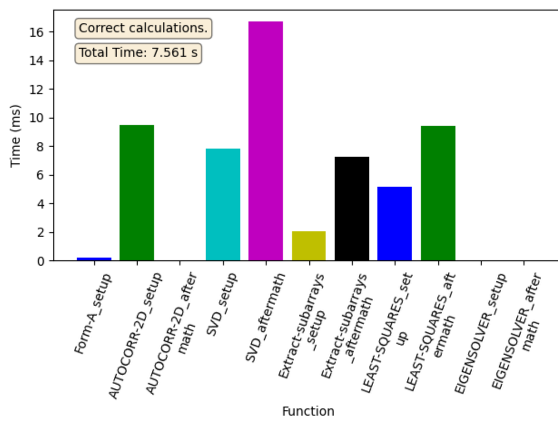
A.11.2 FP32, size 360



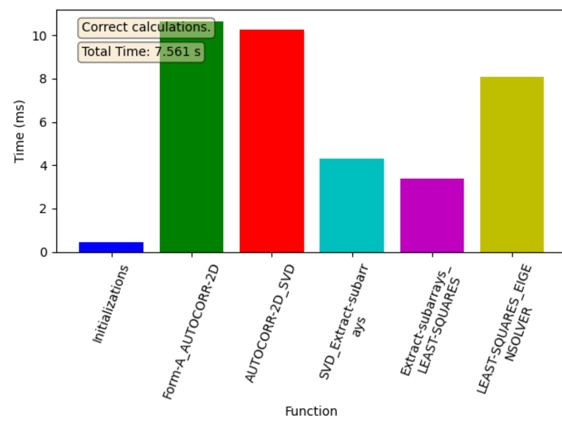
(a) Function mean time



(b) Compute mean time

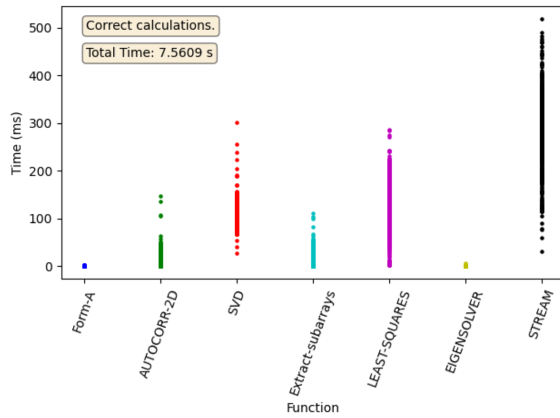


(c) Memory mean time

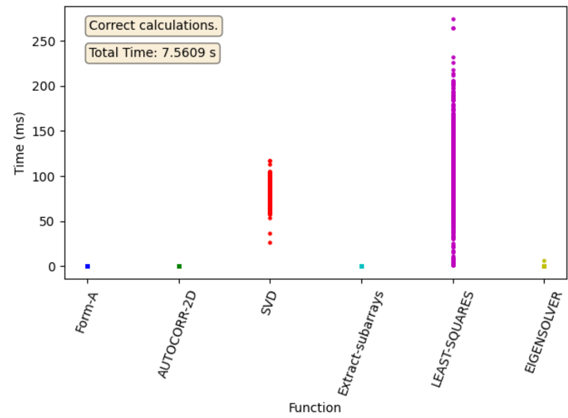


(d) Inter-function mean time

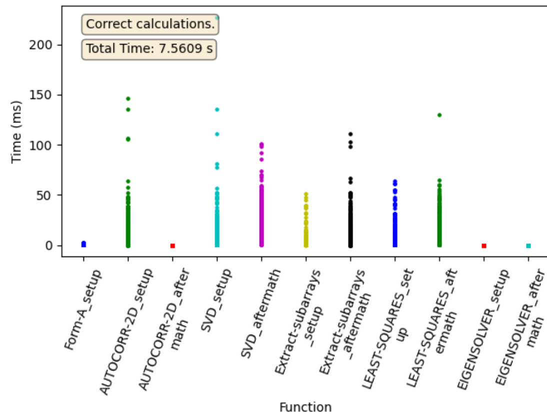
Figure A.67: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 20 threads with 25 CUDA streams per thread.



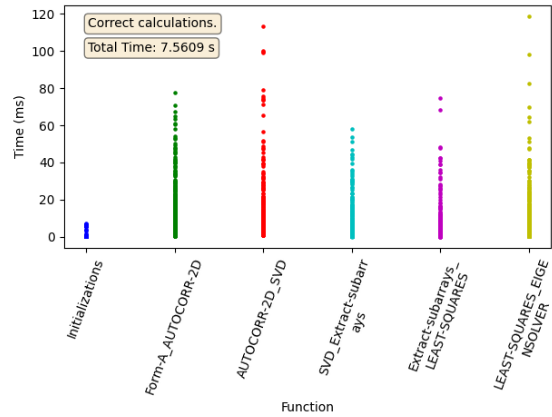
(a) Function mean time



(b) Compute mean time



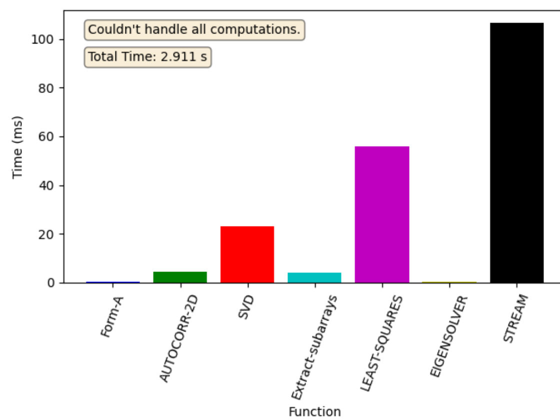
(c) Memory mean time



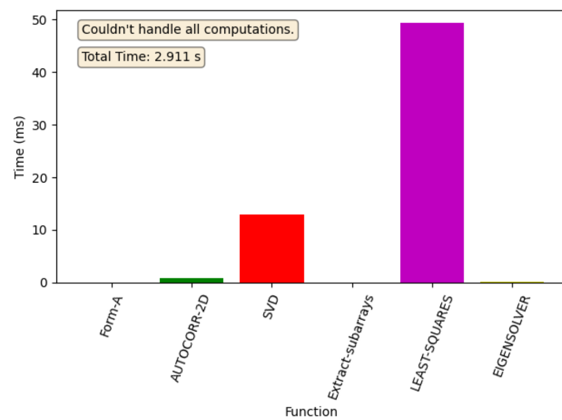
(d) Inter-function mean time

Figure A.68: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 20 threads with 25 CUDA streams per thread.

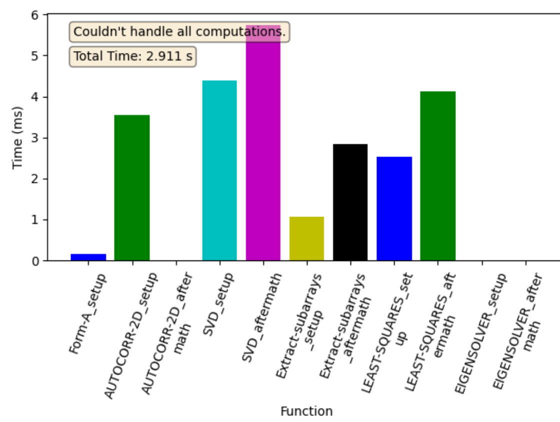
A.11.3 FP64, size 64



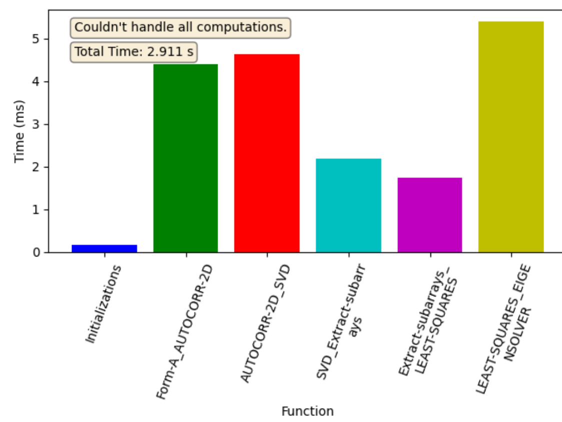
(a) Function mean time



(b) Compute mean time

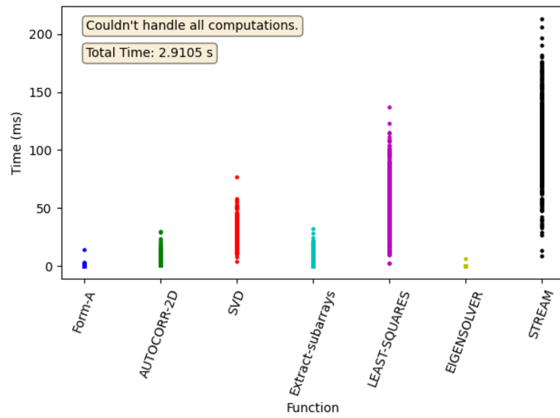


(c) Memory mean time

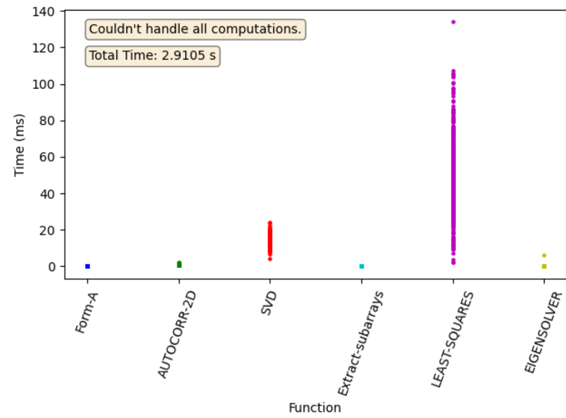


(d) Inter-function mean time

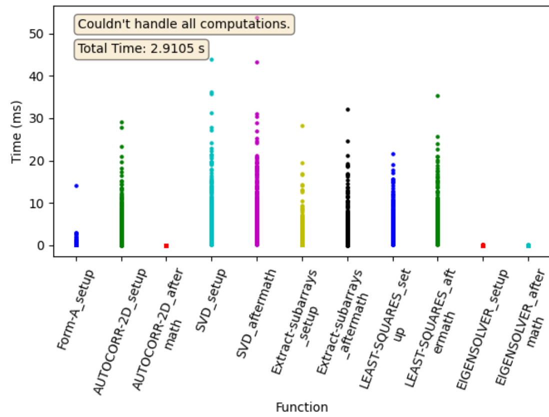
Figure A.69: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 20 threads with 25 CUDA streams per thread.



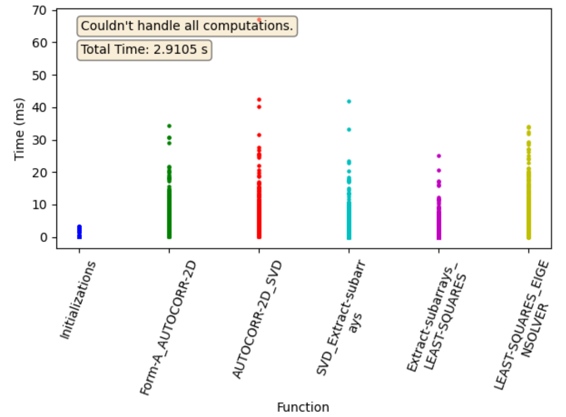
(a) Function mean time



(b) Compute mean time



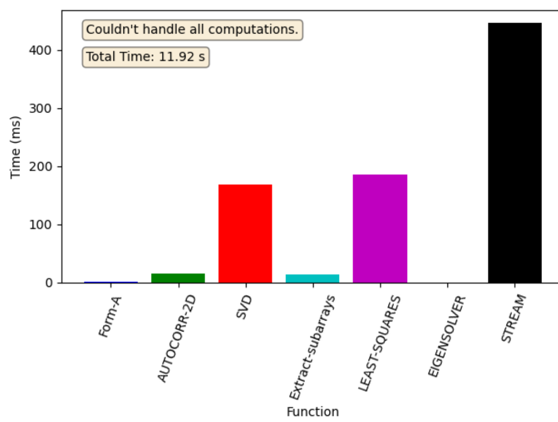
(c) Memory mean time



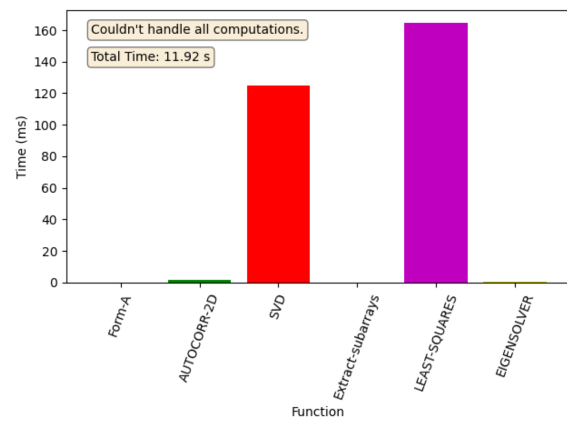
(d) Inter-function mean time

Figure A.70: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 20 threads with 25 CUDA streams per thread.

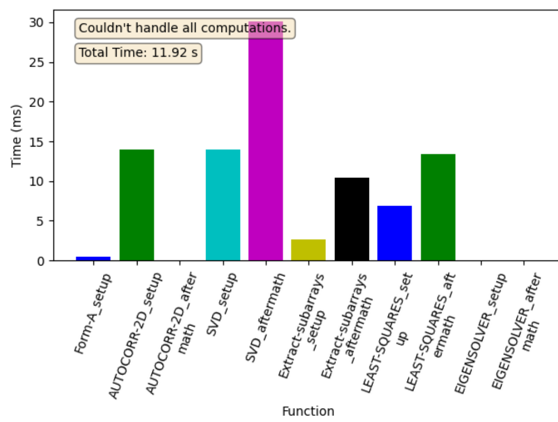
A.11.4 FP64, size 360



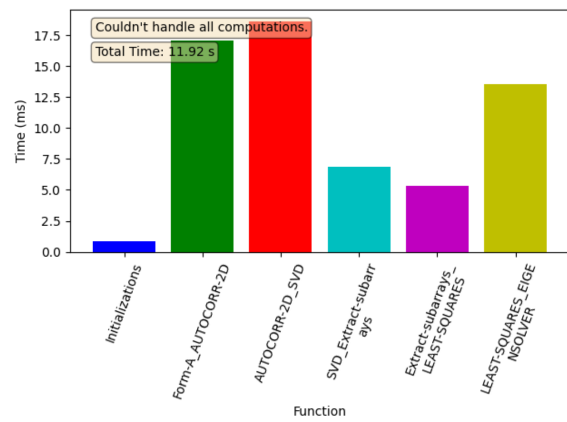
(a) Function mean time



(b) Compute mean time

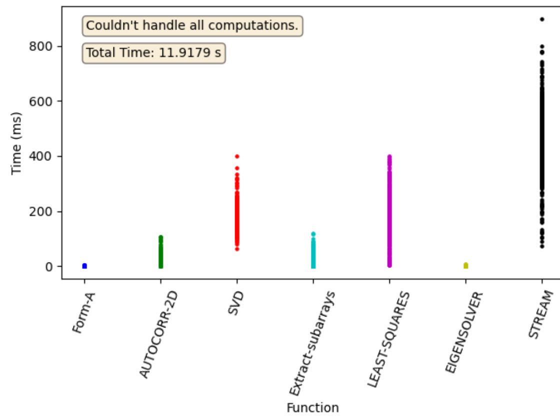


(c) Memory mean time

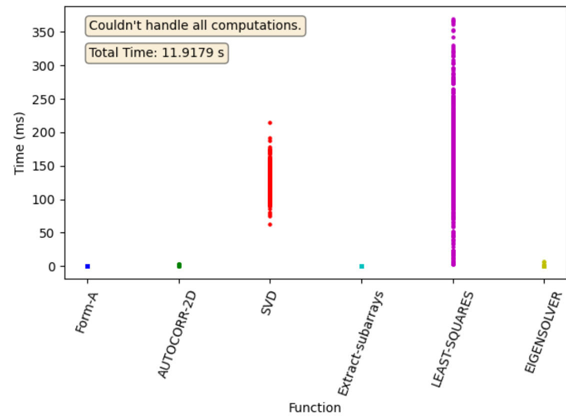


(d) Inter-function mean time

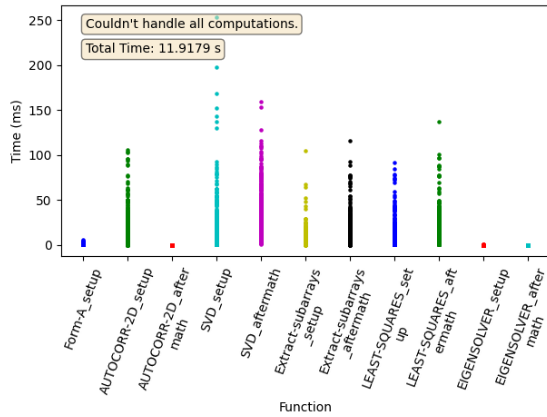
Figure A.71: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 20 threads with 25 CUDA streams per thread.



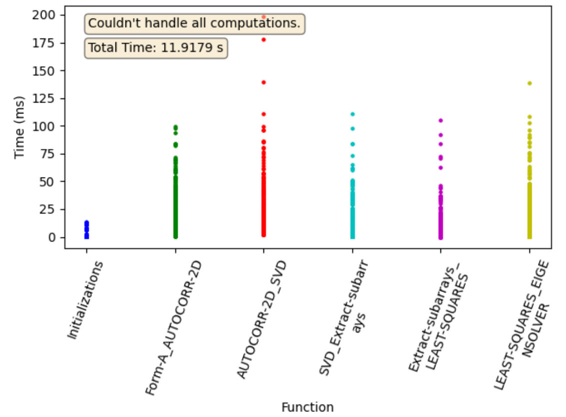
(a) Function mean time



(b) Compute mean time



(c) Memory mean time

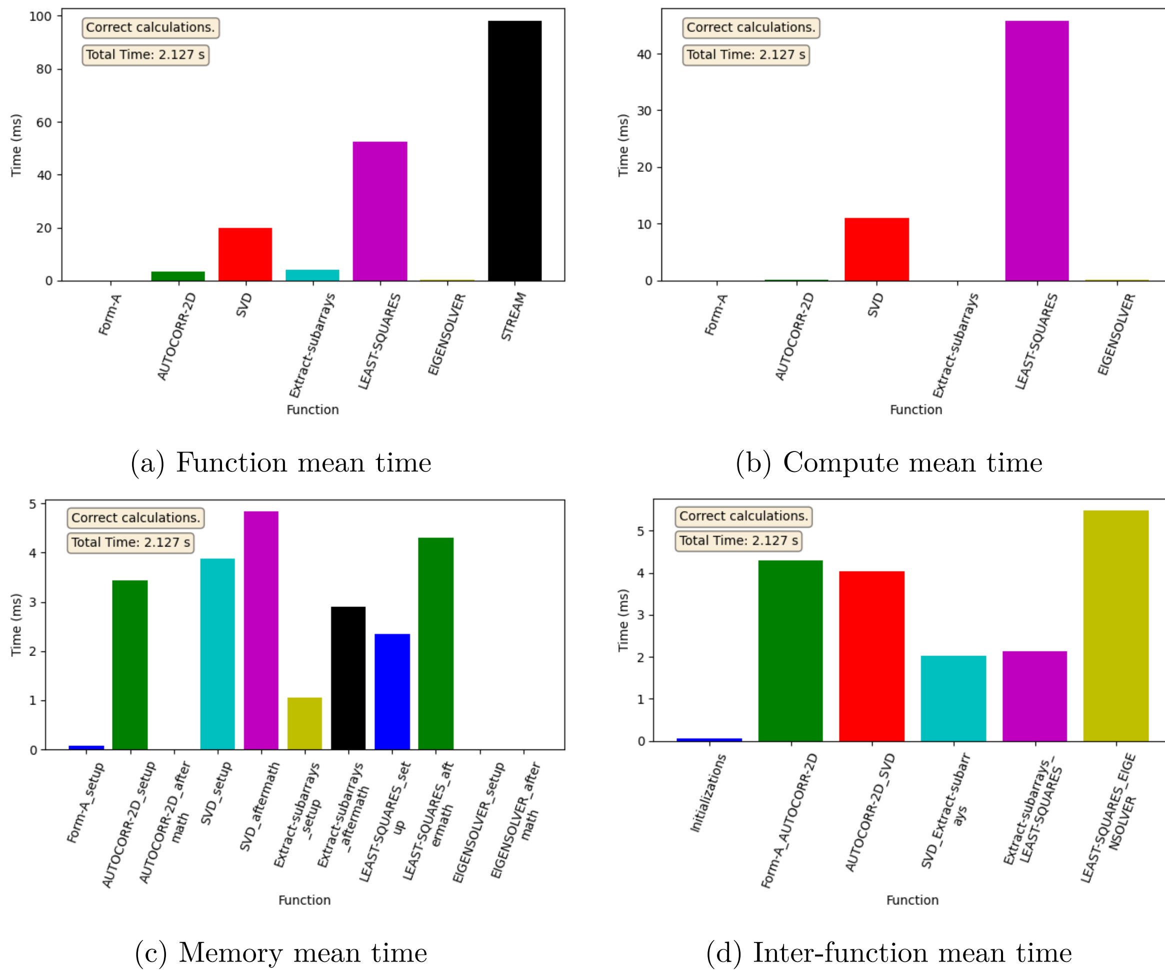


(d) Inter-function mean time

Figure A.72: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 20 threads with 25 CUDA streams per thread.

A.12 Version 3, 25 threads, 20 streams

A.12.1 FP32, size 64



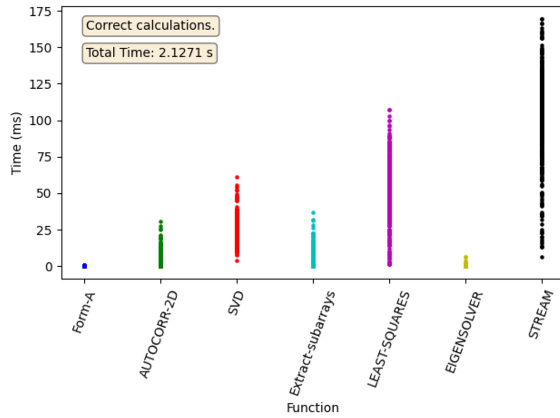
(a) Function mean time

(b) Compute mean time

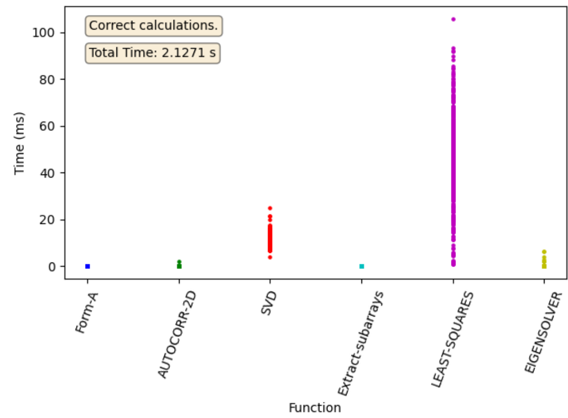
(c) Memory mean time

(d) Inter-function mean time

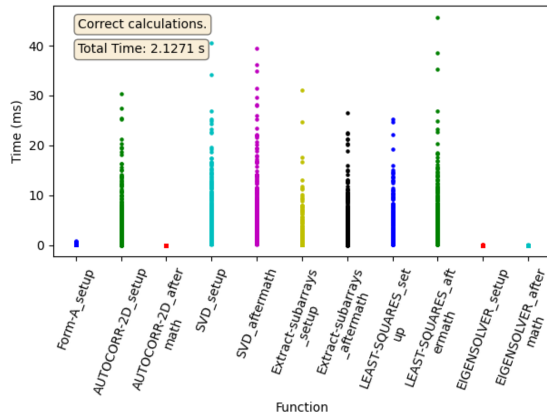
Figure A.73: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 25 threads with 20 CUDA streams per thread.



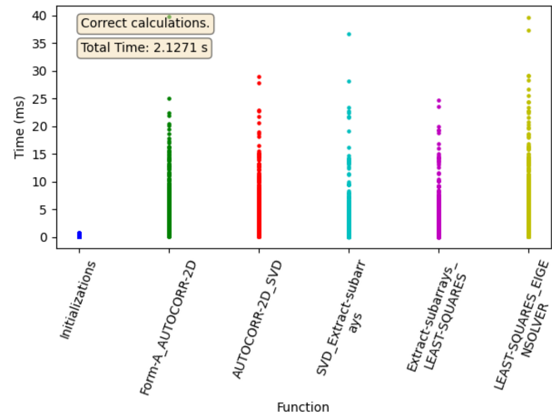
(a) Function mean time



(b) Compute mean time



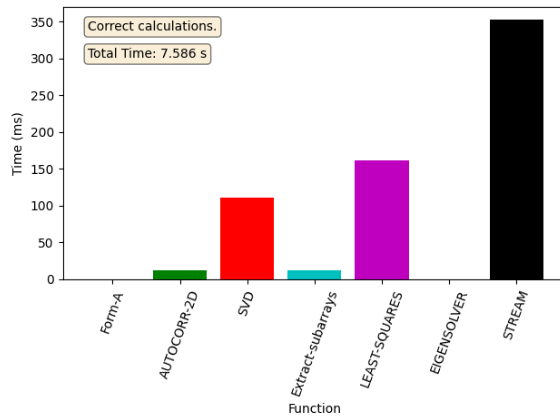
(c) Memory mean time



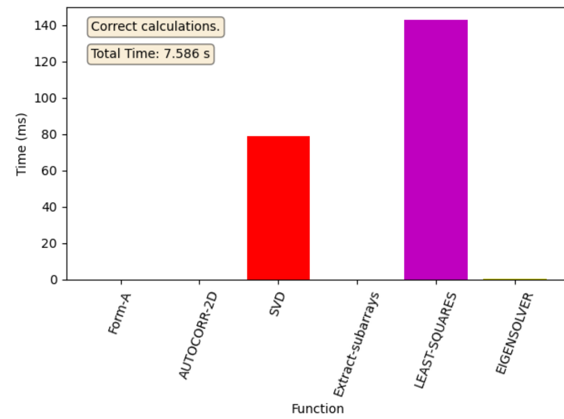
(d) Inter-function mean time

Figure A.74: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 25 threads with 20 CUDA streams per thread.

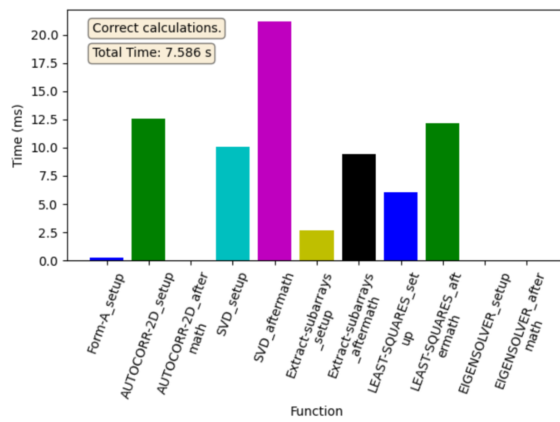
A.12.2 FP32, size 360



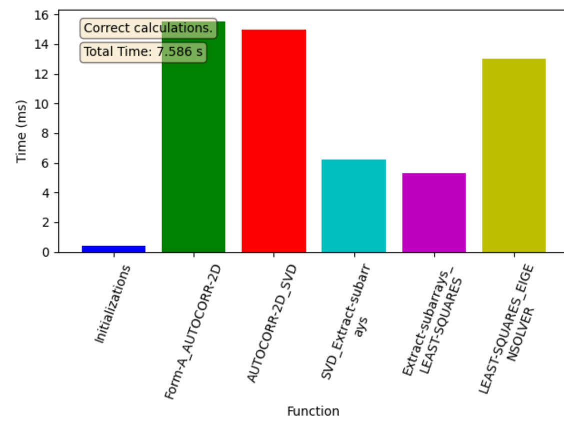
(a) Function mean time



(b) Compute mean time

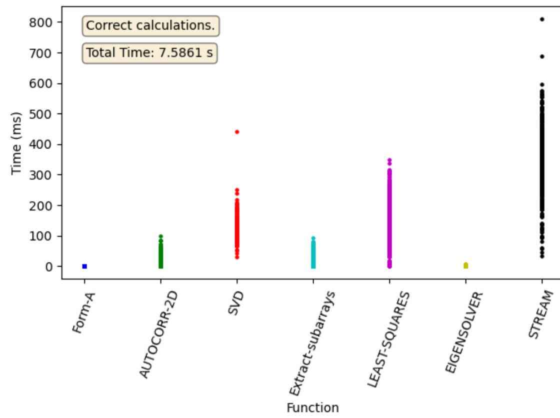


(c) Memory mean time

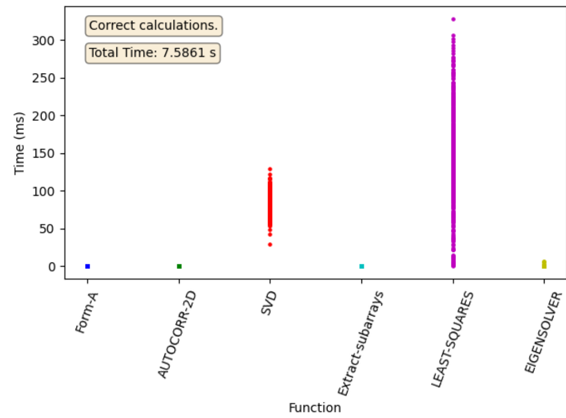


(d) Inter-function mean time

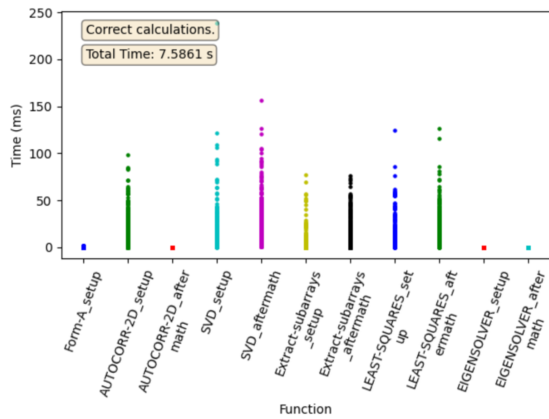
Figure A.75: Mean times for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 25 threads with 20 CUDA streams per thread.



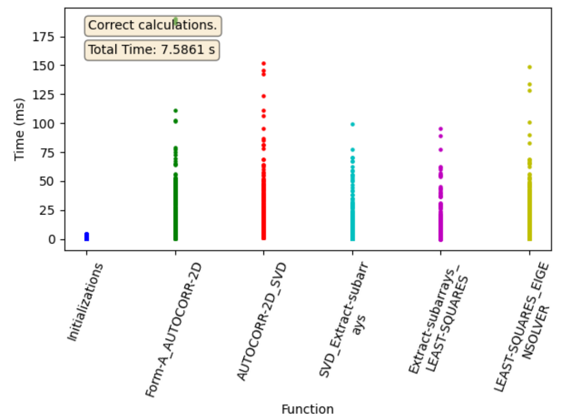
(a) Function mean time



(b) Compute mean time



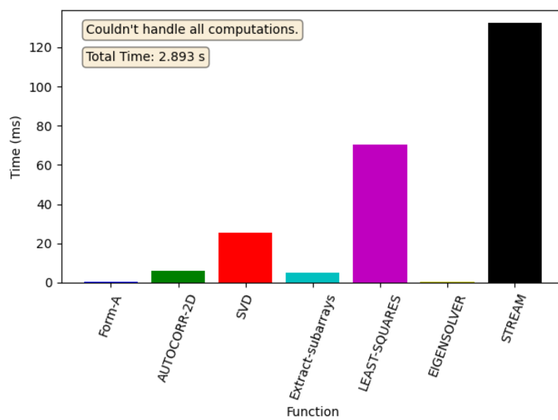
(c) Memory mean time



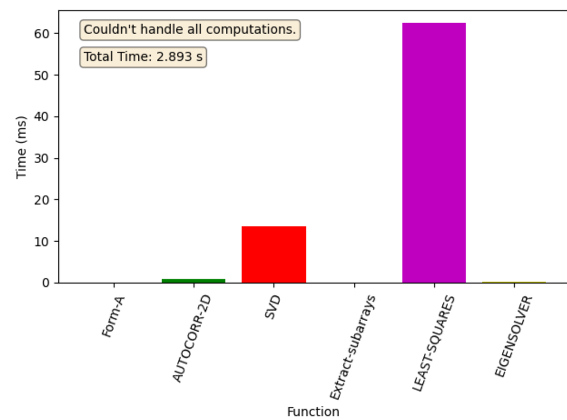
(d) Inter-function mean time

Figure A.76: Scatter plots for FP32 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 25 threads with 20 CUDA streams per thread.

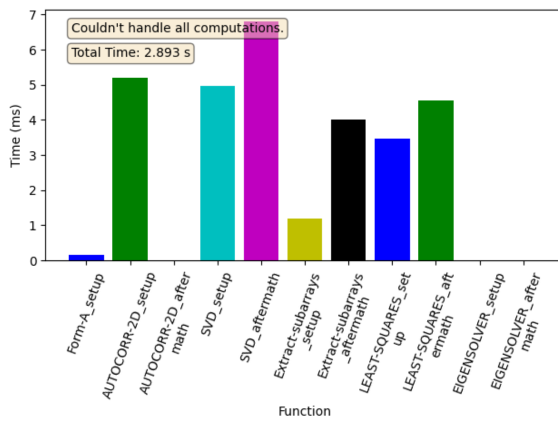
A.12.3 FP64, size 64



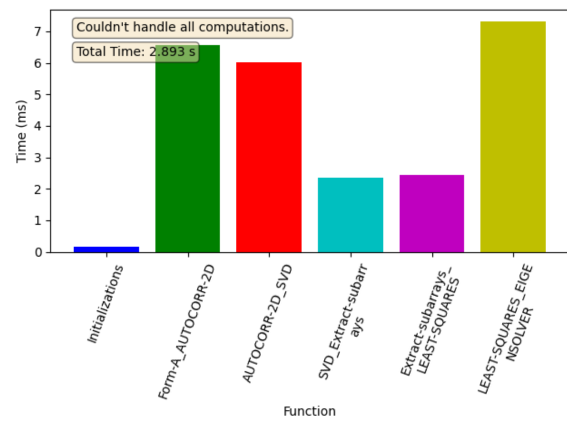
(a) Function mean time



(b) Compute mean time

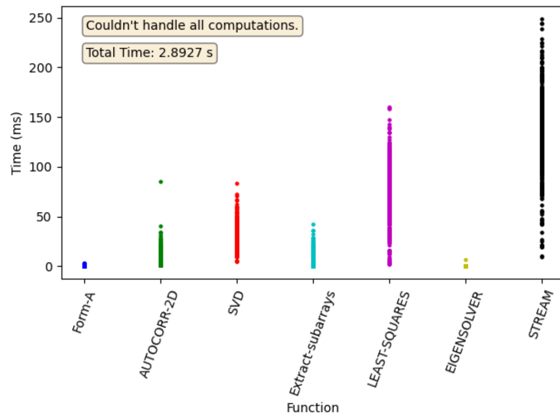


(c) Memory mean time

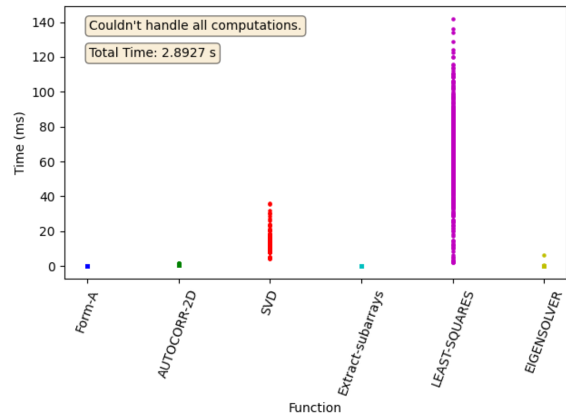


(d) Inter-function mean time

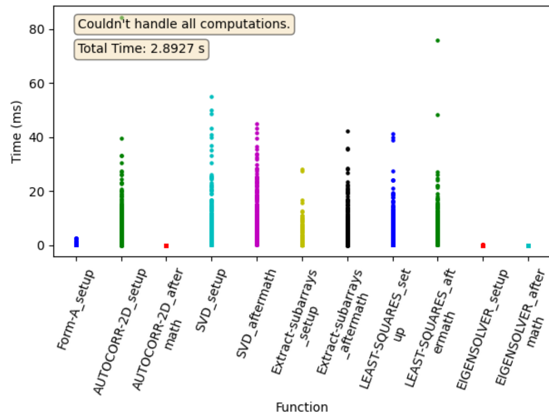
Figure A.77: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 25 threads with 20 CUDA streams per thread.



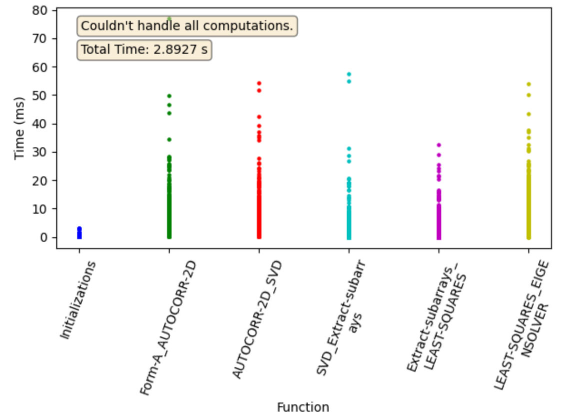
(a) Function mean time



(b) Compute mean time



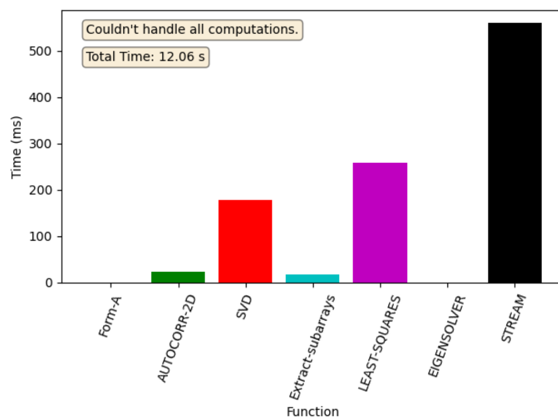
(c) Memory mean time



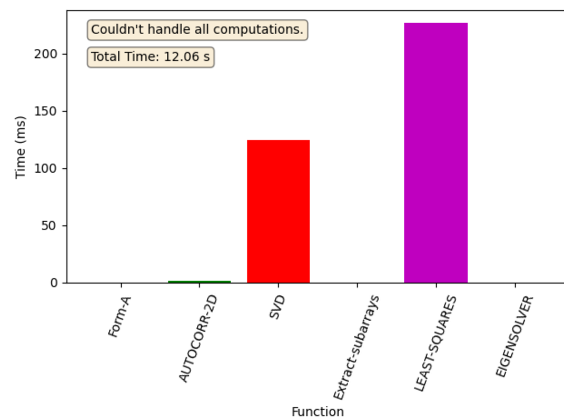
(d) Inter-function mean time

Figure A.78: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 64 elements. Configuration is 25 threads with 20 CUDA streams per thread.

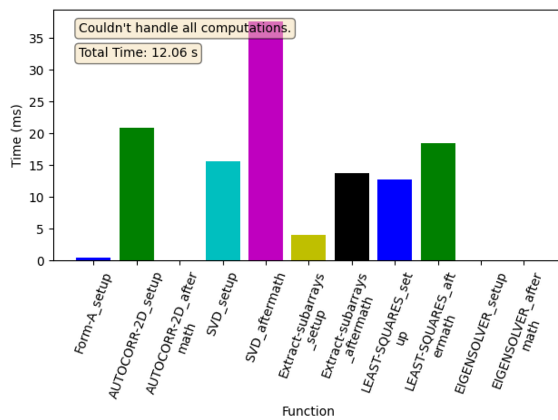
A.12.4 FP64, size 360



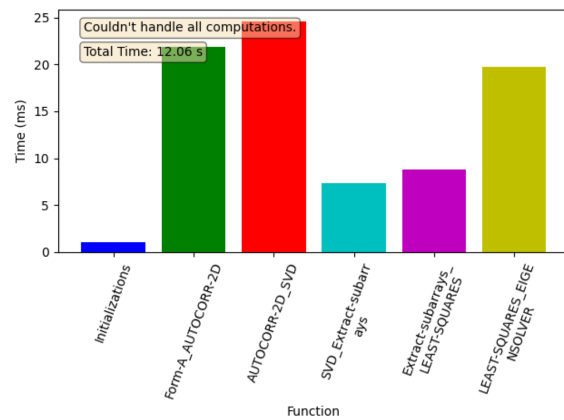
(a) Function mean time



(b) Compute mean time

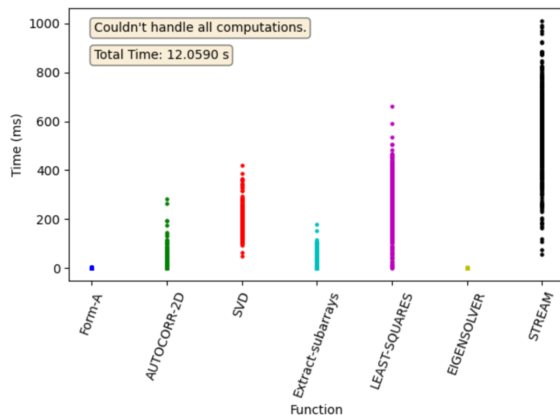


(c) Memory mean time

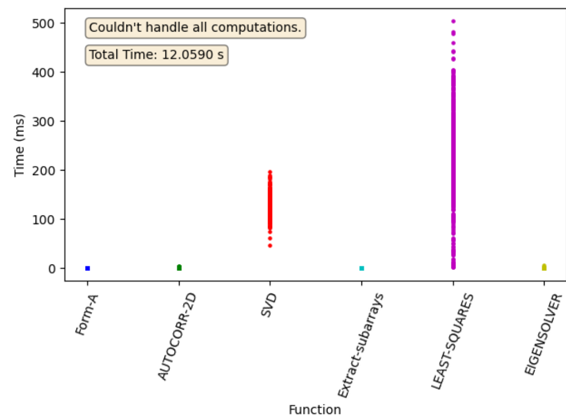


(d) Inter-function mean time

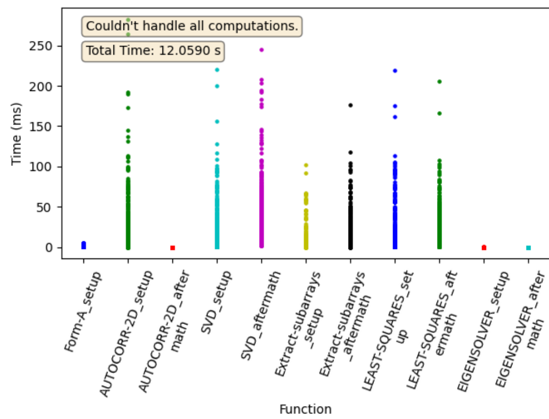
Figure A.79: Mean times for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 25 threads with 20 CUDA streams per thread.



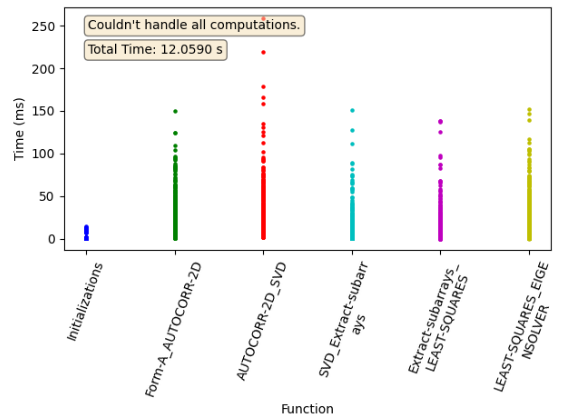
(a) Function mean time



(b) Compute mean time



(c) Memory mean time



(d) Inter-function mean time

Figure A.80: Scatter plots for FP64 using the third version of the algorithm with a covariance matrix size of 360 elements. Configuration is 25 threads with 20 CUDA streams per thread.