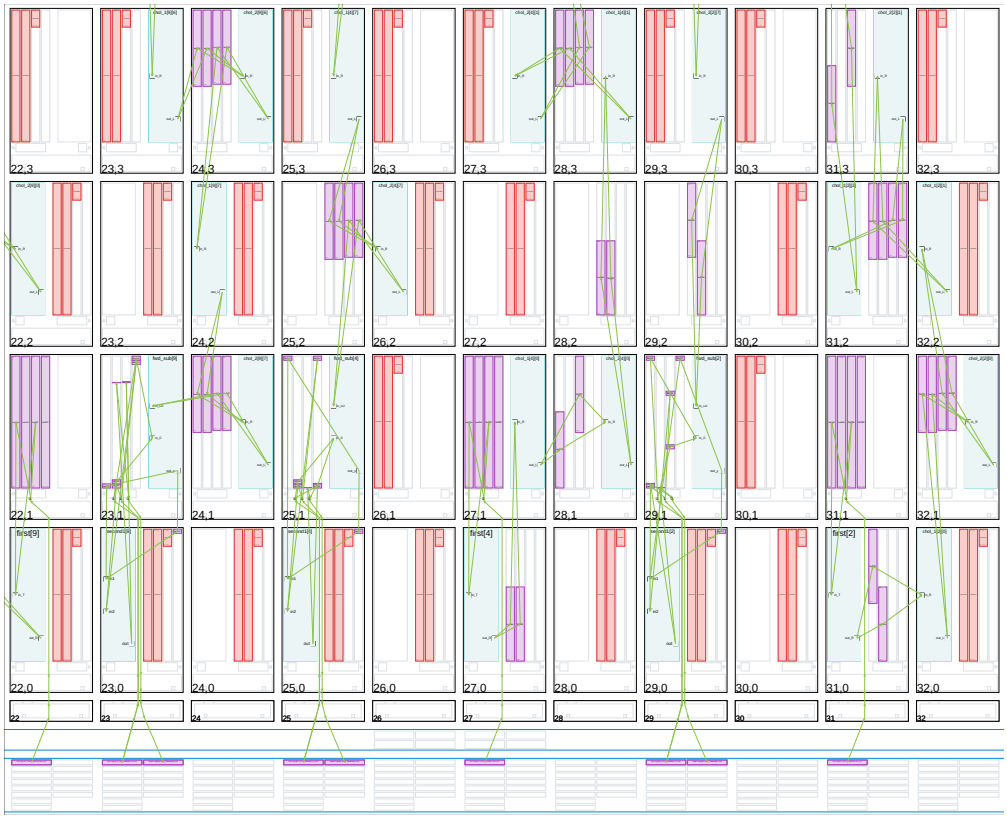
CHALMERS
UNIVERSITY OF TECHNOLOGY



Performance Evaluation of Versal AI Engines for Radar Signal Processing

Master's thesis in Embedded Electronic System Design

Albin Juopperi
Samuel Sjölén

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Master's thesis 2026

Performance Evaluation of Versal AI Engines for Radar Signal Processing

Albin Juopperi
Samuel Sjölén

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2026

Performance Evaluation of Versal AI
Engines for Radar Signal Processing
Albin Juopperi
Samuel Sjölén

Supervisor: Lars Svensson, Microtechnology and Nanoscience
Company advisor: David Wilkins, Saab AB
Examiner: Per Larsson-Edefors, Microtechnology and Nanoscience

Master's Thesis 2026
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The image shows an array artificial intelligence engine tiles. The image is based off a simulation made during this project. The image can be viewed in its entirety in Appendix B.

Typeset in LaTeX
Gothenburg, Sweden 2026

Performance Evaluation of Versal AI Engines for Radar Signal Processing
Albin Juopperi
Samuel Sjölén
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

This thesis aimed to evaluate the artificial intelligence engines found on the Versal VCK 190 evaluation board. To perform the evaluation, a non-homogeneous detector based on the generalised inner product was used. This algorithm was implemented in its entirety in MATLAB and on a graphics processing unit, which was used for comparison. For the Versal board however, only parts of the algorithm were implemented in the design. Further, the design was never run on the Versal board, instead, simulations were used to achieve the results presented in this thesis.

This thesis found no real advantages of using the artificial intelligence engines for the parts of the generalised inner product algorithm that were implemented. For computations of matrices larger than 24×24, the graphics processing unit outperformed the artificial intelligence engines in terms of speed. Although this report did not find any real advantages for using the artificial intelligence engines, further studies are required, especially regarding the power consumption in comparison to the throughput, before any final conclusions can be drawn regarding the artificial intelligence engines.

Keywords: artificial intelligence engines, generalised inner product, non-homogeneous detector, graphics processing unit,

Acknowledgements

We, the authors of this thesis, would like to extend our gratitude to the people who have aided us in our work. We would especially like to thank **Nicklas Wright** for his daily assistance with both technical and practical problems. Also, for his help with improving this report. We would also like to thank **Carolina Weber** for giving us this opportunity and for providing weekly guidance throughout the entire project. We also thank **Patrik Dammert** for sharing his great knowledge of radar theory with us when we felt lost.

Further, we also want to thank our supervisors. Thank you, **David Wilkins**, our company advisor, for helping us achieve a technical height by critically reviewing our work every week. We would also like to thank **Lars Svensson**, our supervisor at Chalmers University of Technology, for being our weekly rubber duck. Gratitude is also sent towards our examiner, **Per Larsson-Edefors**, for providing feedback regarding this thesis.

Lastly, we want to thank **Daniel Wallström** and his team at Saab Surveillance for letting us do our master's thesis with you, and for making this both a highly rewarding and enjoyable experience.

Thank you all!

Albin Juopperi & Samuel Sjölén, Gothenburg, June 2026

Contents

List of Figures

Abbreviations

AGU address generator unit. 14
AIE artificial intelligence engine. xiii, xiv, 2, 3, 11, 13–17, 19–23, 27–31, 37, 39, 40, 44, 45, 47–52, 54–57, IV
AMF adaptive matched filter. 2
API application programming interface. 15, 18, 30, 35
APU application processing unit. 11

CFAR constant false-alarm rate. 1–3, 6, 7, 55
CLB configurable data block. 13
CPI coherent processing interval. xiii, 5, 6
CPU central processing unit. 11, 17, 40, 50, 54, 55
CUDA Compute Unified Device Architecture. 18, 26
CUT cell under test. xiii, 1, 6–8, 23–26, 33, 34, 48, 49, 51

DDRC double data rate memory controller. 12
DMA direct memory access. 14
DSP digital signal processing. 13

ED energy detector. xiv, 3, 7, 8, 20, 37, 38

FPGA Field-Programmable Gate Array. 3

GIP generalised inner product. xiii, xiv, 1–3, 7, 8, 19, 21, 23–26, 30, 33, 37–41, 44, 45, 48–53, 55–57, IV
GPU graphics processing unit. xiv, 3, 8, 17–21, 23, 26, 30, 37, 39, 40, 42, 43, 45, 49–52, 54, 55, 57
GUI graphical user interface. 22

HLS High-level synthesis. 27

IDE integrated development environment. 20
IOU input/output unit. 12
IP intellectual property. 13

LLMs large language models. 56

MM2S memory mapped to stream. 27

NHD non-homogenous detector. xiv, 1, 2, 7, 8, 20, 37, 38
NoC network on chip. 13

OCM on-chip memory. 12

PL programmable logic. xiii, 11, 13, 17, 27–30
PMC platform management controller. 12
PS processing system. xiii, 11, 12, 17, 27–30

RAM random-access memory. 13
RCS radar cross section. 1
RPU real-time processing unit. 12
RTL register transfer level. 27

S2MM stream to memory mapped. 27
SIMD single instruction, multiple data. 14, 17
SINR signal-to-interference-plus-noise ratio. 2
SNR signal-to-noise ratio. 23
SoC system on chip. xiii, 11–13
STAP space-time adaptive processing. 2, 55

VLIW very-long instruction word. 2, 14, 15

1

Introduction

In recent years, the development of military technology has rapidly advanced [1]. As a result of this, the modern battlefield poses many new threats to human life, that includes not only military personnel, but also innocent civilians [2]. If a state is to protect its citizens from attacks from foreign aggressors, the ability to detect and track such attacks is necessary. To do this, competent and reliable radar systems play an important role.

For a radar system to decide whether or not a target has been observed, there are multiple techniques that can be used. The technique known as constant false-alarm rate (CFAR) is a commonly used one. It is defined, as suggested by the name, by the ability to maintain a constant probability of false-alarm rate [3]. This is achieved by using an adaptive threshold, which changes the detection requirements for a data sample based on the surrounding noise [4]. The CFAR detector will be explained further in Section 2.1, but it is important to know that in order to calculate the adaptive threshold of the CFAR detector, an estimated noise covariance matrix is required [5].

The covariance matrix is calculated from the cells surrounding the cell under test (CUT). These surrounding cells are referred to as either secondary data cells or training data. Moreover, the detection method assumes the training data does not contain any targets. If a target were to be located inside the training data, the covariance matrix would no longer be a good estimate of homogenous noise, i.e. noise equally spread over all frequencies. The adaptive threshold would in such a case rise far above a desired level, and targets might not be detected [6]. This is especially bad for targets that were already hard to detect. In real life, this could for example be a swarm of drones, or other objects with small radar cross section (RCS), located close to each other.

By excluding the non-homogeneous noise from the training data, the detection performance of CFAR-based radar systems could be improved. Such an exclusion could be made by utilising a non-homogenous detector (NHD) based on a generalised inner product (GIP) [6]. Just as the name suggests, the purpose of an NHD is to detect non-homogeneous noise. If non-homogeneous noise is detected in a cell, the cell can be excluded from the training data in favour of a cell only containing homogenous noise. The drawback of this method is the vast increase in computational capacity that is required to perform the NHD related calculations. The NHD algorithm requires calculations of GIP values for all cells in a video being made multiple times. The video in this case is a matrix consisting of radar data. This is computationally

expensive as every GIP value is calculated through various matrix operations [6].

By using artificial intelligence engines (AIEs) on the Versal VCK190, the GIP could potentially be implemented in a more efficient way than previously. AIEs consist of multiple AIE tiles, each of which consists of a vector processor optimised for very-long instruction word (VLIW) processing. According to AMD, this architecture is beneficial for signal processing [7]. Furthermore, AIEs could also enable CFAR or similar algorithms to run on edge-devices, which for example could decrease latency in a system. Evaluating AIEs for radar signal processing is therefore of interest and will be the aim of this thesis.

1.1 Related Work

The problem with detecting targets in non-homogeneous noise is well known. Research has been conducted to improve the detection of targets in non-homogeneous noise, and utilising the GIP is one possible solution. This is seen in [6], where an adaptive matched filter (AMF) method is used to evaluate the GIP-based NHD. The study presented an improvement of approximately 2 dB in signal-to-interference-plus-noise ratio (SINR) when using an AMF based on an NHD versus an AMF without an NHD.

Furthermore, as seen in [8], there are multiple algorithms that can be used for performing space-time adaptive processing (STAP) together with a GIP-based NHD. Two of the three tested algorithms in [8] were improved when using the GIP, while the third gave the desired results even without the GIP. What all STAP algorithms and the GIP have in common is the dependency on matrix operations.

Weber and Wright [9] have previously shown the theoretical potential in combining CFAR Feature Plane with neural networks on the Versal VCK190 evaluation board to enhance detection performance. Unfortunately, their design was never implemented on the board. However, the part of their project which focused on target injection in the noise video could be reused in this project.

In contrast to Weber and Wright's work, this report will not focus on the feature plane implementation nor neural networks. Instead, a GIP-based NHD will be implemented on the Versal VCK190 evaluation board. In addition, this project will cover non-homogeneous noise, something Weber and Wright did not do.

1.2 Purpose and Goals

The purpose of this project is to evaluate the performance of AIEs to find out how well suited they are for handling resource-demanding matrix operations required for radar signal processing. The GIP algorithm is an example of a resource-demanding algorithm, which makes it well suited for evaluating the performance of the AIEs. To achieve the purpose, the work is divided into four separate goals.

The first goal of the project is to implement the GIP algorithm as a MATLAB model. The second goal is to calculate a GIP value using the AIEs on the Versal

VCK190 board. The third goal of the project is to implement a GIP algorithm on a graphics processing unit (GPU). Lastly, the fourth goal of the project is to compare the performance of all implementations with each other.

1.3 Delimitations

The project is carried out within a number of delimitations to ensure it is completed on time and achieves the desired goals. First and foremost, as the project is carried out in collaboration with Saab, the hardware used for the project will be chosen and provided by them. Thus, no other board than the Versal AI Core Series FPGA VCK190 board will be used as target hardware. The same argument applies for the GPU. The Quadro RTX 6000 is chosen and provided by Saab and will therefore be used as a benchmark to compare against the Versal board. In addition, only simulated results will be used to evaluate the AIEs; the implementations will not run on actual hardware.

Secondly, all test data will be simulated. The simulated data will not contain any clutter, i.e. unwanted pulse echoes from, for example, the ground. This is because clutter elimination lies outside the scope of the thesis. However, slight modifications, such as target injection, will take place during the course of the project. Further, this test data will only feature one antenna element.

Lastly, there are multiple CFAR detectors that could be used for this project. This thesis will only use the energy detector (ED). As the focus of the project is to evaluate the hardware, a less advanced CFAR algorithm is preferred in order not to unnecessarily waste time.

1.4 Thesis Outline

This thesis aims to evaluate the performance of AIEs through the so-called GIP algorithm. The thesis will thus begin in Chapter 2 by discussing broad mathematical and radar concepts that are central to the thesis. This background chapter will also feature the hardware used in the project, where the Versal board is the main focus.

After the technical background, the methodology of the project will be discussed in Chapter 3. This chapter will touch upon the workflow itself and the development environments. The chapter will also discuss how the achieved results for different implementations were verified, and most importantly, how the testing of each implementation was conducted.

Next up, the design will be described in Chapter 4. The chapter presents block diagrams for the implementations alongside a further explanation of how the GIP algorithm was implemented in general. Design choices are also motivated in this chapter.

Lastly, before we conclude the thesis in Chapter 7, Chapter 5 presents results and Chapter 6 discusses the implications of the results. Besides discussing the results from the project, the discussion chapter also focuses on what future projects can learn from this one.

2

Technical Background

The following chapter will touch upon theoretical concepts which are of importance to the project. Initially, theory related to radar technology will be presented, followed by some sections regarding mathematical algorithms. Lastly, aspects of the hardware itself will be discussed.

2.1 Radar Theory

Radar utilises radio waves to detect and track targets [3]. The technology has been around for more than 100 years, and many advancements have been made to the technology since it was first invented. Fundamentally, the technique works by having a radar unit transmit a radio wave. The wave will, if any object is located in front of the radar unit, reflect off the object and return to the radar unit. By measuring the time between the transmission and the reception of the wave, the distance to the object is easily calculated [3].

However, in reality, radar detection is not always as easy as the prior example makes it out to be. Noise, clutter and unwanted targets make radar detection difficult. To improve the probability of detecting desired targets, signal processing techniques can be used. The following chapter will discuss some techniques that can be used for detection improvement in a radar system. The radar datacube, shown in Figure 2.1, is a central part of the processing techniques discussed in the following chapter.

The dimensions of the cube represent the number of range bins (L), the number of antenna elements (N) and the number of pulses in a coherent processing interval (CPI) (M). Generally for this thesis, only one antenna element is used, thus $N = 1$. This is why the cube is turned into a rectangle in Chapter 4. Further, the CPI is important since it sets the resolutions of our measurements. A higher CPI means more pulses are sent during an interval, which increases the resolution [3]. However, it will also result in more computations required, thus slow the signal processing down. The same argument can be made for the range bins, L and N.

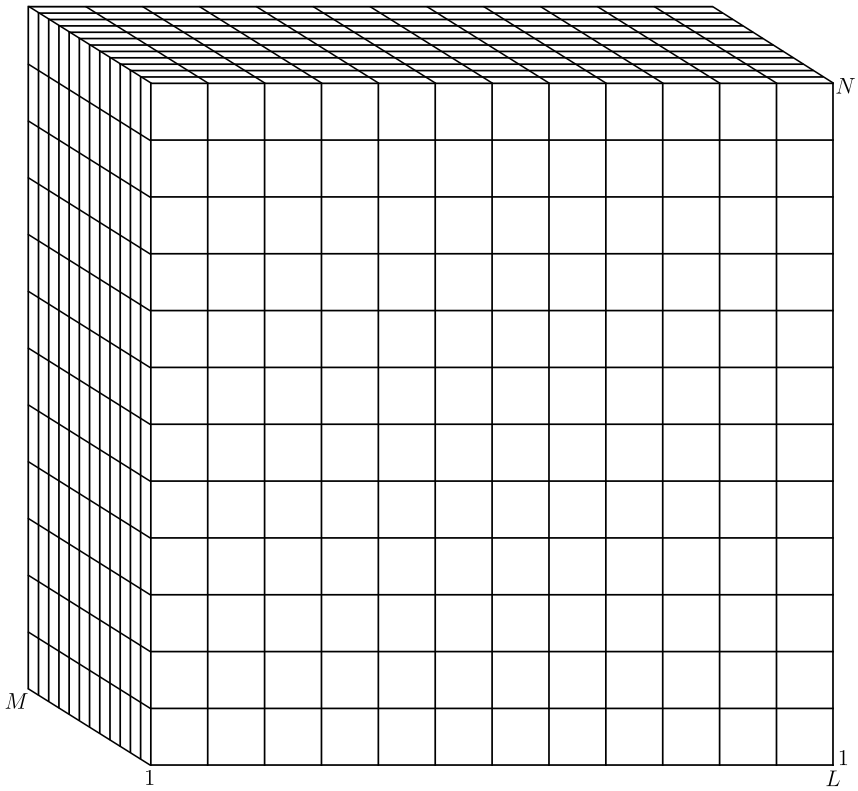


Figure 2.1: An illustration of a radar cube. M represents the number of pulses in a CPI, N the number of antenna elements and L is the number of range bins.

2.1.1 Constant False-Alarm Rate

Radar detection functions by comparing the returned signal to a threshold value. In the simplest case, the threshold is a constant value, and if the signal reaches above that value, it is labelled as a target. To measure the functionality of a detector two metrics are used: probability of detection P_d and probability of false-alarm P_{fa}. The aim is to maximise P_d and keep P_{fa} beneath a desired level. However, non-homogeneous noise, i.e. noise that is not equally distributed over all frequencies, has a tendency to cause false detections. To guarantee that the P_{fa} remains constant, CFAR can be used. A constant false alarm rate is beneficial because it ensures reliable and consistent detector performance, regardless of changes in the noise environment [5].

CFAR works by analysing the surrounding noise and computing a threshold [10]. This is accomplished by utilising training cells around what this thesis calls the CUT. In accordance with the Reed-Mallett-Brennan rule [11], the number of training cells K have to be

$$K \geq 2MN, \tag{2.1}$$

with M being the number of pulses within a CPI, and N the number of antenna elements, as seen in Figure 2.1. MN training cells are selected symmetrically on each side of the CUT [6]. Normally, a number of guard cells are inserted between the training cells and the CUT to avoid a single target lapping into the training data

[12]. If the target were to be inside the training data, the detection performance would drastically decrease. The training cells are then concatenated, creating a $2MN \times MN$ matrix. This matrix of training data is denoted $\boldsymbol{Z}$.

2.1.2 Energy Detector

The energy detector (ED) is a relatively simple detector. It only uses the CUT and the noise covariance matrix to calculate the detection statistic t, which is compared against a constant threshold η [5]. From here on, the CUT will be denoted $\boldsymbol{z} \in \mathbb{C}^{MN\times 1}$, and the estimated covariance matrix will be denoted $\boldsymbol{S}$. To calculate $\boldsymbol{S}$, the training data $\boldsymbol{Z}$ is multiplied with its own Hermitian.

$$\boldsymbol{S} = \boldsymbol{Z}\boldsymbol{Z}^H \tag{2.2}$$

With $\boldsymbol{S}$ calculated, the detection statistic t can then be calculated for the specific $\boldsymbol{z}_i$, where i = 1, 2,..., L. L is the maximum number of range bins.

$$t_i = \boldsymbol{z}_i^H \boldsymbol{S}_i^{-1} \boldsymbol{z}_i \tag{2.3}$$

As the detection statistic has been calculated for $\boldsymbol{z}_i$, a hypothesis test will be conducted. Hypothesis H_1 is defined such that if the computed detection statistic is larger than a constant threshold η, a target has been detected. Hypothesis H_0 is defined as the opposite; if the value of the detection statistic is lower than the threshold, no target is detected. The constant η is to be chosen so that a desired P_{fa} value is achieved.

$$t \underset{H_0}{\overset{H_1}{\gtrless}} \eta \tag{2.4}$$

2.1.3 Non-Homogeneous Detector

CFAR detectors will have difficulties detecting multiple targets that are located close to each other. The reason for this is that contamination of training cells will occur. Although the guard cells offer some protection, their main purpose is to avoid a single target lapping into adjacent cells [12]. To counteract the contamination of training data caused by multiple adjacent targets, a non-homogenous detector (NHD) can be used [6]. In this thesis the NHD will be based on a generalised inner product (GIP), which will be thoroughly explained in this section.

The algorithm is similar to the one used for the ED in Section 2.1.2. First and foremost, a CUT, $\boldsymbol{z}_i$, is chosen. However, instead of directly calculating a threshold value for the CUT, a GIP value will first be calculated for each of the surrounding cells. In total, $4M$ GIP values are calculated for each CUT, which means there are $2M$ values on each side of the CUT. The GIP value is denoted P_i, where $i = 1, 2, ..., 4M$. For each of these specific cells $\boldsymbol{x}_i$, an estimated covariance matrix is required. To calculate the estimated covariance matrix, $2M$ secondary cells are

required similarly to (2.2). However, the training data is selected symmetrically around $\boldsymbol{x}_i$, not $\boldsymbol{z}_i$, and the matrix containing the training data is denoted $\boldsymbol{X}$. Thus, $\hat{\boldsymbol{R}}_{GIP}$ is calculated accordingly to [6].

$$\hat{\boldsymbol{R}}_{GIP} = \frac{1}{K}\boldsymbol{X}\boldsymbol{X}^H \tag{2.5}$$

Further, to calculate P_i, an equation similar to (2.3) is used. The difference is that the energy detector uses the CUT value $\boldsymbol{z}$. However, to calculate P the current GIP-cell, $\boldsymbol{x}$ is instead used [6].

$$P = \boldsymbol{x}^{\boldsymbol{H}}\hat{\boldsymbol{R}}_{\boldsymbol{GIP}}^{-\boldsymbol{1}}\boldsymbol{x} \tag{2.6}$$

With all $4M$ GIP values calculated, the next step of the process is to apply the ED to calculate detection statistics. This step will proceed as explained in Section 2.1.2, but in order to form the covariance matrix $\hat{R}$, the GIP values are compared to the expected value [6].

$$E(P) = \frac{M}{1 - \frac{K}{M}} \tag{2.7}$$

The $2MN$ GIP values closest to $E(P)$ will be used to calculate the covariance matrix $\hat{\boldsymbol{R}}$. By doing so, targets will most likely be excluded from the training data.

An NHD based ED is more computationally complex than a regular ED due to the addition of the GIP algorithm. However, the GIP algorithm causes the complexity to increase at a higher rate compared with a regular ED. The equations used for both the ED, (2.2) and (2.3), and the GIP algorithm, (2.5) and (2.6), are of similar complexity. For every range bin added to the noise video, the complexity increases linearly for both the ED and the GIP algorithm. However, the rate of complexity increase is much higher for the GIP, as another $4MN$ computations are required for every added range bin, whilst the ED only requires one.

2.2 Matrix Inversion Algorithms

The following section will explain the inversion algorithm used in this project. LU decomposition will be explained first alongside the Doolittle method, which is used for the GPU implementation. Next up, Cholesky decomposition will be explained and after that, the Schur complement.

2.2.1 LU Decomposition

A square matrix $\boldsymbol{A}$ can be divided into two different matrices, $\boldsymbol{L}$ and $\boldsymbol{U}$.

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U} \tag{2.8}$$

As the name suggests, these two matrices have values in the lower or upper part of their respective matrices.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{2.9}$$

By dividing the matrix into L and U, the number of operations needed for a linear equation is decreased. This decrease happens because the $\boldsymbol{L}$ linear equation can be calculated using forward substitution and $\boldsymbol{U}$ using backward substitution.

For solving the equation

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{2.10}$$

the first part becomes

$$\boldsymbol{Ly} = \boldsymbol{b} \tag{2.11}$$

which is solved using the following forward substitution

$$\begin{aligned} y_1 &= \frac{b_1}{l_{11}} \\ y_i &= \frac{1}{l_{ii}} \left[b_i - \sum_{y=1}^{i-1} l_{ij} y_j \right] \end{aligned} \tag{2.12}$$

where i iterates through the number of rows in $\boldsymbol{y}$. Then the second part becomes

$$\boldsymbol{Ux} = \boldsymbol{y} \tag{2.13}$$

which uses the $\boldsymbol{y}$ from the first part and is solved using the following backward substitution with N being the number of rows in $\boldsymbol{x}$.

$$\begin{aligned} x_N &= \frac{y_N}{u_{NN}} \\ x_i &= \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{N} u_{ij} x_j \right] \qquad i = N-1, N-2, ..., 1 \end{aligned} \tag{2.14}$$

After these two substitutions $\boldsymbol{x}$ is the solution to (2.10) [13]. However, for a matrix inversion, the vectors in the linear equation are replaced with matrices.

$$\begin{gathered} \boldsymbol{AA}^{-1} = \boldsymbol{I} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \widetilde{a}_{11} & \widetilde{a}_{12} & \widetilde{a}_{13} \\ \widetilde{a}_{21} & \widetilde{a}_{22} & \widetilde{a}_{23} \\ \widetilde{a}_{31} & \widetilde{a}_{32} & \widetilde{a}_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{gathered} \tag{2.15}$$

The problem with having matrices instead of vectors is solved by splitting the $\boldsymbol{A^{-1}}$ and $\boldsymbol{I}$ into vectors which generates the vector linear equations.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \widetilde{a}_{11} \\ \widetilde{a}_{21} \\ \widetilde{a}_{31} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \widetilde{a}_{12} \\ \widetilde{a}_{22} \\ \widetilde{a}_{32} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{2.16}$$
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \widetilde{a}_{13} \\ \widetilde{a}_{23} \\ \widetilde{a}_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

This splitting can now be done for $\boldsymbol{L}$ and $\boldsymbol{U}$. When both the $\boldsymbol{L}^{-1}$ and $\boldsymbol{U}^{-1}$ have been calculated they can be multiplied together forming $\boldsymbol{A}^{-1}$.

The Doolittle method is a way of decreasing the amount of computations for the LU-decomposition. For a square matrix the following equations are used for calculating the cells for $\boldsymbol{L}$ and $\boldsymbol{U}$ shown in (2.9)

$$\begin{aligned} u_{i1} &= a_{i1} \\ l_{i1} &= \frac{a_{i1}}{a_{11}} \\ u_{i2} &= a_{i2} - a_{i1}l_{21} \\ l_{i2} &= \frac{u_{i2}}{u_{22}} \\ u_{i3} &= a_{i3} - u_{i1}l_{31} - u_{i2}l_{32} \\ l_{i3} &= \frac{u_{i3}}{u_{33}} \end{aligned} \tag{2.17}$$

these equations calculate a whole matrix for both $\boldsymbol{L}$ and $\boldsymbol{U}$ but only the corresponding values to form the matrices in (2.9) are used [14].

2.2.2 Cholesky Decomposition

Cholesky decomposition is an even more efficient method to solve systems of linear equations compared to LU decomposition. Cholesky decomposition is roughly twice as efficient as LU decomposition. However, the Cholesky decomposition only works for matrices that are Hermitian and positive-definite. A square-shaped matrix has a Cholesky decomposition if it can be written as

$$A = LL^T \tag{2.18}$$

where L is a lower triangular matrix

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \tag{2.19}$$

and L^T is the conjugate transpose of the lower triangular matrix [15].

$$L^T = \begin{bmatrix} L_{11} & L_{12} & L_{13} \\ 0 & L_{22} & L_{23} \\ 0 & 0 & L_{33} \end{bmatrix} \tag{2.20}$$

To calculate the elements of the lower triangular matrices, the equations (2.21) and (2.22) are defined as

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-i} L_{j,k}^* L_{i,k}} \tag{2.21}$$

$$L_{i,j} = \frac{1}{L_{j,j}}(A_{i,j} - \sum_{k=1}^{j-1} L_{j,k}^* L_{i,k}) \; for \; i > j \tag{2.22}$$

where $*$ symbolises the complex conjugate. The inverse of A, A^{-1}, can be calculated from here by solving the system of linear equations.

2.2.3 Schur Complement

Another method that can be used to calculate a matrix inversion is the Schur Complement which is used for partitioning the matrix inversion [16]. The method assumes a partitioned matrix Q.

$$Q = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{2.23}$$

Each partition must be of size $(p \times p)$, $(p \times q)$, $(q \times p)$ and $(q \times q)$. If partition A is invertible, then the inverse of Q, Q^{-1}, is easily calculated according to (2.24).

$$Q^{-1} = \begin{bmatrix} A^{-1} + A^{-1}BS^{-1}CA^{-1} & -A^{-1}BS^{-1} \\ -S^{-1}CA^{-1} & S^{-1} \end{bmatrix} \tag{2.24}$$

The letter S represents the so-called Schur Complement and is calculated as shown by (2.25).

$$S = D - BA^{-1}C. \tag{2.25}$$

2.3 AMD Versal AI Core Series VCK190

The Versal VCK190 is an evaluation kit developed by AMD that uses their Versal adaptive system on chip (SoC) [17]. While the SoC consists of many parts, the three most significant ones to this project is the programmable logic (PL), the processing system (PS) and the AIE. As shown in Figure 2.2, the PS part is made up of six parts, the central processing unit (CPU), the application processing unit (APU),

the real-time processing unit (RPU), the L2 cahce, the on-chip memory (OCM) and the input/output unit (IOU) [18] [19]. All these parts are, as shown, interconnected and can communicate directly with each other. Right next to the PS, the high speed I/O, XPIO, is found, as well as the double data rate memory controller (DDRC) and platform management controller (PMC) [18] [20].

Table 2.1: Specifications of Versal VCK190 SoC [21].

Specifications	
AI Engines	400
DSP Engines	1,968
LUTs	899,840
Power	180 W

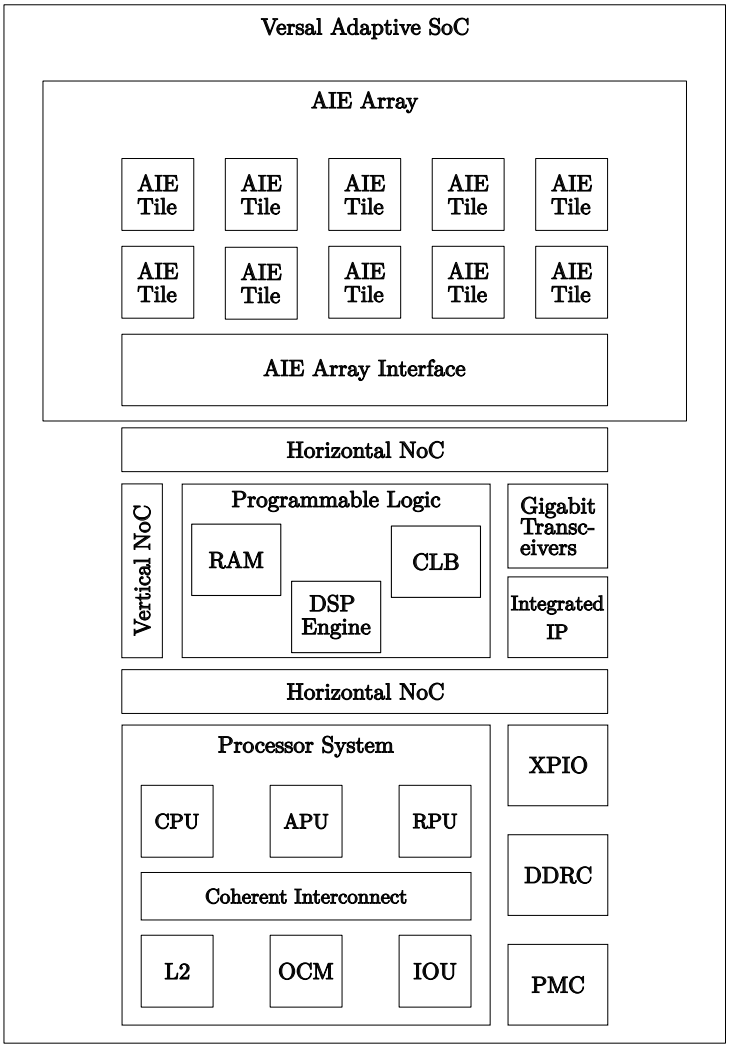


Figure 2.2: Block diagram of the Versal Adaptive SoC, showing the layout of the board as a whole and how the they different parts are interconnected. The figure was inspired by images in [18].

Figure 2.2 also shows that the PL part is made up of three different parts, a random-access memory (RAM), a digital signal processing (DSP) engine and a configurable data block (CLB) [18] [19]. Right next the PL, transceivers and the integrated intellectual property (IP) is found. The NoC is used to transfer data in either horizontal or vertical directions, allowing for all three parts to communicate with each other. The AIE part, which is also shown in Figure 2.2, only consists of AIE tiles and the interface array. These will be further discussed in Section 2.3.1. Further, a number of core parts and the amount available on the board are listed in Table 2.1.

2.3.1 Artificial Intelligence Engine

As Figure 2.2 shows, all AIEs are located in the AIE array. An AIE array, as shown by Figure 2.3, mainly consists of AIE tiles and interface tiles. There are two types of interface tiles: one used to communicate directly with the PL, and one used to communicate via the NoC. These tiles also contain AIEs. Furthermore, there is also one configuration tile per array, which is used as a clock generator among other things. This means an AIE array acts as a single clock domain.

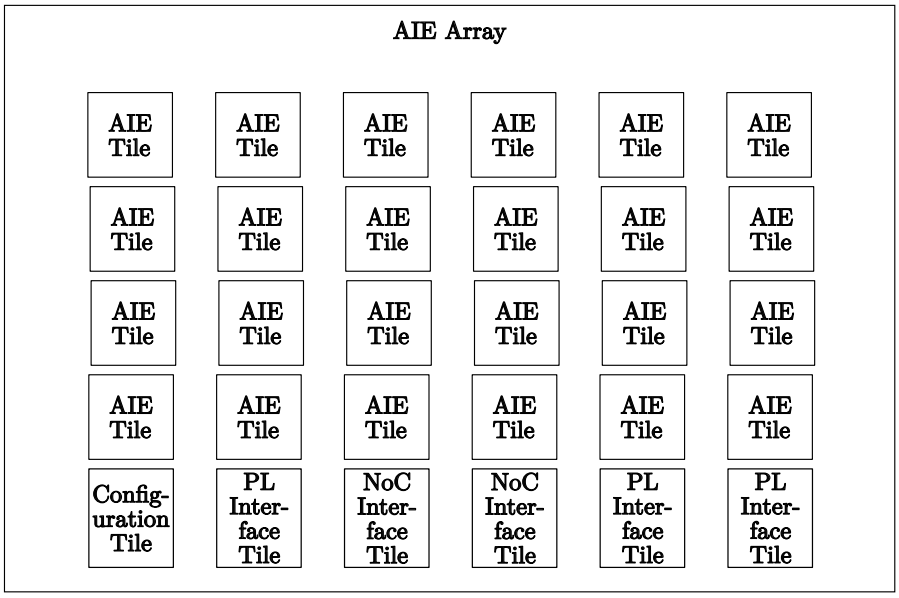


Figure 2.3: Block diagram showing an example of an AIE array. The figure was inspired by images in [18].

An AIE tile consists of a memory module and an AIE. The memory module is connected to an AXI4 interconnect, which links it to the other memory modules [18]. As seen in Table 2.1, there are a total of 400 of these tiles available on the SoC. Figure 2.4 shows a block diagram which illustrates the layout of an AIE tile. The tiles can transfer data in three different ways. The black arrows in Figure 2.4 represent the AXI4 stream, which can be used for external blocks to read and write on the AIE tile memory. The PL is an example of such a block. The blue arrows represent the "tile memory access", which allows neighbouring AIE tiles to access each other's memory.

Lastly, the green arrows represent a cascade stream, which allows streaming directly into the AIE tile memory. As seen, only the AXI4 interconnections do not have access to the memory directly, but it is also the only one that is not limited to the neighbouring tiles [18]. Each tile has a limited to 16 kB of program memory. Further, each tile can only utilise two inputs and two outputs simultaneously.

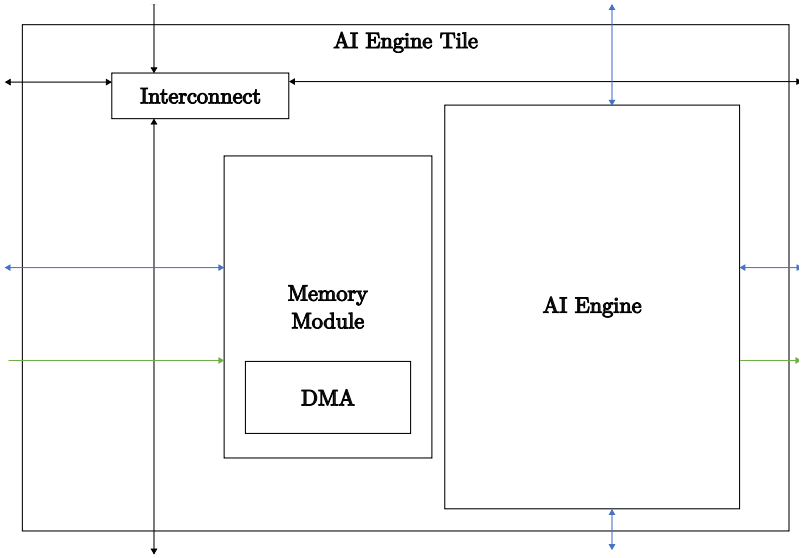


Figure 2.4: The black arrows represent AXI4 interconnects. These are connected to a net of AXI4-streams. The green and blue arrows are directly connected to neighbouring tiles. The blue arrows represent how neigbouring tiles can access the memory module and engines, meanwhile the green arrows represent how the engine's capability to stream directly into neighbouring tiles. as seen, the memory modules contains direct memory access (DMA). The figure was inspired by images in [18].

The AIEs themselves are made up of multiple parts. A vector unit, a scalar unit, a number of address generator unit (AGU)s for loading and storing data and an instruction fetch and decode unit. This is shown in Figure 2.5. The engine is can be described as a "highly-optimized processo", which utilises single instruction, multiple data (SIMD) and VLIW architecture[18].

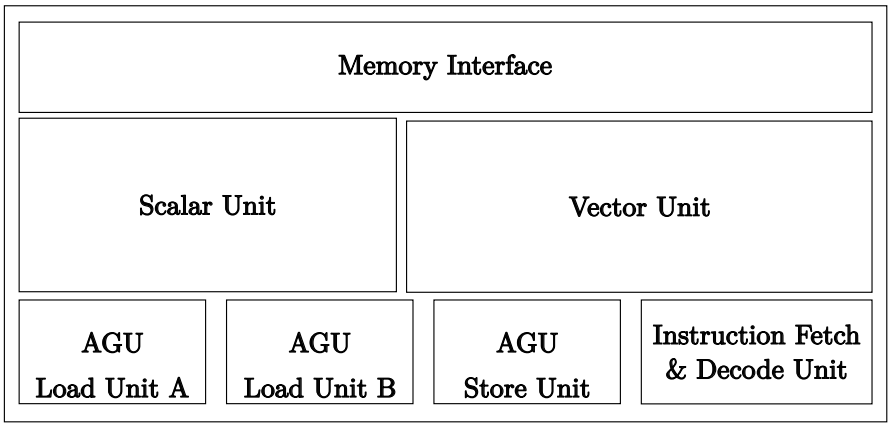


Figure 2.5: Block diagram of an AIE. The figure was inspired by images in [18].

2.3.2 Designing the AIEs

In order to take full advantage of the AIEs, the AMD AIE application programming interface (API) has to be used. As mentioned in Section 2.3.1, the VLIW architecture is a core part of how the AIE technology operates. This means that all data that is to be processed by the AIEs has to be vectorised. The API supports a number of data types that can be used within these vectors; these data types are listed in Table 2.2.

Table 2.2: Supported data types and corresponding vector sizes in the AMD API[22].

Type	Supported Sizes
`int8`	16/32/64/128
`uint8`	16/32/64/128
`int16`	8/16/32/64
`int32`	4/8/16/32
`float`	4/8/16/32
`cint16`	4/8/16/32
`cint32`	2/4/8/16
`cfloat`	2/4/8/16

The data types `int` and `float` follow standard C++ syntax, meaning they are used when integers or decimal values are to be handled. Data types `cint` and `cfloat` are specific to the AIE API. The letter `c` indicates that the data types are complex versions of their real-valued counterparts, meaning `cint` handles complex integers and `cfloat` can handle complex values with decimals. Notice that increased complexity limits the possible vector size, as seen in Table 2.2. The table also shows that the `cfloat` uses single-precision since it uses 64 bits for the entire complex value.

The AIE API offers several predefined operations that can be used to process the data within the vectors. Addition, subtraction, transpose and conjugate are a few examples of available operations that perform element-wise computations on the

vectors [22]. Moreover, a function to compute matrix multiplications is also predefined. The dimensions of the operation are, however strictly limited by the data types used to perform the multiplication. Table 2.3 and Table 2.4 present the different possible configurations for various combinations of data types. Table 2.3 and Table 2.4 shows the supported supported matrix dimensions for a matrix $\boldsymbol{A}$ of size $m \times k$ and a matrix $\boldsymbol{B}$ of size $k \times n$ [22].

Table 2.3: Matrix multiplication configurations for real number data types for a matrix $\boldsymbol{A}$ of size $m \times k$ and a matrix $\boldsymbol{B}$ of size $k \times n$ [22].

Mode	**Supported Matrix Sizes** ($m \times k \times n$)
`int8 x int8`	4x8x4, 4x16x4, 8x8x4, 2x8x8, 4x8x8, 2x16x8, 4x16x8
`int16 x int8`	4x4x4, 8x4x4, 4x8x4, 4x4x8
`int8 x int16`	4x4x8, 4x4x4, 8x8x1
`int16 x int16`	4x4x4, 2x4x8, 4x4x8, 4x2x8, 8x8x1
`int32 x int16`	2x4x8, 4x4x4, 4x2x4, 2x2x4, 2x4x4, 4x4x2, 2x2x8
`int16 x int32`	4x2x2, 2x4x8, 4x4x4
`int32 x int32`	4x2x4, 2x2x2, 2x4x2, 2x8x2, 4x2x2, 4x4x2, 2x4x4, 4x4x1
`float x float`	4x2x4, 2x2x2, 2x4x2, 2x8x2, 4x2x2, 4x4x2, 2x4x4, 4x4x1

Table 2.4: Matrix multiplication configurations for complex number data types for a matrix $\boldsymbol{A}$ of size $m \times k$ and a matrix $\boldsymbol{B}$ of size $k \times n$ [22].

Mode	**Supported Matrix Sizes** ($m \times k \times n$)
`int16 x cint16`	4x2x2, 4x4x4, 4x4x1
`int16 x cint32`	2x4x2, 2x4x4, 2x8x2, 4x4x2, 4x4x1
`cint16 x int16`	2x2x4, 2x2x8, 2x4x4, 2x4x8, 4x2x4, 4x4x2, 4x4x4
`cint16 x cint16`	2x2x2, 2x4x2, 2x8x2, 2x4x4, 4x2x2, 4x4x2, 4x2x4, 4x4x1
`cint16 x int32`	2x2x2, 2x4x2, 2x8x2, 2x4x4, 4x2x2, 4x4x2, 4x2x4, 4x4x1
`cint16 x cint32`	2x2x2, 2x4x2, 4x2x1
`int32 x cint16`	2x2x2, 2x4x2, 2x8x2, 2x4x4, 4x2x2, 4x4x2, 4x2x4, 4x4x1
`int32 x cint32`	2x2x2, 2x4x2, 4x2x1
`cint32 x int16`	2x4x2, 2x8x2, 2x4x4, 4x4x2
`cint32 x cint16`	2x2x2, 2x4x2, 4x4x1
`cint32 x int32`	1x2x2, 2x2x2, 2x4x2, 4x4x1
`cint32 x cint32`	1x2x2, 2x2x1, 2x2x2, 2x2x1
`float x cfloat`	2x2x2, 2x4x2, 4x2x1
`cfloat x float`	2x2x2, 2x4x2, 2x4x1
`cfloat x cfloat`	2x2x2, 2x2x4, 2x4x2, 4x2x2, 4x2x1

As Table 2.3 and Table 2.4 show, handling complex numbers with high precision is costly. An increased precision leads to a reduction in operation size. To compensate for the size reduction, a greater number of operations will be required.

2.3.3 Vitis

For creating an AIE design, the AMD-created tool, Vitis, is usually used. This tool offers two ways of simulating the AIEs, the x86 simulator and the AIE Simulator.

The x86 simulator only simulates the functionality of a design, it does not take any hardware constraints into account. This lack of constraint makes it possible to add multiple print statements that could be used for debugging. The AIE simulator then places the kernel tiles on the AIE array, creates the routing between them, analyses the timing and memory of both the buffers between the kernels and the program on the tiles. By testing the design together with the PS and PL implementation, the hardware can be emulated. This builds an implementation which takes all three parts into account [23].

2.4 Central Processing Unit

The CPU is the main processing unit of a computer. It follows an instruction cycle where it goes through three stages: fetch, decode and execute. First, the fetch stage gets the instruction, the decode translates the instruction, and then the execute stage executes it. The execution is either an ALU calculation or a memory operation. The instruction cycles are then repeated for the next instruction, sequentially moving through, from when the computer turns on until it turns off. Designing it this way gave the CPU a strength in sequential execution. This hindered performance, leading modern CPUs to consist of multiple cores. Each core could then have its own instruction cycle [24]. The CPU used for this project is the Intel Core Ultra 9 Processor 285H, the specifications for this CPU are shown in Table 2.5.

Table 2.5: Specification for the Intel Core Ultra 9 Processor 285H [25].

Specifications	
Cores	16
Threads	16
Performance-core Base Frequency	2.9 GHz
Processor Base Power	45 W

2.5 Graphics Processing Unit

The hardware architecture of an GPU can help in implementing parallel computing. To achieve this, the hardware is built from a large number of cores, each core being a simplified version of a CPU core. This makes it possible to fit a larger number of cores on the chip and creates the parallel computation capabilities [26]. All these cores then use the SIMD paradigm. The paradigm implies all cores perform the same type of computation, with the difference that the cores perform the computation on different parts of a larger data set [27].

Each of the cores has a hierarchical structure with the smallest part being a thread. Multiple threads are then grouped into thread blocks. These thread blocks often share memory, making it possible for the threads to share data with low latency between them. The thread blocks are then placed into a grid, which can perform the complete task [26]. In Table 2.6 the specifications for the NVIDIA QUADRO RTX 6000 are shown.

Table 2.6: Specification for the NVIDIA QUADRO RTX 6000 GPU [28].

Specifications	
CUDA Parallel-Processing Cores	4608
NVIDIA Tensor Cores	576
NVIDIA RT Cores	72
GPU Memory	24 GB (GDDR6)
FP32 Performance	16.3 TFLOPS
Maximum Power Consumption	295 W

2.5.1 Programming the GPU

To program the GPU, NVIDIA have developed three separate but connected platforms and libraries. Compute Unified Device Architecture (CUDA) is a platform for accessing the parallelisation possibility of the GPU [29]. It gives the possibility to assign functions to either threads or thread blocks. Further, cuBLAS is an API library used for executing matrix arithmetic on the GPU. For more advanced matrix operations, for example LU-decomposition, the cuSOLVER API library is instead called. These are two separate libraries, but they are linked together. The cuSOLVER library is built upon the cuBLAS library. This means during an API call to cuSOLVER, the cuSOLVER API calls cuBLAS for the simpler matrix operations [30] [31].

3
Method

This chapter will explain in detail the steps taken during the project. First the development methodology will be explained, which includes the workflow and the development environments. The following section, Section 3.2, will explain how the results of the implementations were verified. Lastly, Section 3.3 will explain steps taken to achieve the testing results.

3.1 Development Methodology

The two following sections will discuss the thoughts behind the methodology. The first section will mainly revolve around the layout of the workflow. The second section will go into detail about the development environments that will be used for the project.

3.1.1 Development Workflow

The implementation workflow began with designing the MATLAB implementation. As MATLAB offers a relatively forgiving development environment, it was a good choice for developing a prototype of the GIP algorithm. In addition, there was already a complete function for target injection provided by Weber and Wright [9]. Reusing code helped save time and maintain focus on the project's main objective.

The design of the GPU-implementation commenced once the design of the MATLAB implementation started entering its final stages. It was desirable to start designing the hardware as soon as possible, and working on implementations in parallel was a good way to achieve that. As the MATLAB design functioned as a template for the design on the GPU, it was not optimal to start working on both designs in parallel from the project start, as the GPU would have suffered from any delays in the MATLAB design. Furthermore, the GPU implementation was also be dependent on noise video generation from the MATLAB implementation, as it imported the data instead of generating it, more of this in Section 4.1.

Designing the AIE implementation took place last, even though it was the main objective of the project. The reason for this was simply due to practicalities. The project had access to the development environment related to the GPU implementation first, thus it was more time efficient to start working on that design first.

Similar to the GPU implementation, the MATLAB implementation acted as a template for the AIE implementation. However, due to the more complex structure of

an AIE project compared to the GPU implementation, taking larger sidesteps from the MATLAB implementation was expected. Notice that the project favoured suitability before similarity, which meant that the algorithms that suited the hardware architecture better was picked. Thus there were some differences between the implementations at the end of the project. This will be discussed further in Chapter 4.

3.1.2 Development Environments

For the MATLAB implementation, the MATLAB integrated development environment (IDE) was be used for all development, simply because it is the easiest tool to use for MATLAB development. For the GPU development, a mix of environments was used. NVIDIA CUDA Compiler Driver was used for compiling the applications, while Visual Studio Code was used for writing the applications. The MATLAB IDE was used as a wrapper, which both called the applications and read the results.

The development of the AIE implementation only used AMD's development environments. The Vitis IDE was the main environment where most of the coding took place. Simulations, emulations and hardware connections are also found native in Vitis. Vitis does not offer analysis tools for block diagrams or waveforms; here, Vivado will be used as a complement. Vivado was also used for generating hardware platforms to use in the Vitis project for the AIE implementation.

3.2 Verifying the Implementations

Verification played an important role in the project. It was of high priority to ensure that the functionality of the algorithm remained the same regardless of the platform. The following sections will further discuss the verification process for all three implementations.

3.2.1 MATLAB Implementation

For the MATLAB implementation, the main way of verification was to compare the output from the non-homogenous detector (NHD) with an energy detector (ED). This meant an ED also had to be designed and implemented in MATLAB besides the NHD implementation. By varying the position of the injected targets and then comparing the outputs of both, it was clearly observed that the NHD implementation more distinctively detected targets that were located closer to each other.

3.2.2 GPU Implementation

Since the verification of the GPU implementation took place after the verification of the MATLAB implementation, the MATLAB implementation could be used as a reference standard to test the GPU implementation against. As mentioned in Section 3.1.2, MATLAB was used as a wrapper, meaning it was used to call on the terminal and run the GPU code. In practice, a noise video was generated and

then input directly to both the MATLAB implementation and the GPU implementation. The outputs could then be directly compared against each other by using the Frobenius norm [32].

By generating new data which is directly input into both implementations, many iterations of test data could easily be used. This lead to a higher degree of certainty that the GPU implementation behaved as expected since it produces results similar to the MATLAB implementation.

3.2.3 AIE Implementation

Similar to the GPU implementation, the AIE implementation would be compared to the MATLAB implementation to ensure the desired functionality is achieved. However, in contrast to the GPU implementation, only GIP values were calculated by the AIE implementation. Thus, the GIP values from the MATLAB implementation was used to compare the AIE results by simply calculating the difference between the value of each implementation.

3.3 Testing the Implementations

Extracting results was another vital part of the project. The following sections will discuss the approaches used for achieving the results which are presented in Chapter 5. As the MATLAB results are achieved trivially, only the GPU and AIE implementations will be discussed.

3.3.1 GPU Implementation

Execution time was the main result to collect from the GPU implementation. The collection was handled by a MATLAB script that read and plotted the time data output from the GPU code. Inside the GPU code, time measurements are handled by the `<sys/time.h>` header by simply putting time stamps in the code. These time stamps were written to a `.txt` file in the end of the code.

For the implementation that performs the entire GIP algorithm, manual re-runs were performed about 15 times. The relatively low number of re-runs was considered enough as the resulting curves stabilized around here.

For the implementation that only calculated single GIP values, higher accuracy was needed, as microsecond instead of minutes were measured. Therefore, the calculations were repeated 10,000 times by looping the code. The time stamps were put outside the loop, and because of this the final time had to be divided by 10,000 to retrieve the avarage execution time. when the execution time of the whole GIP implementation was measured. When individual parts of the algorithm was measured, the time stamps were put inside the loop. The time stamps were wrapped around each function of interest. The average time of the iterations was calculated once all measurements had been completed.

3.3.2 AIE Implementation

To perform tests on the AIE implementation, the project was first built and then run under the simulator menu in Vitis. The results could then be viewed either by directly using the Vitis graphical user interface (GUI), or by accessing the generated simulation files directly. Accessing the files directly was beneficial when using MATLAB scripts for reading result data.

For measuring the time of the entire implementation, the AIE simulation output was used. The simulator used a processor frequency of 312.5 MHz, which is the standard for the AIEs. For measuring the time of the individual parts, the waveforms were exported and inspected individually.

4

Design

This chapter will first explain the design of the GIP algorithm. The explanation of the GIP design will be followed by a description of every implementation created during this project. For the GPU, the design utilising the GPU will be described. The AIE implementation on the other hand is divided into two parts; one hardware design and one simulation design. The hardware design was intended to run on hardware, but never made it that far, while the simulation design is the design used to retrieve the results presented in the next chapter.

4.1 Noise Generation and Target Injection

The noise video is formed using the MATLAB function `randn`, which forms a matrix with normal distributed numbers. By changing the input of the function, the size of the matrix can be altered. The function is then called for both the real and imaginary values, together forming a matrix of complex numbers. To normalise the noise levels, the matrix is then divided by $\sqrt{2}$. Randomising the noise this way makes it possible to change the matrix size for easier debugging. It also means clutter is not an issue and more focus can be spent on the GIP implementations.

After the construction of the noise video, target injection takes place. The target injection function from Weber and Wright was used, which allows targets to be placed at a certain position and at a specific strength in the noise video [9]. The targets are added to the noise video based on the given input strength and position, but also based on a steering vector that is calculated in MATLAB.

The export of the noise matrix is handled by writing the values to a `.txt` file. The first two rows consist of the number of pulses and range bins. One value is then given one line going row-wise through the matrix.

For the GIP design, the number of pulses, M, was equal to 32. In addition, a total of 8 guard cells were used, i.e. 4 on each side of the CUT, and the injected targets had an SNR of roughly 20 dB. The number of pulses, 32, was chosen as it offered a good balance between being computationally tough to run, but still being able to perform computations in a realistic time frame. Further, the number of guard cells was chosen since it provided detection statistics where noise and targets were clearly distinguishable from each other. Using any more guard cells did not increase the gap between noise and targets significantly.

The precision for the complex floats in the MATLAB implementation used double

precision. This choice was made to have the highest accuracy but still be within reasonable bounds for the number of bits.

4.2 Designing the GIP Algorithm

The first step of the GIP algorithm was to form what we refer to as the GIP matrix. The GIP matrix consists of all range bins which will be required to calculate the needed GIP values for a given CUT. As described in Section 2.1.3, there are a total of $4M$ GIP values calculated for each CUT. In turn, each GIP value requires $2M$ range bins to be computed. With these parameters in mind, the dimensions of the GIP matrix will have to be of size $M \times 6M$ to fit all the required range bins. Of those $6M$ range bins, GIP values will only be calculated for the $4M$ bins placed in the middle of the GIP matrix, the remaining bins are solely included to compute the GIP values at the edges. Figure 4.1 shows the first iteration of a sliding window approach used to calculate GIP-values for a given CUT coloured green. In Figure 4.1, the blue-coloured range bin is the first of $4M$ range bins to be calculated, while the yellow coloured bins represent the training data used to compute that specific GIP-value.

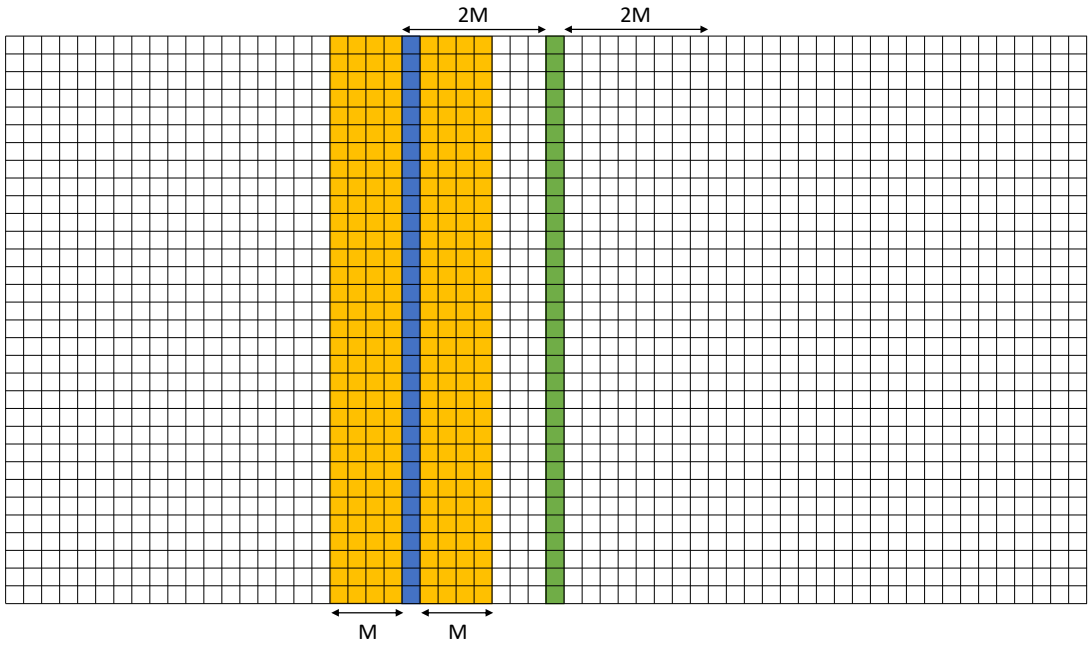


Figure 4.1: First iteration of a sliding window. The GIP value is being calculated for the blue coloured range bin, while the yellow range bins represent the training data used to compute that specific GIP value. The green range bin marks the CUT whose GIP-values is being calculated.

In the next step of this sliding window approach, the blue and yellow range bins will move one step to the right, as shown in Figure 4.2. A new GIP-value is thus calculated. This sliding window will keep moving one bin at a time until all $4M$ GIP values have been computed.

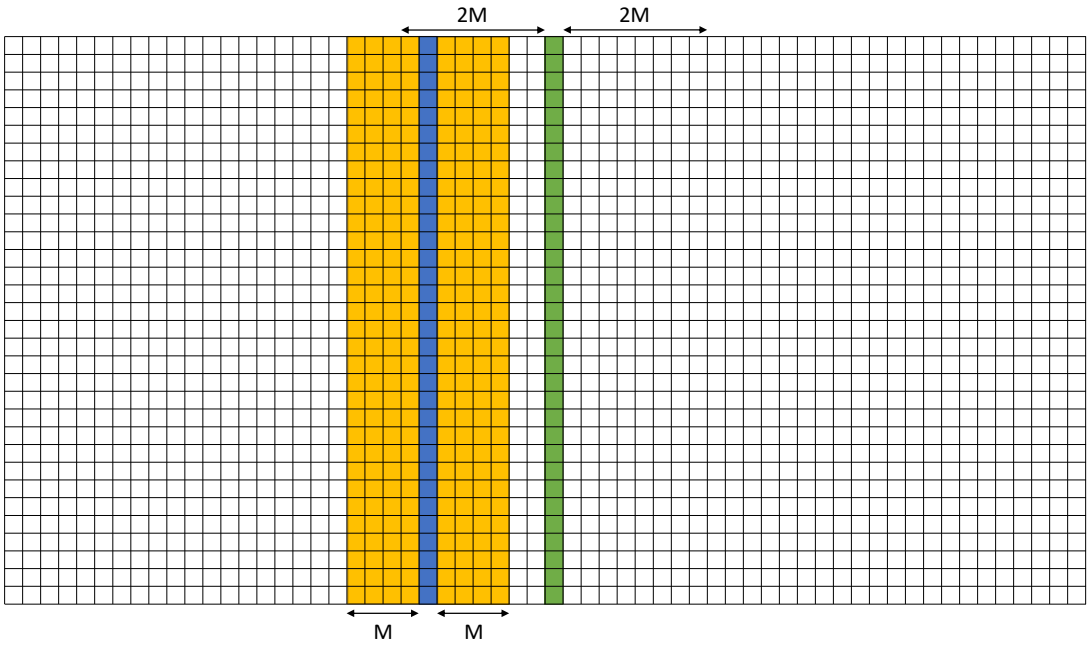


Figure 4.2: Second iteration of a sliding window. Both the blue and yellow coloured range bins have moved one step to the right compared to Figure 4.1.

When the CUT is located less than $3M$ from the leftmost range bin in the noise video, the GIP will wrap around the matrix to data for the CUTs close to the edge of the noise video. This is shown in Figure 4.3. The same principle applies when the CUT is located less than $3M$ bins from the rightmost bin.

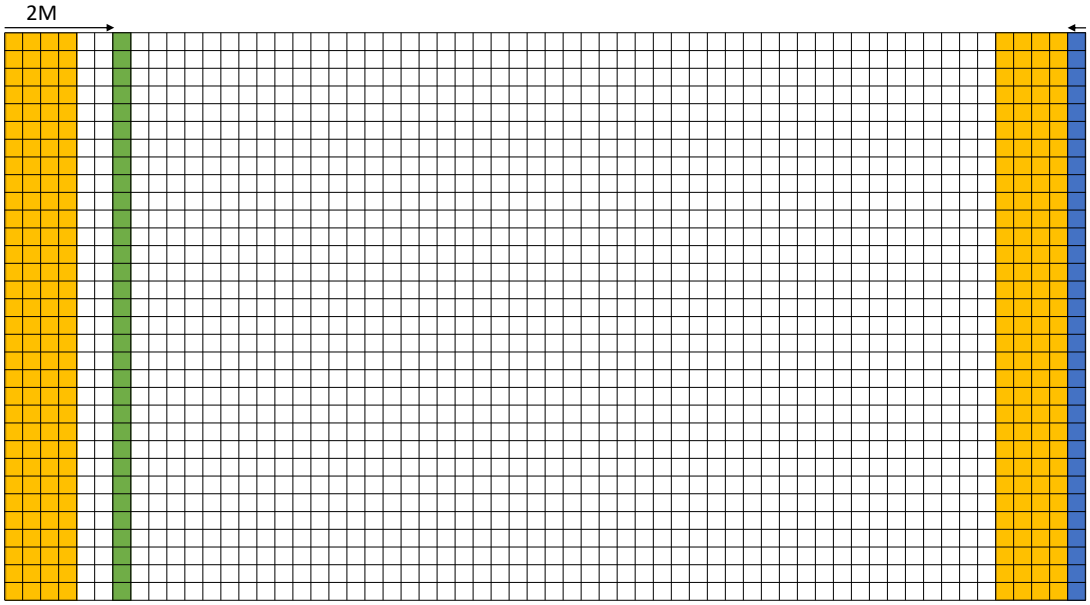


Figure 4.3: When the CUT is located close to the edge of the noise video, range bins on the other side of the noise video have to be used for the current CUT.

After all $4M$ GIP-values have been computed, they are sorted based on the size of the error to the expected value $E(P)$ as described in Section 2.1.3. The $2M$ values closest to $E(P)$ are then used to compute the detection statistic for the CUT, which is the green coloured range bin in Figure 4.1. This procedure then repeats with a sliding window approach until detection statistics for all CUTs have been computed. This entire procedure is also described in pseudocode found in Appendix A.

4.3 GPU Implementation

The GIP-algorithm is implemented for the GPU using the overall structure as the MATLAB implementation. But for the matrix operations, it instead uses cuSOLVER and cuBLAS. For the matrix multiplication of the input training data, `gemm` was used, which is a function for matrix multiplication. When the covariance matrix has been calculated, `getrf` is used for performing a LU decomposition. This function saves both the L and U in the same matrix called $\boldsymbol{LU}$. The function `getrs` then solves the linear system with the $\boldsymbol{LUy} = \boldsymbol{x}$ using forward- and backward substitution and saves the resulting $\boldsymbol{y} = \boldsymbol{LU}^{-1}\boldsymbol{x}$. Finally, `dotc` is used for the last threshold calculations $t = \boldsymbol{x}^{\boldsymbol{H}}\boldsymbol{y}$. These choices were made to improve execution speed by not having to perform the matrix inversion. This is instead handled by the forward- and backward substitution.

Two GPU implementations were created, one utilising `cuComplex`, which uses 64 bits for the real and imaginary parts together, and one utilising `cuDoubleComplex`, which uses 128 bits. The two implementations also changed which version of the earlier-mentioned functions to call. These two implementations aimed to analyse how the speed and computational accuracy changed in comparison to the MATLAB implementation.

4.3.1 Importing and Exporting Data

Since CUDA can use C++ libraries, it was possible to use these for reading and writing to files. However, MATLAB stored the complex values as $a \pm bj$, which meant every row had to be correctly parsed, removing the j and dividing up the real and imaginary value to correctly form either the `cuComplex` or `cuDoubleComplex`. Since the two libraries use column-wise vectors to represent a matrix, the read value had to be saved to the correct location.

4.4 AIE Hardware Implementation

The design intended for the hardware, unfortunately, never reached an actual hardware implementation and was only used on the built-in emulator in Vitis. Figure 4.4 shows an illustration of the data flow for the design. As shown, the data is first generated inside the PS, then it is transferred through the PL to the AIEs, where the data processing occurs. Once the processing is complete, the data is transferred back to the PS through the PL. The intention of the design is merely to handle a smaller matrix inversion of size 4×4.

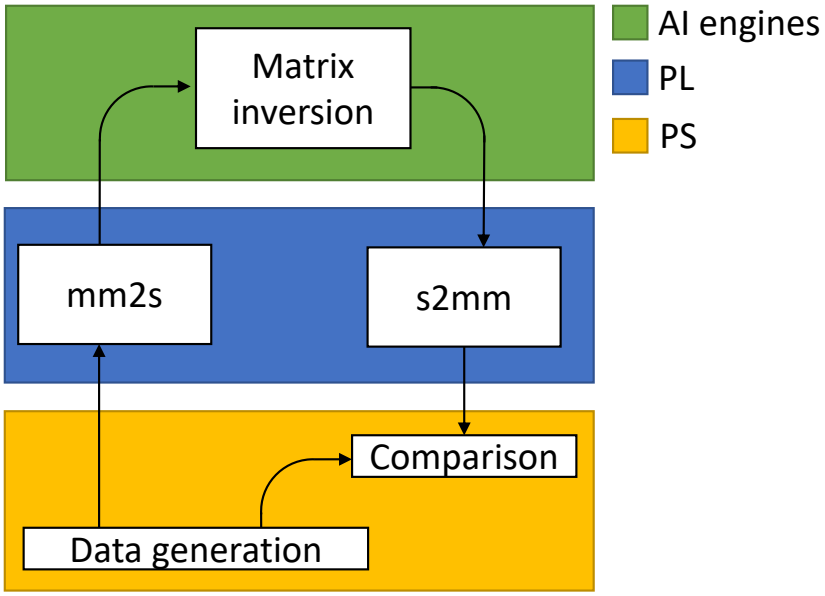


Figure 4.4: Block schematic showing the data flow for the intended hardware design.

4.4.1 Memory and Stream Interconnection

As shown in Figure 4.4, the PS and AIEs are interconnected through the PL. To design these PL blocks, High-level synthesis (HLS) was used for creating the register transfer level (RTL) designs. The reason for this was mainly due to the openly available code from AMD. This code could be reused, which was time-efficient.

The PL functions memory mapped to stream (MM2S) and stream to memory mapped (S2MM) are handling data transfers across the domains. MM2S reads the content stored at a specific address in the DDR-memory. The data from that specific address is then directly streamed to the AIE domain. S2MM works similarly, but in reverse. Output data from the AIE domain is streamed to a specific memory address, where it is stored. Simplified, MM2S acts as a read function for the AIEs, while the S2MM acts as a write function.

Figure 4.5 shows the tiling and wiring layout generated by Vitis for the intended hardware design. As seen in the figure, there are buffers in between all kernels. The kernels wait for these buffers to become fully loaded before they initiate execution.

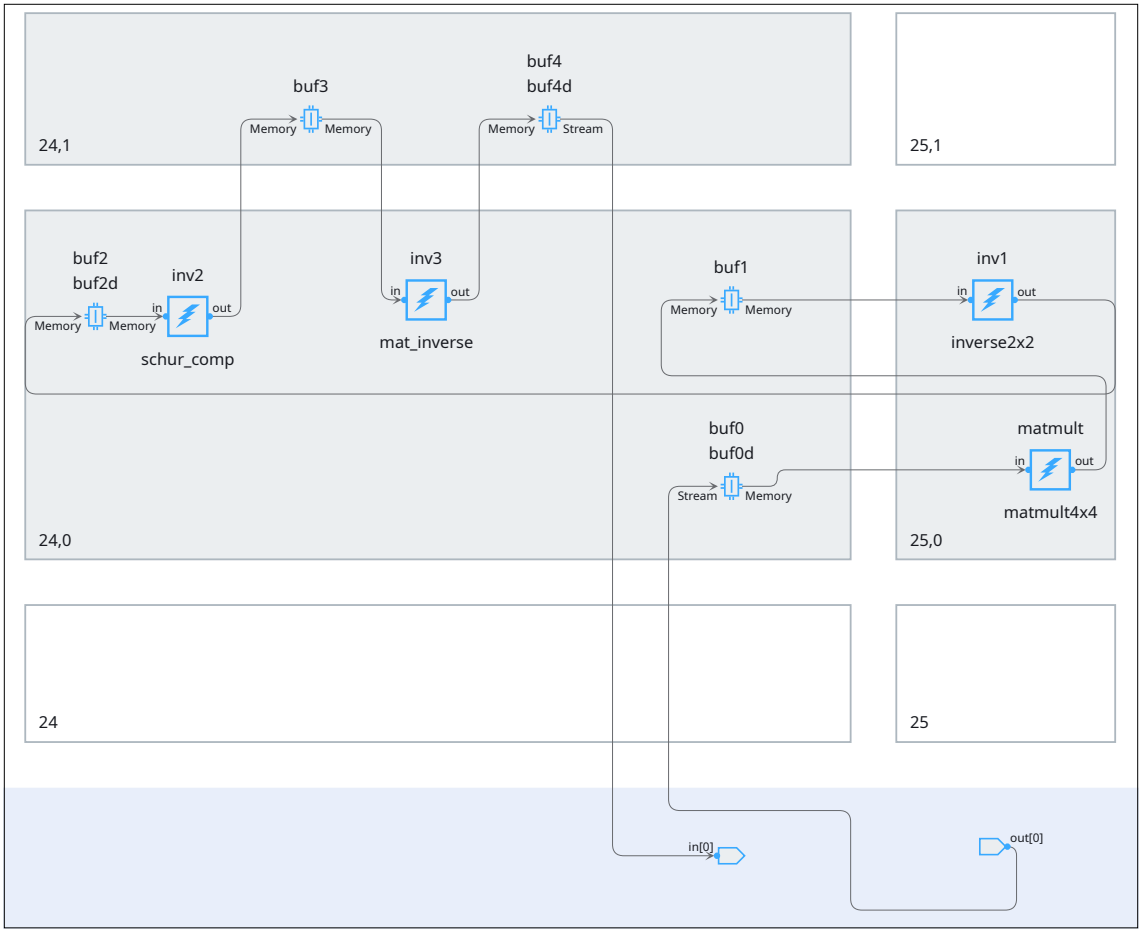


Figure 4.5: A Vitis figure that shows the tiling and wiring layout for the intended hardware design. Block 24 and 25 are interface tiles. The remaining numbered blocks represent an artificial intelligence engine (AIE) tile, while the blue area represents the PL. The ports `in[0]` and `out[0]` are seen from the PL and PS perspective, meaning `out[0]` is the input to the AIEs and `in[0]` is the output from the AIEs.

4.4.2 Matrix multiplication and Matrix Inversion

The hardware design was implemented to handle a matrix multiplication and matrix inversion for 4×4 matrices. Figure 4.6 shows a graph directly taken from Vitis, which illustrates the different blocks the design is made up of. Note that each square containing a blue lightning bolt represents an AIE kernel.

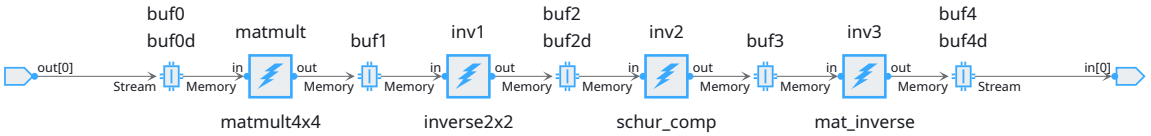


Figure 4.6: A graph from Vitis which shows the different parts of the hardware design. The blue-edged squares containing a lightning bolt symbol represent an AIE kernel. Note that `out[0]` and `in[0]` refers to the PL, which means the output is at `in[0]` and vice versa.

The first kernel and the first step of the design, referred to as `matmult4x4`, was intended to multiply two 4×4 matrices with each other. In the current design, a matrix is simply multiplied by itself. The resulting matrix is also split into smaller partitions in this step. All the partitions are shuffled through to the `inverse2x2` block, but only the top leftmost block, referred to as A in Section 2.2.3, is being processed here. The block is inverted, as it is required in order to calculate the Schur Complement, which is the next step of the process. Once the inversion of partition A has been completed, all partitions are once again shuffled through to the block named `schur_comp`. In this stage of the process, the Schur Complement, which is referred to as S in Section 2.2.3, is calculated.

Inside the kernel named `schur_comp`, the Schur Complement S is both calculated in accordance with (2.25) and inverted. The inversion is done in order to faster compute (2.24) in the next step of the process. When the Schur Complement has been both calculated and inverted, the partitions referred to as A^{-1}, B, C and S^{-1} in Section 2.2.3 are moved to the last part of the process. The kernel named `mat_inverse` represents the last stage of the 4x4 matrix inversion, and inside this kernel (2.24) is applied. The Schur Complement was chosen since it was seen as a good fit for the AIE architecture and provided an efficient way of performing matrix inversions.

As seen in Figure 4.5, some functions share tiles. Due to the relatively small size of the functions, the runtime ratio for all kernels was set to 0.5. In the AIE compiler, the runtime ratio specifies the expected utilization of a tile by a kernel and is used during resource allocation and placement. A value of 0.5 indicates that the kernel is estimated to occupy no more than half of the execution time available on a single AIE tile.

Notice that the ports called `out[0]` and `in[0]` refer to the PL and PS. This means `out[0]` is the output of the PL and in turn the PS, thus it is the output port seen from the AIE perspective. Vice versa applies for `in[0]`.

4.5 AIE Simulator Implementation

The AIE simulator design focuses on calculating a single GIP value. This choice was made for clearer comparison of the matrix operation with the other implementations. This meant creating the GIP matrix and picking out the training data would be handled elsewhere, either in the PL or PS. Then the correct matrices would be transmitted to the AIEs. In this design, the two input streams were the $M \times 1$ current GIP cell and the $M \times 2M$ training data. The aim of the design was then to calculate a GIP value for the given input. In Figure 4.7, the flow of these computations is shown.

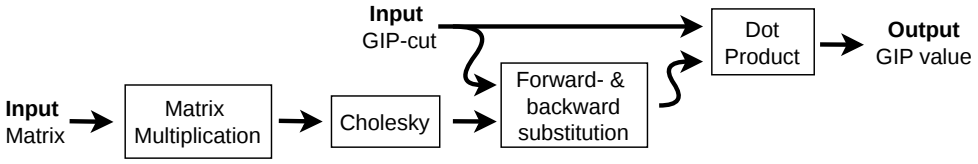


Figure 4.7: Block schematic showing the data flow for the AIE design.

The AIE flow begins with the training data being fed to the matrix multiplication tile, while the GIP cell is fed to both the block handling the forward- and backward substitution and the block handling the dot product. After the matrix multiplication, the matrix is fed to the Cholesky block before the output is fed to the forward- & backwards substitution, its output then is fed to the dot product block. The implementation used buffers across the design, which meant two buffers were placed between each tile in the design. For each of the tiles in the design, a runtime ratio of 1.0 was chosen, giving each block a whole tile.

From Table 2.4 in Section 2.3.2, it is known that when using `cfloat`, the matrix multiplications are limited to 2×4, this led the design to create a 4×4 matrix multiplication block using four 2×4 multiplications. The AIE API also does not utilise matrices but instead flattened vectors. This means the design was built for the 4×4 matrix size but with the scalability to work on a 32×32 matrix. To solve this, an 8×8 2-D array was created with each element of the array being a 16-long vector. This is shown in Figure 4.8. The size of the 2-D array could then be changed to achieve different sizes of the matrix. Each of these sizes would be a multiple of four.

In the GPU implementation, an LU-decomposition was used; this was due to the built-in functions in the cuSOLVER library. But the AIE uses Cholesky instead. This choice was made for two reasons. Firstly, the LU-decomposition normally needs to store two matrices, both the L and U, which in a small tile with little memory creates problems. Secondly, the LU-decomposition has the risk of needing to pivot the different elements. This creates extra computations which become difficult for AIE architecture when using the block design. These two reasons resulted in implementing the Cholesky for the AIE design, even if the LU was used for the GPU.

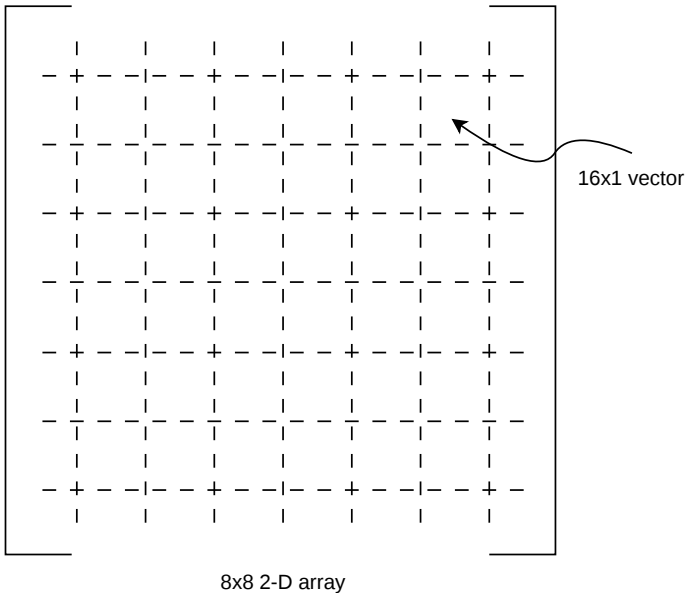


Figure 4.8: Composition of a 32×32 matrix using 16 long vectors and a 8×8 2-D array.

4.5.1 Matrix Multiplication

At the start of the flow, the input matrix is multiplied by itself. The core of this multiplication is a basic matrix multiplication utilising three for-loops, but with optimisations to reassure it fits inside a tile and increase the computational speed.

Since this is a large matrix, it could not all be stored in the tile at the same time. Thus, the values had to be read from an outside buffer. The nature of repeated values in a matrix multiplication made these amounts of read operations cause an increase in computation time. Two techniques were used to decrease the computation time of the matrix multiplications. The first one was storing the two rows which the multiplication loop over. The second one arises from the fact that a matrix multiplied by its own Hermitian results in the upper and lower triangles of the matrix being conjugate to each other. This means only the lower triangle needs to be calculated.

The algorithm begins with the first row, which is symbolised with green in Figure 4.9. It then loops through all the earlier rows, including the current row and stores the two rows with the second row being symbolised by orange. When a row is orange and green, it symbolises that the same row is stored. It then initiated four AIE multipliers, which are used when looping through the two saved rows. On the first iteration, it uses `mul` for multiplying them together, while the other iterations use `mac`, which multiplies and accumulates the results from earlier. The Hermitian is handled inside the 4×4 multiplier; this means the elements get transposed and conjugated. This decision stems from the decision to keep only two rows stored at a time, which means the values are continuously discarded after being used. In Algorithm 1, the pseudo code for this block is shown.

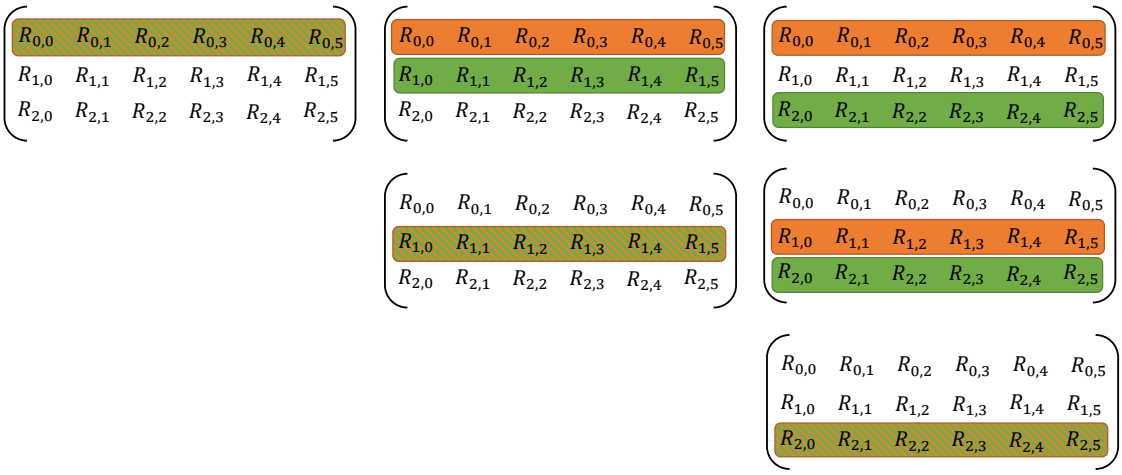


Figure 4.9: The choice of which rows to store for computations. The green is chosen for one iteration, and then the orange one cycles through the other rows. When a row is coloured both orange and green, that row is saved twice for that iteration.

Algorithm 1 Matrix multiplication algorithm

```
M ← size of matrix
for i ← 0, M do
    A ← Load row i
    for j ← 0, i do                          ▷ only loops lower triangle
        B ← Load row j
        for k ← 0, 2M do
            R ← matmul(A_k, B_k)
        end for
        out ← R
    end for
end for
```

4.5.2 Cholesky Block

Since the implementation utilised the 2-D array of vectors, the Cholesky needed to be altered to match this. The main part of this was updating the rest of the elements in the 2-D array after the Cholesky had been calculated for a 4×4 matrix. These calculations are shown in Figure 4.10. The Cholesky function is the calculations shown in (2.21) and (2.22) for a 4×4 matrix, while the rest are matrix operations performed with the elements in the 2-D array. The operation labelled as R^{-H} is the Hermitian followed by an inverse. The next Cholesky is then applied to the second element on the diagonal, the first column does not have to be updated this time. This is repeated until the Cholesky has been calculated for each diagonal and all corresponding values have been updated.

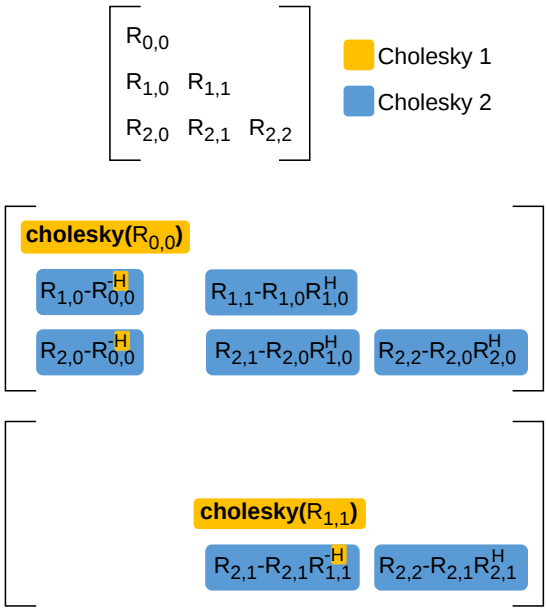


Figure 4.10: Computations for the Cholesky block design. The yellow parts are done by Cholesky 1, while the blue ones are done by Cholesky 2.

This Cholesky design does not fit in the 16 kB program size for a single tile. To solve this, it was divided into two parts, in Figure 4.10, the yellow operations are performed by the first part, while the blue ones are performed by the second part. For the next iterations, it is fed to another Cholesky pair, creating a pipeline structure as seen in Figure 4.11. The choice of having the first part do the Hermitian was to decrease the slower computation time of the second Cholesky. When increasing the matrix size, the number of these pairs of Cholesky implementations increases. The main problem with this implementation is the need to move the whole lower triangle matrix between each tile and the large number of tiles that increases with the matrix size.

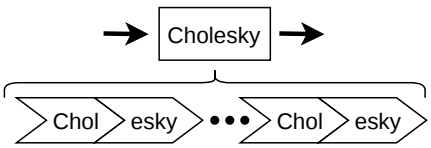


Figure 4.11: Splitting the Cholesky block into a pair. The number of pairs is then decided by the matrix size.

4.5.3 Forward- & Backward Substitution

The forward and backward substitutions are placed in the same tile. The forward substitution starts by reading the lower triangle, it then proceeds by taking the inverse of the diagonal element and multiplying with the read GIP-CUT-value, this is stored at the correct location. At the next iteration, it then uses the stored values before these values are multiplied by the element in the matrix, and this value is

then subtracted from the read CUT value. This value is then multiplied by the inverse of the second element on the diagonal before being saved at the correct place in the vector.

The backward algorithm then starts at the end of the read values. It takes the first element on the diagonal, taking the inverse Hermitian, and multiplying with the stored values from the forward substitution. It then loops through the elements on the row, taking the Hermitian before multiplying with the saved value from before. When the two loops finish, the values are output. The pseudo code for these two can be seen in Algorithm 2.

Algorithm 2 Forward- & Backward substitution algorithm

```
1:  M ← size of matrix
2:  for i ← 0, M do                                   ▷ Reading matrix input
3:      for j ← 0, i do
4:          L(i)(j) ← L_in++
5:      end for
6:  end for
7:
8:  for i ← 0, M do                                   ▷ Forward solve
9:      A ← C_in++
10:     for j ← 0, i do
11:         A ← A - L(i)(j)*y(j)
12:     end for
13:     y(i) ←L(i)(i)^-1 *A
14: end for
15:
16: for i ← M − 1, 0 do                               ▷ Backward solve
17:     A ← y(i)
18:     for j ← i + 1, M do
19:         A ← A - L(i)(j)^H *x(j)
20:     end for
21:     x(i) ←L(i)(i)^-H *A
22: end for
23:
24: for i ← 0, M do                                   ▷ Outputting x
25:     out ← x(i)
26: end for
```

4.5.4 Dot Product

The last block of the flow is the final dot product between the last two vectors. Since these are two $M \times 1$ vectors, the second vector does not need to be transposed. This is instead handled by the for-loop that goes through the vector, as shown in Algorithm 3.

Algorithm 3 Dot algorithm

1: $M \leftarrow$ size of matrix
2: **for** $i \leftarrow 0, M$ **do**
3: x $\leftarrow X$_in++
4: c $\leftarrow c$_in++
5: $c_c \leftarrow$ c* ▷ Conjugate
6: **for** $j \leftarrow 0, 4$ **do**
7: S $\leftarrow S + c_c(i) * x(i)$
8: **end for**
9: **end for**
10: out $\leftarrow$ real(S)

4.5.5 Support Functions

Four support functions were created, which were used at different stages in the design. The first one is a Hermitian for a 4×4 matrix, the AIE API has functions `aie::conj` and `aie::trans` which together form a Hermitian. But doing it this way seemed to increase the program size, which is why at places where the available program space was low, the created Hermitian was used instead.

The other two implementations were for matrix operations, the first one being $4 \times 4 \cdot 4 \times 4$ operations. This implementation uses the `aie::mul` for doing four $4 \times 2 \cdot 2 \times 4$ operations and concatenating the results together. Importantly, this is not the function that was used for the first large matrix multiplication, since that function was designed for numerous multiplications where the results were accumulated. This one performs a single matrix multiplication. The second one performs the $4 \times 4 \cdot 4 \times 1$ operations, which were done without utilising the AMD API and instead multiplying the different elements of the vectors. This choice was made since this size was not supported for `cfloat` as seen in Table 2.4. The fourth one is the matrix inversion for a 4×4 block. This block was constructed by combining the three blocks used for inversion from the hardware part.

5

Results

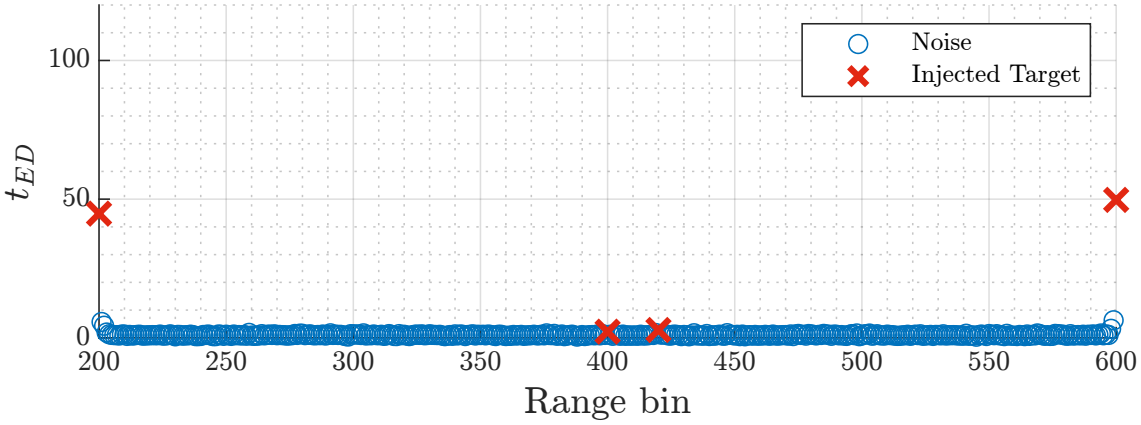
The following chapter will present all relevant results produced by the project. The chapter will first focus on verification of the generalised inner product (GIP) algorithm for each platform. Thereafter, a section regarding performance evaluation will follow. The section on performance evaluation will focus on comparing mainly how the platforms differ in terms of execution time. Lastly, the AIE utilisation will be presented.

5.1 Verification of the GIP Algorithm

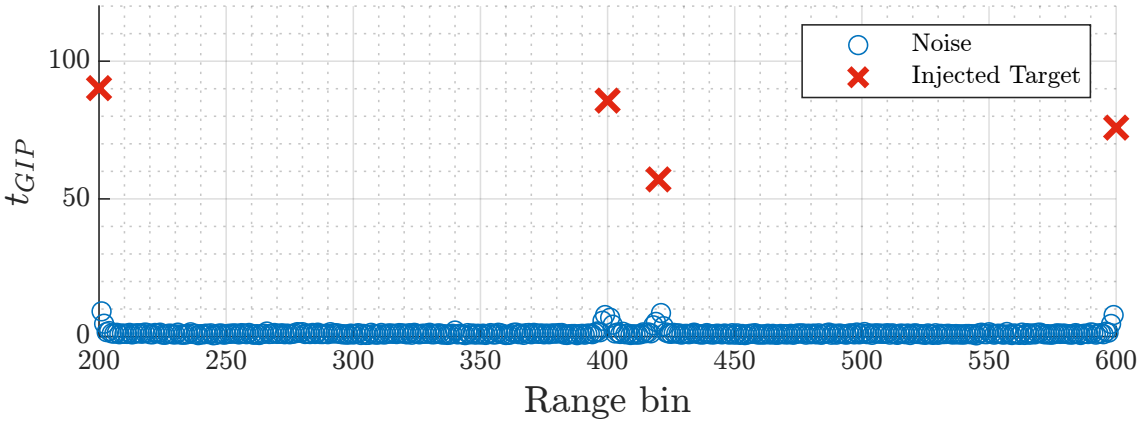
To ensure the GPU and AIE implementation follow the expected behaviour, they were compared against the MATLAB implementation. This MATLAB implementation was in turn compared against another MATLAB implementation that ran an energy detector (ED) without the aid of an non-homogenous detector (NHD). The effectiveness of the GIP-based NHD can be clearly observed when comparing Figure 5.1a with Figure 5.1b.

Note that there is no threshold set for these figures. The main purpose is to show that the injected targets are of higher magnitude for the ED that uses GIP-based NHD than for the ED without an NHD. By simply observing the figures, it is clear that it would be easier to fit a threshold in Figure 5.1b than in Figure 5.1a.

Figure 5.1a and Figure 5.1b show the difference between the two methods. Inside both figures, the injected targets are marked with red crosses, while the blue circles represent detection statistics calculated for range bins that only contain noise. Two of the four injected targets are located close enough for them to appear in each other's training data, located in range bins 400 and 420. As seen in Figure 5.1a, these adjacent targets are of lower magnitude compared to some of the range bins that only contain noise. Consequently, a number of false alarms will appear before the real targets are detected. However, if the NHD is applied to the ED, the threshold value is increased for the range bins that contain the two adjacent targets. In Figure 5.1b, the two adjacent targets can be seen to appear above all noise, meaning it is possible to detect both targets without any false alarms.



(a) Detection statistics for various data points after using the ED. Injected targets are marked with red crosses. The targets are placed in range bins 200, 400, 420 and 600.



(b) Detection statistics for various data points after using the GIP based NHD. Injected targets are marked with red crosses. The targets are placed in range bins 200, 400, 420 and 600.

Figure 5.1: Comparison of threshold values using the ED- and GIP - based NHD

5.1.1 Verification of the GPU & AIE Implementation

To verify the functionality of the GPU and AIE implementation, both were compared with the MATLAB implementation. Table 5.1 compares the accuracy of a fully implemented GIP algorithm in the GPU implementation with the MATLAB implementation as benchmark. Table 5.1 shows that both GPU implementations come close to the MATLAB standard values. The high precision GPU achieves results that are slightly closer to the standard values, but the difference between them is relatively small. In the table, MATLAB and GPU have the same precision both utilising double-precision, with the GPU low precision uses single. But even with the different precisions, the difference is small.

Table 5.1: Frobenius difference between the MATLAB and GPU implementations for an average over 50 runs. Δ_{avg} symbolises the relative difference compared to the standard value the MATLAB implementation.

Implementation	Δ_{avg}
MATLAB	0
GPU	3.5e-05
GPU low precision	3.9e-05

Table 5.2 shows the accuracy of the calculated GIP-values for the AIE implementation. Similar to Table 5.1, MATLAB is once again used as a benchmark. The comparison is made taking the average error for 50 calculated GIP values. In this table, MATLAB still uses double-precision while the AIE simulation uses single-precision.

Table 5.2: Difference between the MATLAB and AIE implementations for computations of GIP values for an average over 50 runs.

Implementation	Δ_{avg}
MATLAB	0
AIE simulator	2.0673e-07

5.2 Execution Times

In this section, the resulting execution times will be presented. First, the execution time required to calculate a single GIP value will be shown for every implementation, and at different matrix sizes. Secondly, a more detailed view of the execution times for the GPU and the AIE implementation will be presented. Lastly, the execution time for a full GIP algorithm based on the GPU and in MATLAB will be presented.

5.2.1 Execution time for a single GIP value

Figure 5.2 shows the execution time required to calculate a single GIP value from matrix sizes of 4 up to 100. The graph shows the execution times for calculations carried out on different platforms: MATLAB run on CPU, GPU hardware, and AIE simulation. The AIE implementation is relatively fast compared to the GPU; however, it quickly becomes slower as the size of the matrix increases.

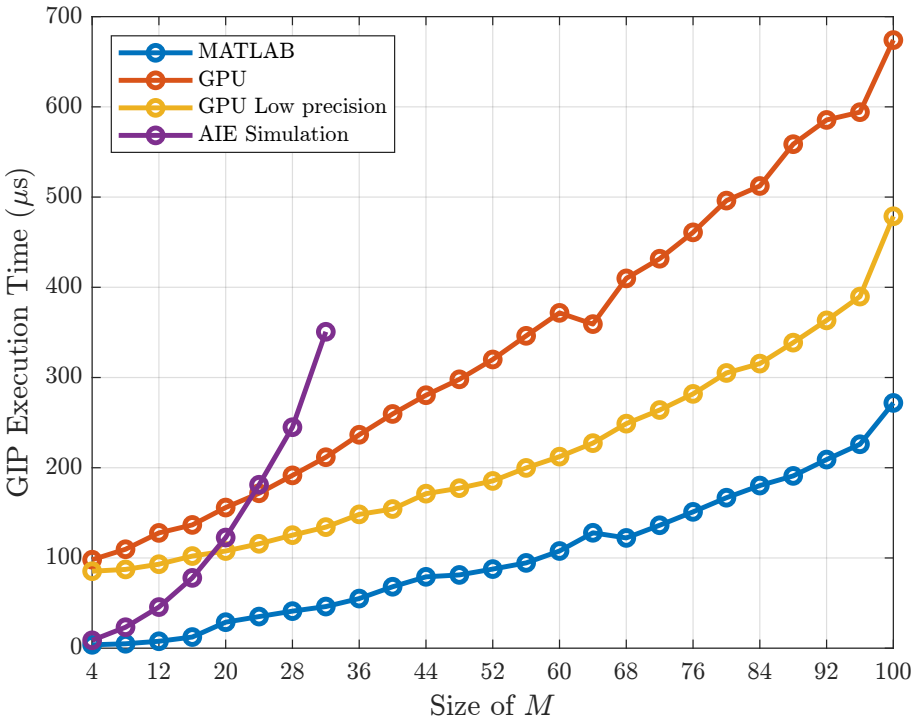


Figure 5.2: Execution time required to calculate a single GIP value for matrix sizes between 4-100.

Figure 5.3 shows the same curves from Figure 5.2, but enlarged between matrix sizes 4 to 32. As seen, the AIE implementation only exceeds the execution time of the GPU for matrix sizes up to 16×16. The low precision GPU implementation surpasses the AIE execution time already at the size of 20×20, while at 24×24 both GPU implementations are faster. The MATLAB implementation does however, outperform all other implementations.

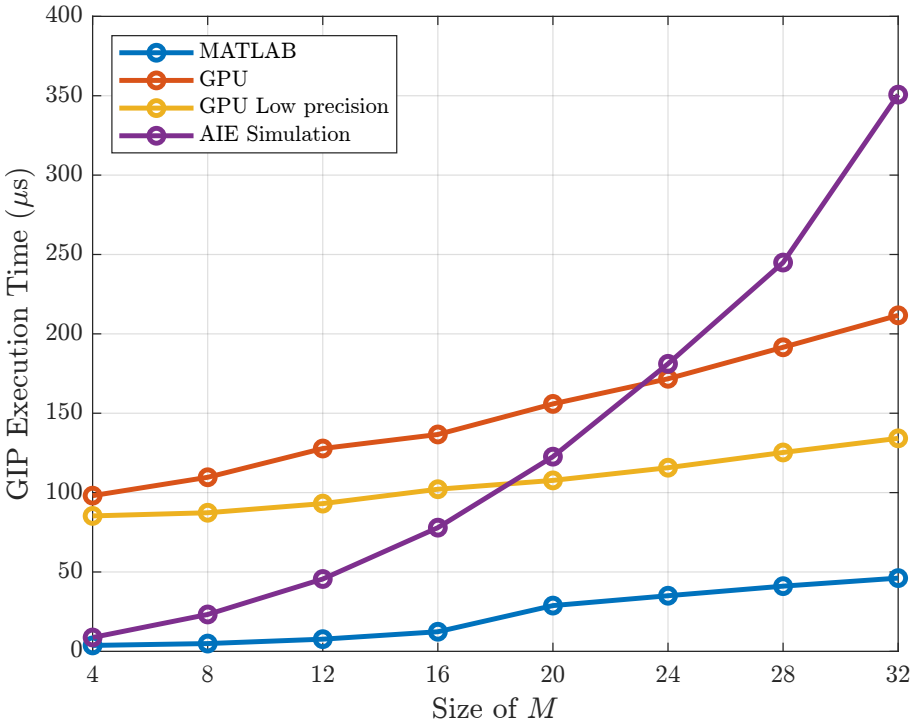


Figure 5.3: Execution time for one GIP-value, matrix sizes 4-36.

5.2.2 Sub-Blocks of a Single GIP Value for the GPU

Figure 5.4 and Figure 5.5 show detailed views of the GPU (non low precision) implementation execution time. Figure 5.4 has divided the execution time into mathematical blocks to overview how much time each step requires. As seen, the dot product scales very poorly in comparison with the matrix multiplication and inversion. Initially, it is actually faster than the matrix inversion, as the size of the matrix increases, the dot product becomes much worse, and in the end, it requires more than 3 times as much time compared to the second slowest part, the matrix inversion. Out of the three, the matrix multiplication is the least time-demanding step in the algorithm, requiring about half as much time as the matrix inversion for all matrix sizes.

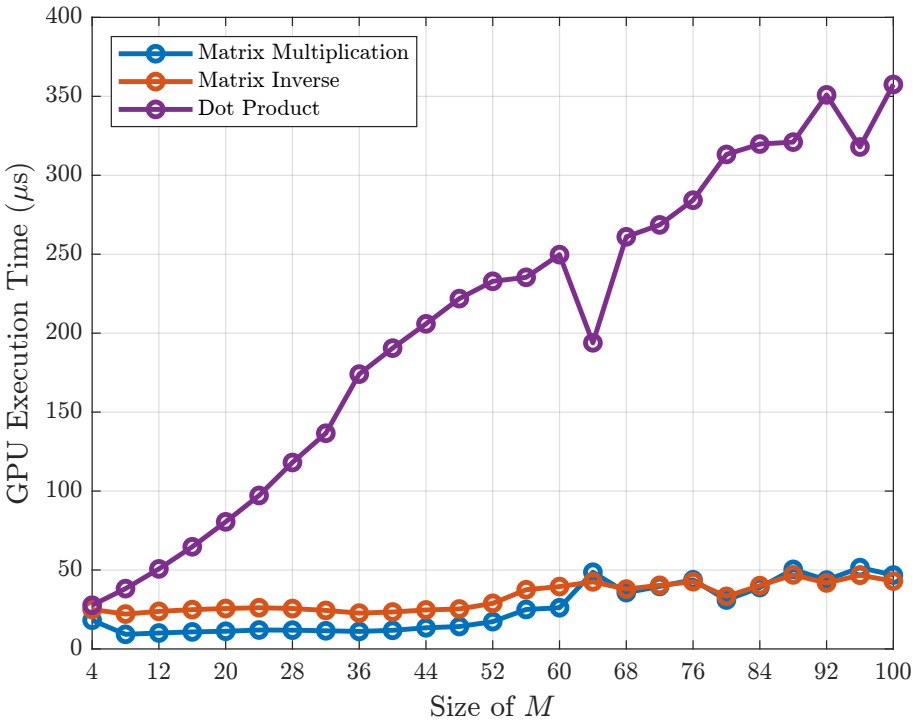


Figure 5.4: GPU execution time for the different parts in the calculation for a GIP value

Figure 5.5 has split the execution time into categories in accordance with the GPU code. The Matrix Multiplication is one function, while the LU-decomposition is another function. The only difference between Figure 5.4 is the matrix inversion being divided into two subgroups, LU-decomposition and forward and backward substitution. The dot product remains much more time demanding compared to all other functions.

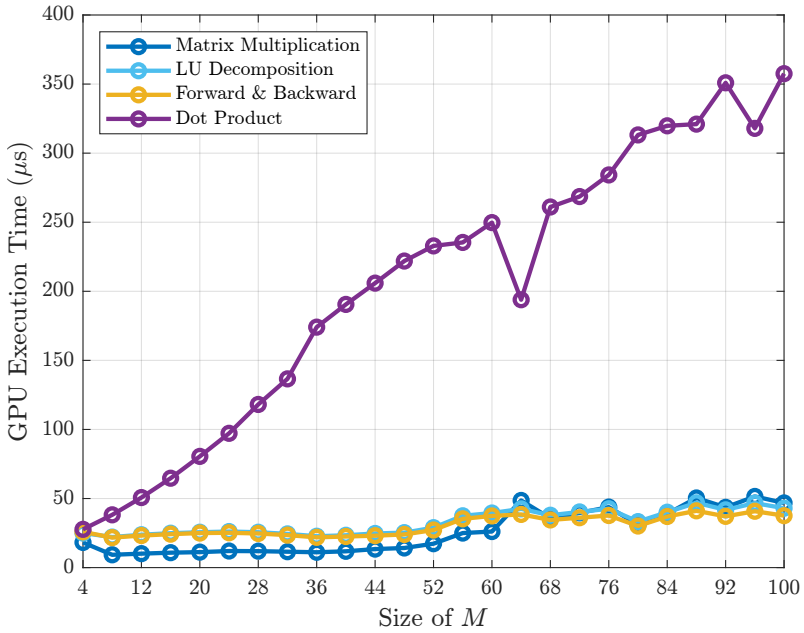


Figure 5.5: GPU execution time for the different parts expsoing the matrix inverse.

Figure 5.6 once again categorises based on the built-in library functions. However, in contrast to Figure 5.5, Figure 5.6 shows how much each function contributes to the total execution time. The graph shows that the dot product outgrows all other functions until a matrix size of 36×36 is reached. Further increase of the matrix size does not increase the dot product's share of execution time as much.

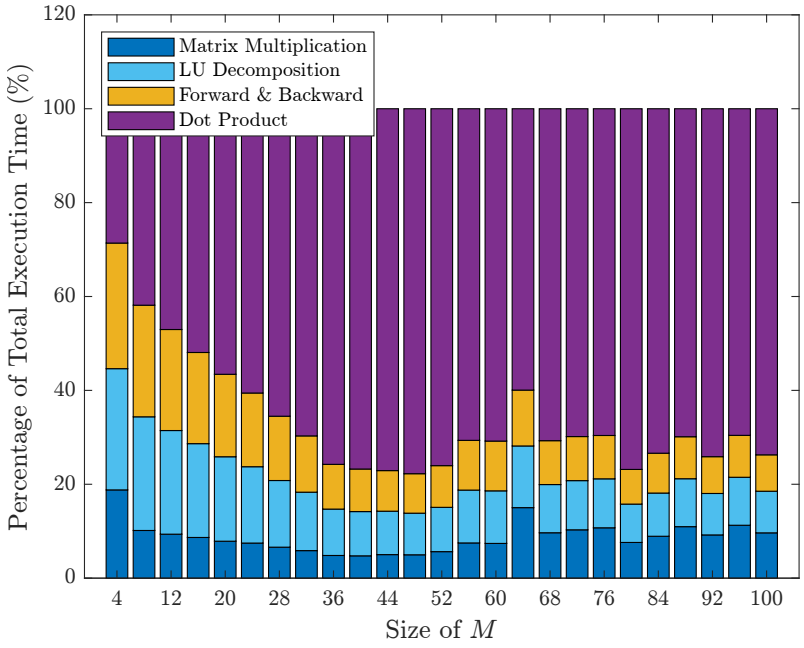


Figure 5.6: The share of total execution time used for different functions in the GPU implementation. Shares for different matrix sizes are spread along the x-axis.

5.2.3 Sub-Blocks of a Single GIP Value for the AIEs

Figure 5.7 shows the execution time for the different steps in the AIE design. Both the matrix multiplication and matrix inversion have a significant increase in execution time as the matrix sizes increase, while the dot product, in contrast, stays relatively constant. In Figure 5.8, the matrix inversion has been divided into two blocks mirroring how the AIE design divided them. Here it is shown that the step increase of the matrix inversion graph in Figure 5.7 comes from the Cholesky, while the forward- and backward substitution have a linear relation with the matrix size. The Cholesky execution time in this graph comes from the start time of the first Cholesky tile to the end of the last Cholesky tile and not the different tiles' execution time added together.

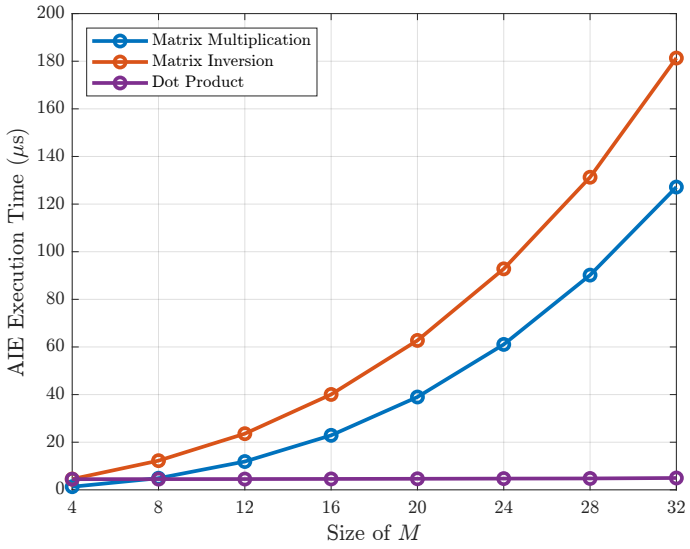


Figure 5.7: Execution time for the different steps.

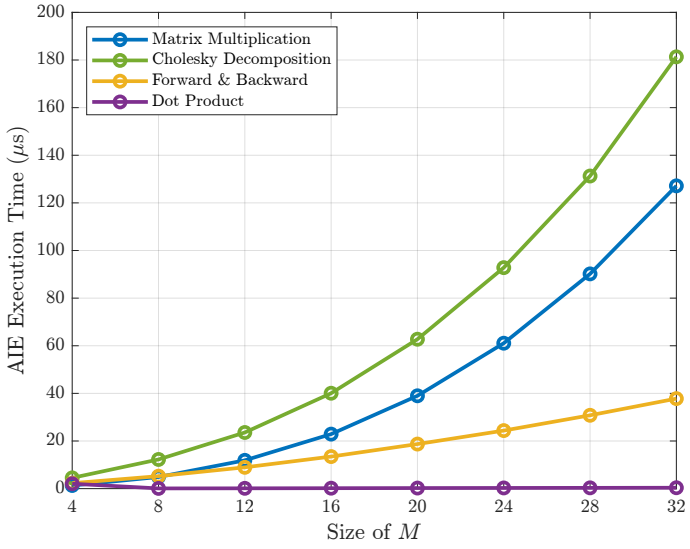


Figure 5.8: AIE execution time for the different steps in calculations of a GIP value.

Figure 5.9 shows the percentage of the aggregate execution time for the blocks. For all sizes, the Cholesky takes the longest time, but the matrix multiplication increases faster in the beginning. The matrix multiplication increases, then slows down when the Cholesky and multiplications start scaling with the same speed. The dot product takes up a large part of the smallest matrix size, but quickly becomes unnoticeable for larger sizes.

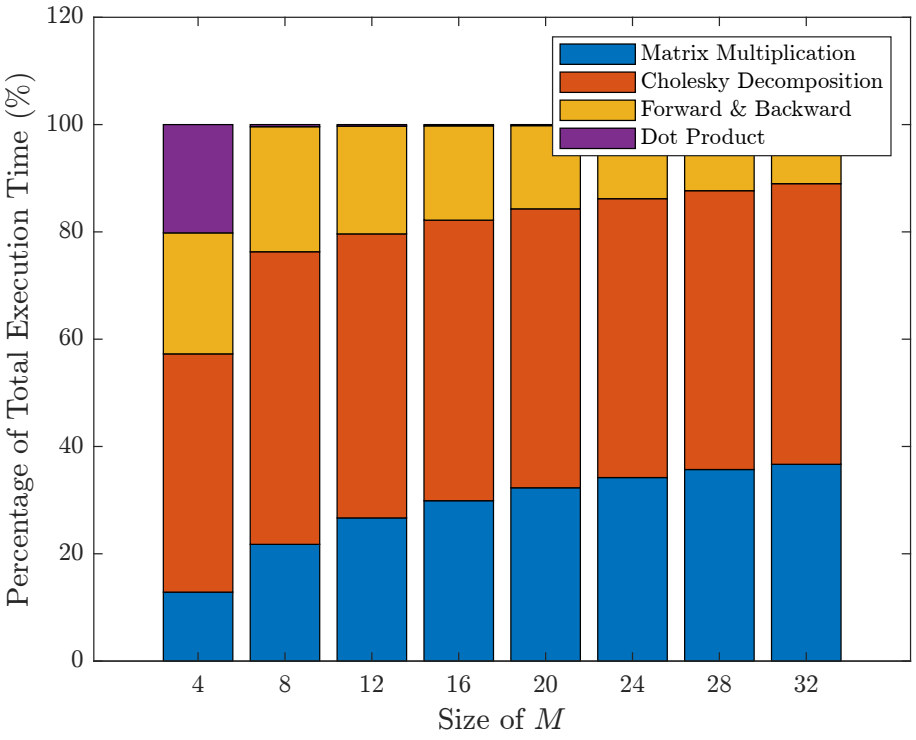


Figure 5.9: Percentage of total execution time the different AIE blocks take.

5.2.4 Full GIP Algorithm

The entire GIP algorithm was implemented on both the GPU and in MATLAB. In Figure 5.10, the execution time for these implementations is shown. MATLAB beats both the GPU implementations across all matrix sizes. In Figure 5.10, the graph is focused on the matrix sizes 4 to 36; here, it is clear that MATLAB is also substantially faster for the small matrix sizes.

For all matrix sizes except 4, the low-precision GPU performs faster than the high-precision GPU for all other sizes. The low precision also scales better than the high precision one. When analysing Figure 5.10, the MATLAB implementation scales worse than the low precision implementation with a sharper increase between sizes 56 and 68. But due to the decrease in execution time at matrix size 76, the MATLAB implementation is faster.

The execution time for all matrix sizes is in the order of seconds, ranging from around two seconds for the GPU up to almost 300 seconds. In radar detection, the time for detection varies due to a number of reasons, but a detection time of five minutes is slow. For a matrix size of 32, all three implementations execute under 35 seconds.

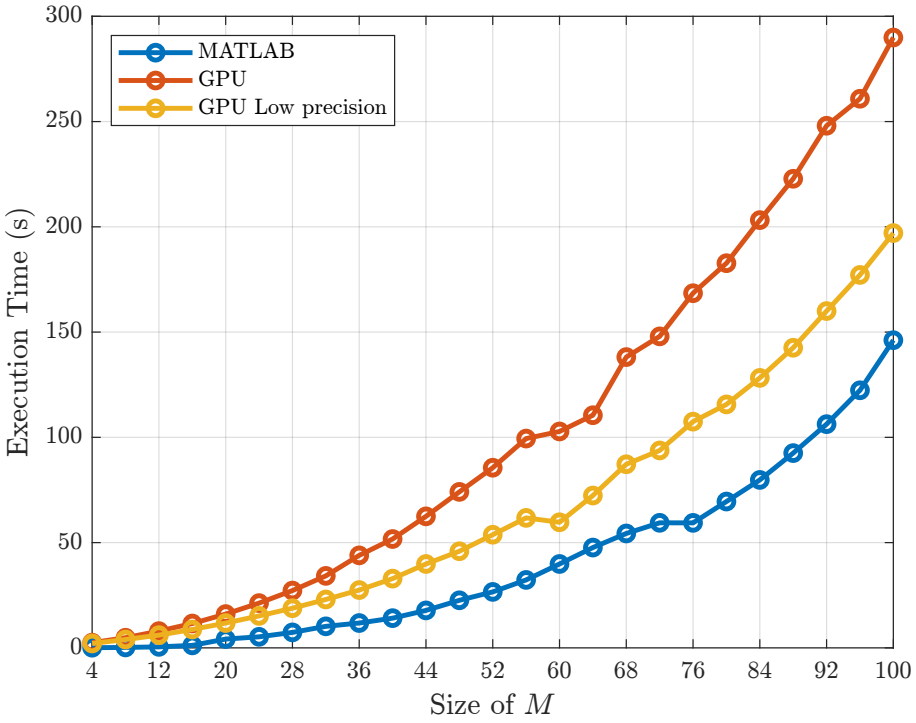


Figure 5.10: Total execution time for the entire GIP algorithm, matrix sizes 4-100.

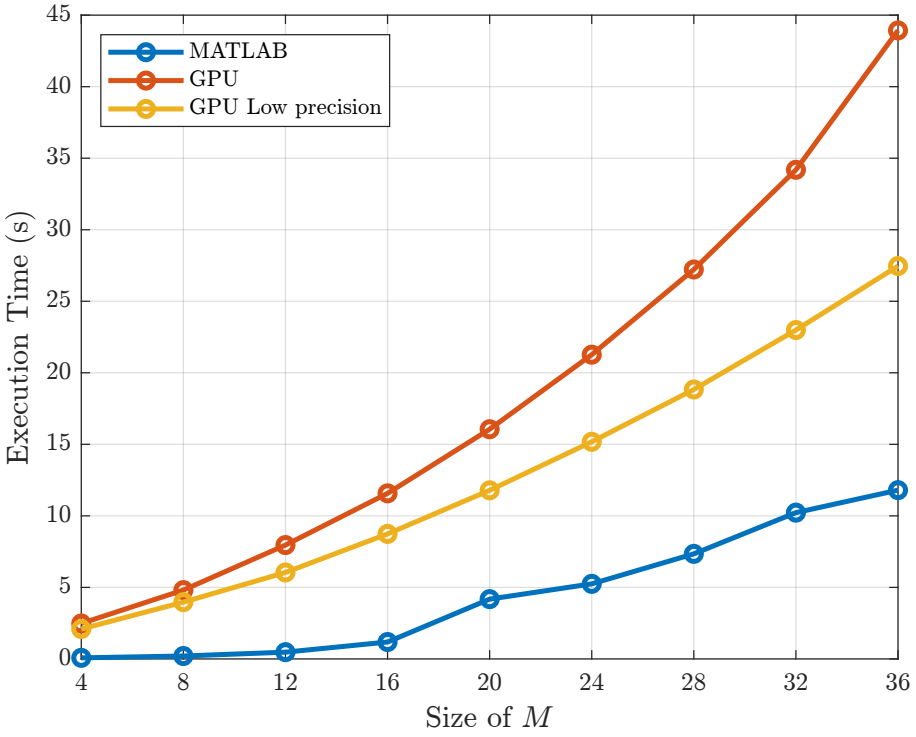


Figure 5.11: Total execution time for the entire GIP algorithm, matrix sizes 4-36.

5.3 AIE Utilisation

Table 5.3 shows how many tiles were needed and what these tiles were used for in the AIE design. The kernel tiles represent the minimum amount of kernel that is required for the current design. These are the tiles with the blocks implemented for the calculations. Since each increase in size adds two extra Cholesky blocks, this is the increase in kernel tiles. The other columns show how many tiles were used for nets, memory and buffers. When building the tool, have the possibility of placing multiple of these on the same tile, which is the fourth and fifth columns. The rightmost column shows the total number of tiles used. Even at the smallest size, this is larger than the number of kernel tiles because three were used for memory.

The total amount of tiles increases linearly until the step from size 20 to 24. At this size, the design needs routing to function, which is seen in the nets going from zero to four. On the next size increase, the tiles go from 24 to 32 because even more routing and memory are needed. For the last size increase, the number of tiles stays the same. This is because, since the number of kernel tiles increases, it needs fewer nets to route the design. But there is also an increase in the necessity of buffers, which could be handled by placing the buffers on the tiles that were already used for nets only.

Table 5.3: The number of tiles and what they are used for, depending on the different matrix sizes.

Size	Kernel Tiles	Nets	Mem.	Nets & Mem.	Nets & Buffers	Tiles
4	5	0	3	0	0	8
8	7	0	3	0	2	12
12	9	0	4	0	0	13
16	11	0	3	0	0	14
20	13	0	5	0	0	18
24	15	4	0	4	1	24
28	17	7	0	7	1	32
32	19	0	7	0	6	32

Table 5.4 shows the program size of the different blocks, with the maximum possible size being 16 kB. When the matrix size becomes larger than 4 there will exist multiple pairs of Cholesky one and two. But the program size will be the largest for the first pair, which is what is displayed in Table 5.4. The program size increases for each size increase for all blocks except for the dot product. The dot product increases until size 20, where it decreases before staying constant. At the smallest size, the Cholesky 1 has the largest program size while Cholesky 2 has the smallest.

Table 5.4: Program size for the different kernel tiles, for Cholesky 1 and 2, only the two largest are shown.

	Program size (bytes)				
Size	**Matrix Mult.**	**Cholesky 1**	**Cholesky 2**	**Fwd- & Bwd Sub**	**Dot**
4	5078	11,636	1344	9556	1722
8	5484	11,882	10,250	10,834	2120
12	9554	12,254	11,498	11,690	2508
16	9746	12,814	11,414	12,840	2996
20	9970	13,470	11,992	12,962	2216
24	10,146	14,332	12,644	13,566	2216
28	10,434	14,622	13,486	14,264	2216
32	10,642	14,612	13,562	14,258	2216

Table 5.5 shows the input buffers for the different blocks all the buffers increase with an increase in matrix size. The largest buffer is the first one, where a $M \times 2M$ matrix must be stored. Both the forward and backward substitution block and the dot product block have two separate input buffers. One comes from the earlier block, while the other is the CUT-matrix input. Since there exist multiple pairs of Cholesky blocks, not all buffer sizes are displayed in the table, but the Total column displays the size of all buffers in the design.

Table 5.5: The input buffer size for the different AIE blocks and the total size of all buffers.

	Input buffer size							
Size	**Matrix mult**	**Cholesky 1**	**Cholesky 2**	**Fwd & Bwd**	**Dot**			**Total**
4	256	128	156	128	32	32	32	3520
8	1024	384	512	384	64	64	64	13,632
12	2304	768	896	768	96	96	96	33,472
16	4096	1280	1408	1280	128	128	128	66,112
20	6400	1920	2048	1920	160	160	160	114,624
24	9216	2688	2816	2688	192	192	192	182,080
28	12,544	3584	3712	3584	224	224	224	271,552
32	16,384	4608	4736	4608	256	256	256	386,112

5.3.1 Parallel Execution of AIE Function

As seen in Table 5.3, the current implementation was not close to utilising all of the 400 available AIE tiles, no matter the matrix size. Therefore, it was tested how many implementations that could be fitted in parallel on the board. It was found that at most, 12 implementations could run in parallel. In other words, twelve GIP values could be calculated in parallel. The total execution time for this parallel implementation was measured to 388.6 μs in the Vitis simulator, and it used a total of 366 tiles, where 228 were kernel tiles. The entire mapping is shown in Appendix B.

6

Discussion

We will in this chapter discuss different aspects of the results. First, the performance of the GIP algorithm will be reviewed followed by suggestions of how it potentially could be improved. After a review of the GIP algorithm, the results regarding the AIE and the GPU implementation will be analysed. Lastly, sections regarding possible improvementes and ethical considerations will follow.

6.1 Potential Changes to the GIP Algorithm

Our implementation of the GIP algorithm uses the GIP matrix. This matrix was constructed by concatenating all the necessary values for calculating the current CUT value. The aim of the GIP matrix is to ease the implementation when wrap-around had to occur at the two ends of the noise matrix. But in our implementation, the GIP matrix was still constructed for each value, even if it did not need wrap-around. This could have been changed by not constructing the GIP matrix when the wrap around was not needed. Since the majority of CUT values do not need the wrap around, the number of computations could be decreased. However, the GIP matrix also helped in handling of the guard cells, which could complicate the implementation.

Another opportunity stems from the nature of the GIP algorithm. When calculating the threshold value for a CUT, four M GIP values are calculated surrounding this cell. When the CUT moves, and the four M cells around it are calculated, there will be a large overlap of cells whose GIP values have been calculated both times. This begs the question: could not all GIP values be calculated in advance and then used when iterating through the thresholds CUTs? Other than available memory, we have not found a reason why this is not used and why it is important to recalculate the GIP values. But changing this could decrease the number of operations needed. There were two reasons why this change was not implemented in this project; firstly, we wanted to follow the literature. The literature used in this thesis described the GIP algorithm as recalculating the GIP values, and this is what we implemented. Secondly, this thesis aimed to evaluate the AIEs, and for the evaluation, a heavier algorithm containing more operations could be preferable, as long as it is identical for all platforms.

6.2 Accuracy of GPU Measurements

From Table 5.1, the difference in numerical accuracy between the two GPU implementations is shown. As seen, the difference in precision is relatively small. In addition, Figure 5.10 shows a rather significant difference in execution time, around ten second for a 32×32 matrix. These two results makes it unclear whether the extra accuracy is worth the extra computation time. The main reason for the difference compared to the MATLAB values arose from the sorting of GIP values. Here, small differences in GIP values could change which training cells were used to calculate the threshold value. This in turn, altered the threshold value.

When analysing Figure 5.5 and Figure 5.2, the total time for a given matrix size is lower than when adding the steps together. This, we believe, stems from how the measurements were conducted. For the total execution time, the start and end time were placed outside a loop completing 10,000 iterations. The total time was then divided by 10,000. But for the different steps, the start and end times were placed around the library functions inside the for-loop, accumulating time before division by 10,000. This means the compiler could have sped up the total execution time by optimising the for-loop for the case when the start and end time were placed outside of it. More importantly, these two measurements were not conducted at the same time. Before measurements started, it was clear that the GPU was not in use, but no check was done for the CPU. This means a busy CPU could also alter the results for these two measurements.

Both GPU implementations were slower than the MATLAB implementation, which continued even as the matrix size increased. Figure 5.4 shows the reason why. Both the matrix operations stay constant when the matrix increases, which was expected from the GPU libraries that were used. But the dot product increases rapidly as the size increases, which creates a bad scaling for the entire implementation. The reason for this is unclear. In Figure 6.1, the MATLAB execution times for the different steps used in the GPU and AIE implementation are shown. Here, the dot product is the fastest, staying constant in relation to the other operations. The same pattern is seen in Figure 5.9, where the dot product quickly disappears in comparison with the other operations. This creates even more uncertainty about the GPU dot product. Could other factors have influenced the result, or was it incorrectly applied? We have not found any other factors, nor have we found alternate ways of applying the functions.

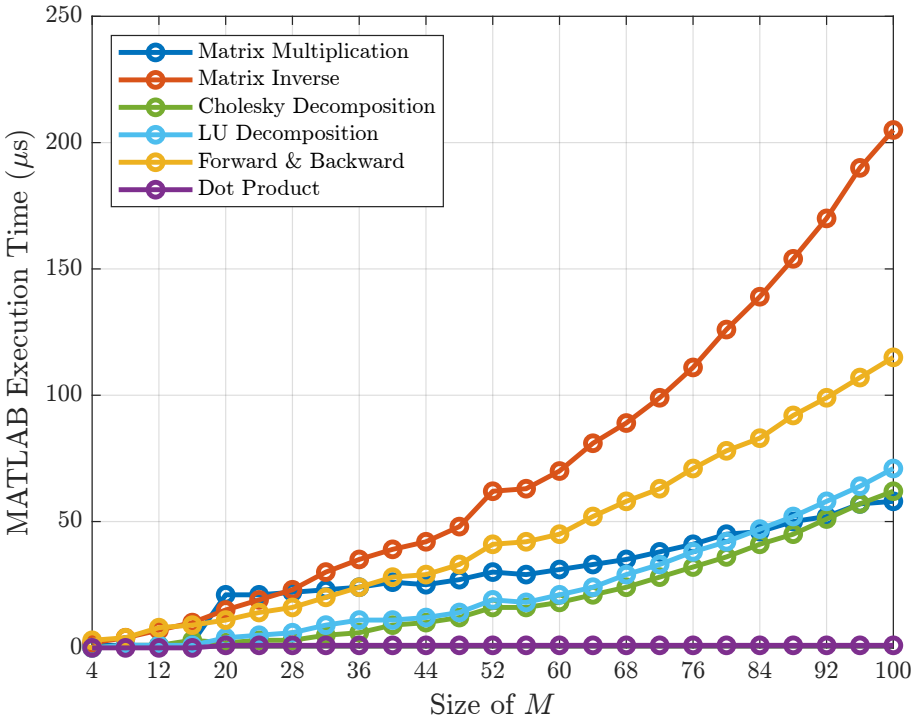


Figure 6.1: MATLAB execution times for different matrix sizes for the different steps used in the implementations

6.2.1 Potential Improvements to the Design Implemented on the GPU

The current two GPU implementations do not have any parallelisation. Instead, the NVIDIA libraries created the parallelisation for the different function calls. This is where the design could be changed. Parallelisation for either multiple GIP values at a time, or multiple threshold values for different CUTs, should be able to halve the execution time if two threshold values are calculated at the same time. This could, however, create some problems. The GPU does not have unlimited resources, and the current results do not analyse how these resources are used. But Figure 5.5 shows that the matrix operations scale well with size. This scaling could be made worse if more resources of the GPU are used for calculating multiple threshold values at the same time.

6.3 AIE Evaluation

Although the actual hardware was never used, there are still things to be learned from the simulation results; not only how the implementation can be improved upon, but also what the potential of the AIEs look like.

6.3.1 Execution Time

Since the AIE implementation was never tested on real hardware, it is not possible to determine whether the resulting execution times are precise or not; they should rather be looked at as a good approximation. However, both the AIE and GPU implementations are clearly slower than the MATLAB implementation as seen in Figure 5.3. This is a clear indicator that both hardware implementations suffer from bad mapping of algorithm to hardware architecture and would require further development before achieving a more expected result. However, Figure 5.3 shows that the AIE implementation holds up relatively well against all other implementations for smaller matrix sizes. It does however, scale the worst of all implementations. Looking at Figure 5.3, it seems as if the execution time of the AIE implementation scales exponentially, while the other three are close to a linear increase in time. This is probably due to MATLAB and the NVIDIA libraries handling the increasing size by parallelising the workload automatically, something that would require manual changes to the code for the AIEs. To get the best performance out of the AIEs should thus be optimised for a specific use case to achieve the optimal performance.

Moving on, Figure 5.7 and Figure 5.8 show that although the implementation is not perfectly optimised, it is still a reasonable implementation. If the MATLAB implementation as seen in Figure 6.1 is assumed to be the gold standard, based on the observations made in Section 5.1, then the AIE implementation is pretty close to the standard implementation in terms of execution time for different parts. The matrix inversion requires the most time to complete, followed by the multiplication and lastly the dot product. This was not the case for the GPU implementation, which shows that the AIE implementation is reasonably designed. This is important when the utilisation and scaling will be analysed.

6.3.2 AIE Utilisation

Table 5.3 and Table 5.4 show how the board utilisation is affected by the increase in matrix size. As seen, the number of kernel tiles increases as the size of the matrix is increased. This fact alone explains to some extent why the execution time increases, because more tiles mean more data transfers. The data transfer was, however, observed to be quite efficient; the largest factor why the execution time increased is simply due to the larger matrices which requires handling. Table 5.4 shows that the program size for the matrix multiplication more than doubles from size 4 to size 32. Cholesky 2 also increases more than tenfold. The large increases are closely related to the increase in execution time, and to decrease execution times further, these blocks would have to be optimised further, for example by parallelising the functions, as the time lost in data transfers is small compared to the amount of time that can be gained from improving the multiplication and Cholesky blocks.

Using more blocks would however decrease the amount of GIP values that can be used in parallel. At the moment, 12 GIP values can be calculated simultaneously for a 32×32 matrix. Parallelising Cholesky 2 and the matrix multiplication might lower that number, thus it might decrease the total potential throughput.

It should be mentioned that it was expected of the AIEs to not lose very much

in terms of execution time. The small time lost, which equals roughly 90 μs for a single GIP value computation, is probably due to the fact that the routing is more restricted for Vitis when many implementations run in parallel. In other words, it may not be possible to lay out the most efficient routing that was created for the single running implementation for twelve implementations in parallel. However, it is still very positive to see such a great increase in throughput for such a small cost in execution time.

6.3.3 Design Improvements

When analysing Figure 5.9, the inverse with the Cholesky and Forward & Backwards added together takes the majority of the execution time. This could lead to the analysis that the Cholesky is the slowest block and should be optimised. But our analysis is that this is not correct. In the Section 4.5.2, it was explained that the Cholesky had to be split into pairs to fit the program size; the number of these pairs was then decided by the matrix size. For a 32×32 matrix, this means the Cholesky is split between 16 tiles. This design choice was necessary but forced an unbalanced pipeline. In Figure 6.2, the execution time for all the different tiles is shown. Here, it becomes clear that the problems with the pipeline are that each of the Cholesky factors becomes faster and faster. But it also shows that the tile that takes the longest time is the matrix multiplication. Here we see two ways to increase the total throughput of the design.

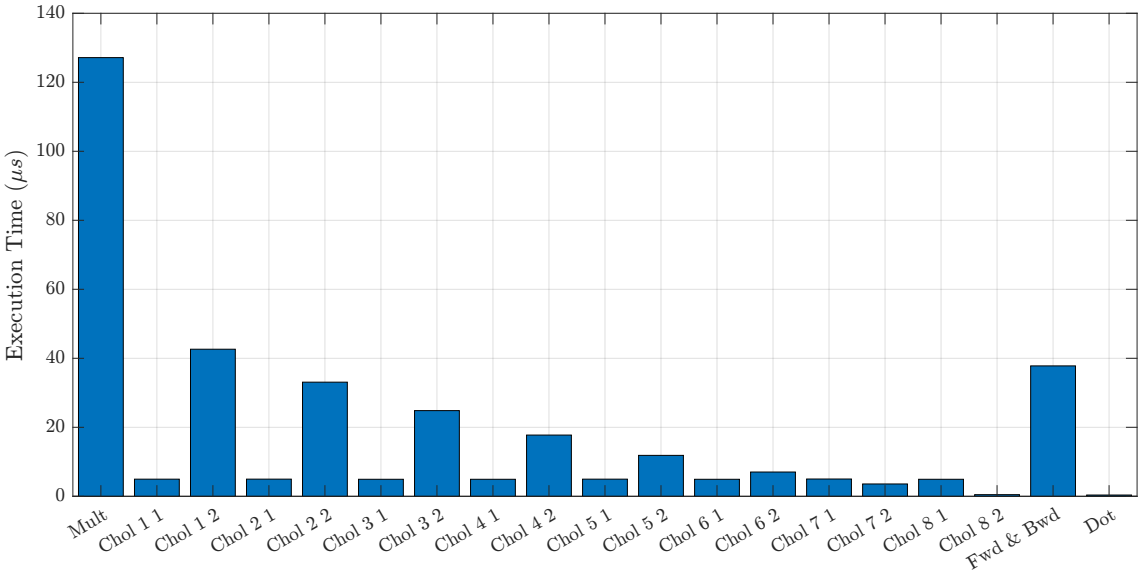


Figure 6.2: Execution times for all the different tiles for a 32×32 matrix size.

The first would be to decrease the time of the matrix multiplication. The first thought here is to use multiple tiles for the calculations. Either splitting it into two parallel computations, which in theory could halve the execution time. The other is to create two sequential tiles, each doing half the operations, which would help the pipeline part of the implementation. To solve the problem with the Choleskys getting faster and faster, one must look into the program sizes of the tiles, the

program sizes are shown in Table 5.4. For each size except the smallest, they do not fit on the same tile. If these could be decreased, multiple steps could be done on the same tile. This would decrease the utilisation and make the execution slower, which would create a better pipeline implementation. We are not sure about why the program size is large for the Cholesky 1, even if the execution times are fast. The only guess is that it uses a lot of floating-point operations from the C++ standard library, for example, the square root function. But it could also be something else.

Using fixed-point precision instead of floating-point precision could potentially decrease the program size of Cholesky 1. If possible, shrinking the design further would increase the potential throughput over the entire board, which is very positive. Further, the usage of a fixed point would also increase the speed of the implementation. The range of the numbers being processed by the implementation is not huge, thus, there could be a lot to gain from using fixed-point instead of floating-point.

6.4 Power Consumption

In Table 2.1, Table 2.5, and Table 2.6, the peak power for each of the three hardware types is shown. Here, the GPU has the highest peak power, with the CPU being the lowest. Here, it is important to note the peak power of the VCK190 board. The board's peak power is for the entire board and not only the AIEs; also, it does not factor in that not all AIEs are being used. This creates problems in the comparison of peak powers between the implementations. The CPU and GPU designs may utilise the entire hardware, but the AIEs only parts of it. Further, it is not fair to compare MATLAB in terms of power consumption, since many other processes are active on the computer running the implementation, which requires power. We therefore refrain from commenting on the energy efficiency of the three implementations and how they compare against each other. This analysis also focused on the peak power from the datasheets of the different hardware types, but a real power consumption analysis must utilise how the power consumption changes over time when the algorithm is executed.

6.5 Improvements of the Methodology

In Appendix C, the GANTT scheme that was laid out at the beginning of this project is shown. The scheme roughly divides the project into three different phases, MATLAB, AIE and GPU, in other words, one phase for each implementation. Initially, the plan was to start designing the MATLAB implementation, then directly start working on the AIE implementation, since it was expected to require the most time.

However, due to practicalities, this plan had to be changed almost immediately because we received access to the GPU environment much sooner than the AIE development environment. Thus, the AIE phase and GPU phase switched places. Looking back, this was not optimal, since more time would have been required to make the AIE implementation work. Moreover, it would also have been preferable to start working on the AIE implementation before the GPU implementation from

a design perspective as well. Design choices that were made for the GPU implementation would necessarily not have been the same if we had a complete AIE design to look at. In retrospect, it would have been better to start working on the AIE design before the GPU, but we were unfortunately not able to control this aspect of the project.

The MATLAB phase also did not work according to the plan. Although the phase was the first one to be executed as planned, much time was still wasted. First and foremost, the coding of the GIP algorithm started too early. More time should have been spent at the early stages of the project on properly reading up on the algorithm. Instead, much time was spent on redesigning the implementations that did not work as they should. At the early stages, there were also confusion about the purpose that CFAR, STAP, and GIP played, which resulted in a bad MATLAB design. This could have been avoided by more properly reading up on the GIP algorithm and radar theory in general.

During the MATLAB phase, unnecessary time was also spent on evaluating the GIP algorithm. Although it was not really the focus of the project, attempts were made to evaluate the GIP algorithm by plotting so-called receiver operating characteristic curves for the algorithm. As the focus of the project was really just to evaluate the hardware, not the algorithm, this sidestep meant time was lost.

In the end, the MATLAB phase required about 2-3 weeks more than what the initial plan had expected. If time had been spent on the correct things, i.e. learning the GIP algorithm properly and not putting time into evaluating it, the initial plan could probably have been held and more time had been left for the AIE implementation, which in retrospect certainly was needed. Further, if the AIE implementation had been initiated before the GPU algorithm, it would have been more obvious how the GPU implementation should have been designed, thus also saving valuable time.

Lastly, not only should the design of the AIE implementation have started earlier, attempts to run things on the Versal board should also have been made much sooner in the project. The design seen in Section 4.4 uses multiple kernels and wiring, attempts only to use a single kernel might have been easier to start with. Making these attempts earlier would also have helped overcome some practicalities which slowed the project down in the final stages.

6.6 Future Work

To further evaluate the AIEs, the energy consumption should be analysed. The comparisons of execution time between the CPU, GPU, and AIE may not give a fair view without the power consumption or a utilisation comparison. All implementations could also be optimised for faster execution times. Both the CPU and GPU could be parallelised using `parfor` in MATLAB and calculate either multiple thresholds or GIP values in the GPU. The AIE would need further analysis for optimisation, but from Figure 6.2 it seems there are multiple ways of optimisation.

The other step to further evaluate the AIEs is to run the actual hardware. Future projects should prioritise getting the hardware up and running early in the project,

maybe by using a simpler algorithm. The GIP algorithm is very demanding, which makes it suitable for performance evaluations. Despite this, it might be badly suited for the evaluation of the AIEs, simply because, from our understanding, the sorting is difficult to perform in the AIEs themselves.

6.7 Societal and Ethical Aspects of the Project

Since Saab is a company that manufactures weapons, it is of great importance to take any ethical dilemmas that is related to the project into consideration. Even though Saab calls themselves a defence company, manufacturing weapons is still the core of the company's business model [33]. Although a radar is not a type of weapon, it can still be used to enhance the efficiency of weapons. Improving radar capability could be seen as two-edged. On one hand, by improving the accuracy and precision of radar systems, accidents were civilians are harmed could be avoided as they are not mistaken for enemy targets. Many radar systems are also used for surveillance and are not directly linked to weapon systems, meaning it essentially helps save lifes of civilian people by issuing warnings of incoming attacks. On the other hand, it could also make it easier for ill-intentioned decision makers to cause damage.

By using export restrictions, the Swedish government is trying to avoid selling weapons to ill-intentioned decision makers [34]. Saab also has its own policy to reassure they work within the UN's guiding principles [35]. The world is unfortunately not only black and white, there are many ethically complex countries all over the globe. Both economical and political incentives can make exports to questionable buyers an attractive option. Such decisions might in the end cause harm to innocent civilians.

Lastly, an argument could be made that by providing local weapons manufacturing, a small country such as Sweden strengthens its sovereignty and thus reduces the risk of being dragged into conflicts. Foreign aggressors will be scared off if a potential attack is seen as too costly. Radar systems also help deter attacks as an attack would be spotted early enough to strike it down. Since military conflicts between countries usually have huge negative impacts, not only on entire societies, but also on surrounding nature and ecological systems, the conflict that never occurs should be considered the best conflict.

6.8 Usage of Large Language Models

This thesis has taken advantage of LLMs. The LLMs have been used for writing smaller code functions, finding bugs and proofreading the reports related to this thesis. All text and all design choices seen in this thesis were created by the authors.

7

Conclusion

In conclusion, this thesis found no direct advantages of using artificial intelligence engines over graphics processing unit for radar signal processing. However, to fully evaluate the artificial intelligence engines, further evaluation is required, especially focusing on the power consumption in relation to the throughput.

The simulated artificial intelligence engine design in this thesis calculated a single generalised inner product value. For small matrix sizes, the execution time for this design was close in speed to the MATLAB design, but scaled worse, being slower than the graphics processing unit implementations at a matrix size of 24×24. The artificial intelligence engine design for a 32×32, which was the largest tested, used 32 tiles in total, with 19 of these being used for kernels.

It is difficult to compare different types of technology with each other, especially when one is simulated and one is run on hardware. Thus, the results of this thesis should be viewed critically. Further studies are required before any real conclusions can be made for the usage of artificial intelligence engine for radar signal processing.

Bibliography

[1] SIPRI Top 100 arms producers see combined revenues surge as states rush to modernize and expand arsenals. Stockholm International Peace Research Institute (SIPRI). Accessed: Jan. 28, 2026. [Online]. Available: https://www.sipri.org/media/press-release/2025/sipri-top-100-arms-producers-see-combined-revenues-surge-states-rush-modernize-and-expand-arsenals

[2] Framtidens autonoma drönarsvärmar – ett växande hot mot civila. Totalförsvarets forskningsinstitut (FOI). Accessed: Jan. 28, 2026. [Online]. Available: https://www.foi.se/nyheter-och-press/nyheter/2025-09-23-framtidens-autonoma-dronarsvarmar---ett-vaxande-hot-mot-civila.html

[3] J. Sullivan, *Radar Foundations for Imaging and Advanced Concepts*. Scitech Publishing, 2004.

[4] Radar CFAR Operator Documentation. NVIDIA. Accessed: Jan. 28, 2026. [Online]. Available: https://docs.nvidia.com/pva/solutions/0.4.0/impl/operator/radarcfar.html

[5] A. Coluccia, *Adaptive Radar Detection*. Artech House, 2022.

[6] M. Rangaswamy, J. H. Michels, and B. Himed, "Statistical analysis of the non-homogeneity detector for STAP applications," *Digital Signal Processing: A Review Journal*, vol. 14, pp. 253–267, 2004.

[7] AMD AI Engine Technology. Advanced Micro Devices. Accessed: Jan. 28, 2026. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/technologies/ai-engine.html

[8] J. H. Michels, B. Himed, and M. Rangaswamy, "Robust STAP detection in a dense signal airborne radar environment," *Signal Processing*, vol. 84, no. 9, pp. 1625–1636, 2004, Special Section on New Trends and Findings in Antenna Array Processing for Radar.

[9] C. Weber and N. Wright, "Real-time Target Detection Using a CFAR Feature Plane in an Embedded System," Master's thesis, Chalmers University of Technology, 2025.

[10] E. Kelly, "An adaptive detection algorithm," *IEEE Transactions on Aerospace and Electronic Systems*, vol. AES-22, no. 2, pp. 115–127, 1986.

[11] Reed, I.S. and Mallett, J.D. and Brennan, L.E., "Rapid Convergence Rate in Adaptive Arrays," *IEEE Transactions on Aerospace and Electronic Systems*, vol. AES-10, no. 6, pp. 853–863, 1974.

[12] Constant False Alarm Rate (CFAR) Detection. MathWorks Inc. Accessed: Apr. 8, 2026. [Online]. Available: https://se.mathworks.com/help/phased/ug/constant-false-alarm-rate-cfar-detection.html

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in Fortan 77*. Press Syndicate of the University of Cambridge, 1997.

[14] Paul S. Dwyer, "The Doolittle Technique," *The Annals of Mathematical Statistics*, vol. 12, no. 4, pp. 449–458, 1941. [Online]. Available: http://www.jstor.org/stable/2235956

[15] Cholesky decomposition. StatLect. Accessed: May 18, 2026. [Online]. Available: https://www.statlect.com/matrix-algebra/Cholesky-decomposition

[16] F. Zhang *et al.*, *The Schur Complement and Its Applications*. Springer New York, 2005.

[17] *VCK190 Evaluation Board User Guide*, AMD.

[18] AMD, *Versal Adapative SoC AI Engine Architecture Manual*, 2025, Accessed: Feb. 20, 2026. [Online]. Available: https://docs.amd.com/r/en-US/am009-versal-ai-engine/Overview

[19] Zynq 7000 SoC Technical Reference Manual (UG585). AMD. Accessed: June 3, 2026. [Online]. Available: https://docs.amd.com/r/en-US/am009-versal-ai-engine/AI-Engine-Array-Features

[20] Versal Adaptive SoC Design Guide (UG1273). AMD. Accessed: June 3, 2026. [Online]. Available: https://docs.amd.com/r/en-US/ug1273-versal-acap-design

[21] AMD Versal™ AI Core Series VCK190 Evaluation Kit. AMD. Accessed: Feb. 20, 2026. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/vck190.html#tabs-a75507a83a-item-df61ba4d87-tab

[22] AI Engine API User Guide. AMD. Accessed: Apr. 9, 2026. [Online]. Available: https://download.amd.com/docnav/aiengine/xilinx2024_1/aiengine_api/aie_api/doc/index.html

[23] AMD, *AI Engine Tolls and Flows User Guide*, 2025, Accessed: May 28, 2026. [Online]. Available: https://docs.amd.com/r/en-US/ug1076-ai-engine-environment/Simulating-an-AI-Engine-Graph-Application

[24] D. Nagpal, *Computer Fundamentals*. S.Chand Publishing , 2015.

[25] Intel Core Ultra 9 Processor 285H. intel. Accessed: May 28, 2026. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/241747/intel-core-ultra-9-processor-285h-24m-cache-up-to-5-40-ghz/specifications.html

[26] NVIDIA, *CUDA Programming Guide*, 2026, Accessed: Mar. 24, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html

[27] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[28] NVIDIA QUADRO RTX 6000. NVIDIA. Accessed: Mar. 25, 2026. [Online]. Available: https://www.nvidia.com/en-gb/products/workstations/quadro/rtx-6000/

[29] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture*, 2007, Accessed: Mar. 24, 2026. [Online]. Available: https://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[30] ——, *cuBLAS*, 2026, Accessed: Mar. 24, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[31] ——, *cuSOLVER API Reference*, 2026, Accessed: Mar. 24, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cusolver/index.html#

[32] "1 - introductory material," in *Computer Solution of Large Linear Systems*, ser. Studies in Mathematics and Its Applications, G. Meurant, Ed. Elsevier, 1999, vol. 28, pp. 1–68. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168202499800022

[33] Operations and employees on all continents. Saab. Accessed: Jan. 28, 2026. [Online]. Available: https://www.saab.com/about/company-in-brief/organisation

[34] Svensk exportkontroll,. Regeringskansliet. Accessed: Jan. 28, 2026. [Online]. Available: https://www.regeringen.se/regeringens-politik/utrikes--och-sakerhetspolitik/svensk-exportkontroll

[35] Saab AB, *RESPONSIBLE SALES POLICY*, Linköping, Sweden, 2023.

A

Appendix 1

Algorithm 4 GIP algorithm for one pulse index

Ensure: Z ▷ This is the noise

$K \leftarrow 2M$
while $i \leq$ range bins **do**
  $c \leftarrow Z(i)$
  idxLeft $\leftarrow i - n - 3M : i - n$
  idxRight $\leftarrow i + n : i + n + 3M$
  for all idxLeft < 0 **do**
    idxLeft $\leftarrow$ idxLeft + range bins
  end for
  for all idxRight $>$ range bins **do**
    idxRight $\leftarrow$ idxRight − range bins
  end for
  $G \leftarrow Z([\text{idxLeft}, \text{idxRight}])$
  for $l \leftarrow M, \text{size}(G) - M$ **do**
    $x \leftarrow G(l)$
    idxLeft $\leftarrow l - M - n : l - n$
    idxRight $\leftarrow l + n : l + M + n$
    for all idxLeft < 0 **do**
      idxLeft $\leftarrow$ idxLeft + size(G)
    end for
    for all idxRight $>$ size(G) **do**
      idxRight $\leftarrow$ idxRight − size(G)
    end for
    $T_D \leftarrow G([\text{idxLeft}, \text{idxRight}])$
    $R \leftarrow T_D \times T_D^H / K$
    $P(l - K - n) \leftarrow x^H \times R^{-1} \times x$
  end for
  idx $\leftarrow$ sort($|P - E|$) ▷ Sort the values but save the indicies
  idx $\leftarrow$ idx($1 : K$)
  $T_D \leftarrow G([\text{idx}])$
  $S \leftarrow T_D \times T_D^H$
  $t(i) \leftarrow c^H \times S^{-1} \times c$
  $i++$
end while

B

Appendix 2

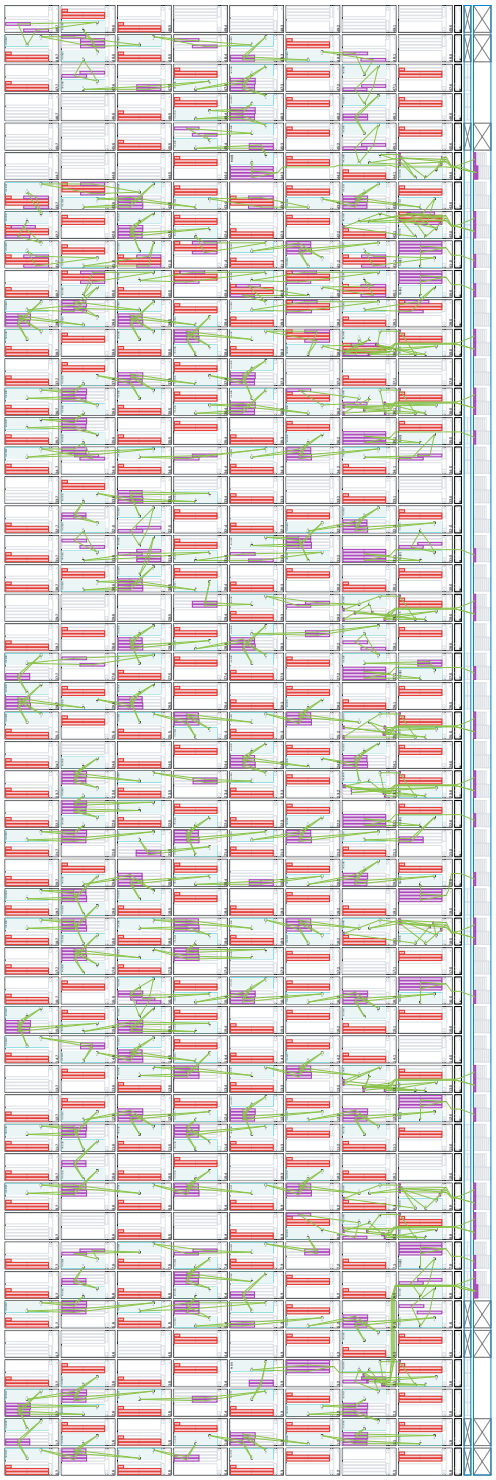


Figure B.1: AIE array for calculating 12 GIP values simultaneously. The grey cells are not in use for the current implementation.

C

Appendix 3

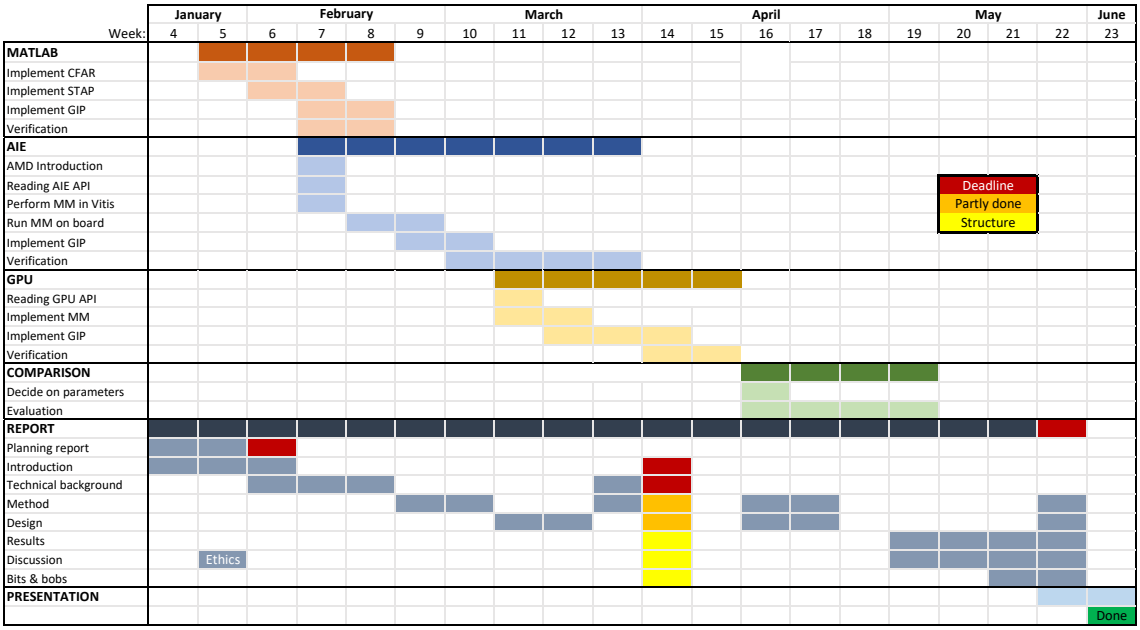


Figure C.1: GANTT chart of the project's time plan.