UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Neural Processing Units to Accelerate Mathematical Models:
A Case Study on a Dynamic Stochastic General Equilibrium Model

VÍCTOR JIMÉNEZ RUGAMA

Thesis Supervisor
MIQUEL MORETÓ PLANAS (Department of Computer Architecture)

Degree
Bachelor's Degree in Informatics Engineering (Computer Engineering)

Bachelor's Thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Abstract

Mathematical models have taken a crucial role in the advancement of scientific research across a wide range of domains — from economics to biology. While these models are essential, they are often computationally intensive, requiring long execution times and consuming significant energy. Accelerators such as GPUs have been repurposed to accelerate these workloads, despite not being originally designed for such tasks.

In response, hardware architects have developed specialized architectures tailored to these needs. Among them, NPUs have emerged as accelerators for matrix-heavy workloads. NPUs not only offer substantial parallelism, but also offload computation from the CPU.

Despite their growing relevance, there is a lack of in-depth analysis on the computational characteristics of NPUs. Little is known about which workloads benefit most from NPUs, or how their architectural features influence performance across different scenarios.

This project addresses the gap by exploring the use of NPUs as accelerators for scientific models. In specific, we take an example economic model and accelerate it through an AMD NPU. We analyze the performance across various strategies and compare the results against scalar CPU and AVX + Multithreaded implementations. The study offers insight into the strengths and limitations of NPUs for scientific computing.

Keywords: Neural Processing Unit, NPU, MLIR-AIE, AMD XDNA, AMD AIE, Artificial Intelligence Engine, Matrix-Matrix Multiplication, Matrix-Vector Multiplication, BF16, AVX, Multithreading, SIMD, Vectorial Extension, High Performance Computing, Workload Characterization, Scientific Computing, Performance Analysis, Parallel Processing.

Acknowledgments

I would like to begin by expressing my sincere gratitude to my advisors at the University of Colorado Boulder, Professor Eric Keller and Professor Tamara Lehman, for their support and guidance throughout the research process. Their mentorship has been crucial in shaping this project and has provided me with invaluable experience that will serve me useful in my PhD studies.

I am also deeply thankful to my advisor at the Universitat Politècnica de Catalunya, Professor Miquel Moretó, for overseeing the formal aspects of the thesis and for his evaluation of my work.

On a personal note, I would like to thank my wife for her emotional support, which has been a constant source of strength. I am equally grateful to my parents and my sister for their daily encouragement and wisdom, and to my brother for both his support and the insightful ideas he offered when I felt stuck.

Contents

List of Figures

List of Tables

1 Introduction & Context

1.1 Introduction

The design of a device such as a Central Processing Unit (CPU) can handle general program-execution related tasks. Likewise, the design of a Graphics Processing Unit (GPU) is optimized for graphics-related tasks. A CPU is well-suited to run the logic behind an operating system or an office program, while a GPU can handle the specific needs of a video game or a high-resolution movie[1]. However, these devices are designed with specific purposes in mind. That is, both CPUs and GPUs are designed to handle a broad range of tasks in their respective domains, while being inefficient at other areas.

One of these inefficient areas is machine learning. Machine learning models, such as the emerging Large Language Models, are not well-suited to run on traditional CPUs. While they can significantly improve their runtime on GPUs[2], these are not designed with Artificial Intelligence (AI) in mind, incurring in high energy consumption[3]. Thus, the use of specialized hardware for AI tasks could decrease the computation latency, while also increasing the energy efficiency.

A solution could be to design an application-specific integrated circuit (ASIC) for each model, which will yield optimal results. However, its inflexibility and high cost makes it an inviable option for most applications, *i.e.*, just the physical tape-out of a single ASIC might cost up to $300 million[4].

In light of the aforementioned, Neural Processing Units (NPU) aim to efficiently accelerate the computation of AI. Since AI models themselves mimic the computation of the human brain, a device that physically mimics the behavior of the human brain will be optimally suited to run AI models, with NPUs fulfilling this role[3].

This Bachelor's Thesis is performed at the Universitat Politècnica de Catalunya under D-modality. The research is conducted at the Department of Computer Science in the University of Colorado Boulder (UCB). In specific, this project is conducted as a collaboration between the UCB Computer Systems Laboratory, UCB Computer Architecture Laboratory, and Prof. Alessandro Peri, from the Departments of Economics. This project is inspired by the work of *Cheela, DeHon, Fernández-Villaverde, and Peri (2025)* [5], who implemented the *incomplete markets, heterogeneous agent model with aggregate uncertainty of Krusell and Smith (1998)* [6] on a Field-Programmable Gate Array (FPGA). With their approach, the runtime on the FPGA was reduced to 7 minutes from the original 8 hours on a single-core CPU. The aim of this project is to use the newer NPU technology to accelerate *The Economic Geography of Global Warming economic model by Cruz, José-Luis and Rossi-Hansberg, Esteban (2021)*[7].

1.2 Concepts

The following concepts are fundamental to understand the project and how the proposed solution can improve upon previous developments.

1.2.1 Stochastic Integrated Assessment Model

A Stochastic Integrated Assessment Model (Stochastic IAM or SIAM) is a macroeconomic model that combines insights from multiple disciplines to analyze the interaction between economic behavior and climate change. Unlike deterministic models, stochastic IAMs incorporate uncertainty by accounting for the random variability of key parameters. They typically consider a wide range of societal factors, including education, health, infrastructure, and governance. These models are primarily used to evaluate the environmental impact of economic policies and support informed decision-making[8][9].

In specific, this project will implement *The Economic Geography of Global Warming model by Cruz, José-Luis and Rossi-Hansberg, Esteban (2021)*[7].

1.2.2 Central Processing Unit

The Central Processing Unit (CPU) is a hardware device with general-purpose computing abilities. It is often referred to as the *brain* of the computer. On the one hand, the CPU is compatible with virtually all software applications, which provides integration with existing systems. Also, it is versatile enough to handle diverse workloads. On the other hand, it has very limited parallelism, which poses a significant bottleneck for applications such as AI or graphics computation[10].

1.2.3 Graphics Processing Unit

The Graphics Processing Unit (GPU) is a hardware device designed to render graphics, such as video games or high-quality videos. Unlike CPUs, which have a handful of cores, GPUs have thousands of cores optimized for parallel computing. This incurs in GPUs having high parallel processing power. In addition, workloads can be easily scaled by distributing them across GPUs. While GPUs excel at parallel tasks, they are not efficient at sequential or single-threaded applications, which makes them unsuitable for applications that a CPU would excel at[10]. Also, GPUs tend to have significant power consumption[3].

1.2.4 Neural Processing Unit

The Neural Processing Unit (NPU) is a hardware device tailored to perform machine learning computations. Similar to how a GPU is designed to perform graphic-related operations with low latency, an NPU is designed to accelerate Artificial Intelligence (AI) workloads[11]. In this sense, it is capable of accelerating the processing and training of machine learning models, while minimizing the power usage. Its low power consumption also makes it ideal for environments such as battery-powered devices or IoT applications. However, its narrow purpose renders it unsuitable for other tasks[10].

1.2.5 AMD XDNA

AMD XDNA is an NPU microarchitecture developed by AMD, first introduced in 2023. This architecture consists of a matrix of Artificial Intelligence Engines (AIE) with a re-programmable interconnection. Each AIE consists of a Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) vector processor optimized for machine learning and advanced signal processing applications[12][13].

Currently, AMD is integrating NPUs into its CPUs, enabling them to offload certain tasks such as video or audio processing while also accelerating them through the NPU. Also, some AMD Xilinx Field-Programmable Gate Arrays (FPGA), such as the Alveo V70 or the Versal AI Series, are incorporating NPUs to accelerate AI tasks that the reprogrammable logic cannot efficiently handle[13].

Figure 1 shows the structure of AMD's XDNA AIE Array. The AIE Array is comprised of several components that will either compute, move, or store data. These components are:

- **Compute Tile:** A compute tile performs the computations and is the equivalent of a core in a CPU. It consists of a compute core to perform operations, a small local memory (L1) for variable reuse, and a Data Memory Accelerator (DMA) to quickly move data in and out of the tile.
- **Memory Tile:** A memory tile acts as a larger shared temporary storage area for frequently used data. Its elements are: the memory that stores the data (L2) and a DMA to quickly move data in and out of the tile.
- **Shim Tile:** A shim tile acts as an adapter to move data in and out of the AIE Array with the External Memory. It only consists of a DMA.
- **Switch:** A switch or interconnect node is a physical component that connects the different tiles in the AIE array to pass the information.

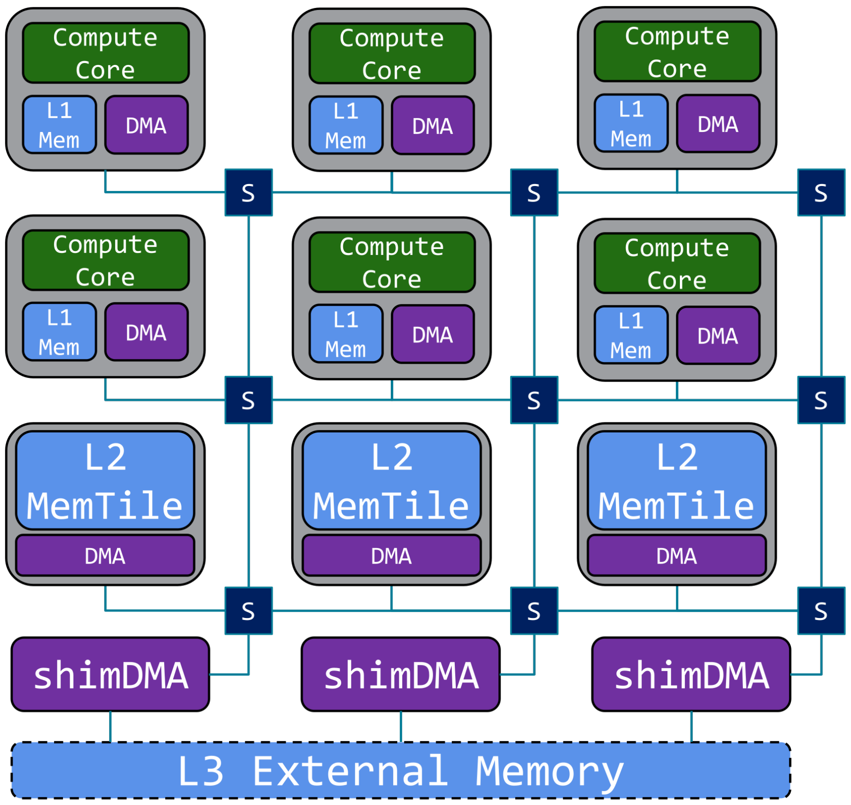


Figure 1: AMD XDNA Structure
[14]

1.3 Programming Languages & Programming Concepts

Several programming languages and concepts are used throughout the development of the project. Different sections of the NPU require different programming languages. All of them are put together during compile time to generate the binary loaded onto the NPU. The following are the programming concepts used during the development of the project:

Multi-Level Intermediate Representation: Multi-Level Intermediate Representation (MLIR) is an intermediate representation language for compilation intended to support a wide variety of programming languages and hardware. Its primary goal is to have a common representation that high-level code can be compiled into, while also facilitating compilation and optimization for any hardware architecture. For instance, a C++ application can be compiled into MLIR. With this same MLIR compilation, an optimized binary can be produced for any target architecture, be it a specific CPU, GPU, or FPGA. Specifically, this paper focuses on MLIR-AIE, which is a version of MLIR for AMD's AI hardware[15].

The following programming languages are used during the development of the project:

C++: C++ is a high-level programming language that was created at the beginning of the 1980s. It was initially created as an object-oriented extension of the C programming language and has significantly expanded over time. C++ was designed with performance and efficiency in mind, while also having the strengths of a higher level programming language. This allows C++ to build software infrastructure while mantaining performance. C++ is a compiled language with the main compilers being LLVM, CLANG, and GCC[16].

Python: Python is a high-level general-purpose programming language, first released in the early 1990s. Python is designed with readability and ease of use in mind. It is dynamically type-checked and garbage-collected, while supporting multiple programming paradigms, such as procedural or object-oriented[17].

1.4 Problem Definition

Pioneer research fields, like machine learning, run their models on optimized hardware architectures, while the field of economics relies on much simpler programming frameworks, such as single-threaded C++ programs. These models are usually highly parallelizable but not optimized. In specific, SIAM models have a lot of input data and are highly computationally expensive on sequential CPUs. As a result, SIAM models can take hours — even days — to complete on a single core CPU[5].

While multi-threading on multicore CPUs can significantly reduce this duration[5], this would still not be completely optimized. DSGE models involve a considerable amount of matrix multiplication operations, which can benefit from vector processing units with Single Instruction Multiple Data (SIMD) instructions. Although modern CPUs offer parallel features like SIMD that could greatly improve model execution runtime and power efficiency, using these features requires specialized knowledge in computer architecture and high performance programming. Unfortunately, economists often lack the proficiency in these areas, making it difficult for them to take full advantage of such optimizations[5].

It is clear that there is a knowledge barrier that prevents economists from taking advantage of hardware features to optimize their models' performance. Here is where the project fits in, by aiming to improve model execution efficiency without requiring high performance programming expertise.

1.5 Stakeholders

The main beneficiaries of this project will be economists, who will be able to run their models in a shorter time without needing to become high performance computing experts. By running macroeconomic models more efficiently, they can significantly boost their research productivity and increase the output of published work.

In a broader sense, the goal of this project is to help experts from scientific fields, such as biology and environmental science, who could benefit from hardware acceleration but lack advanced high-performance computing expertise. It will enable them to develop and deploy models more efficiently, which has the potential to enhance the overall productivity of the scientific community.

Finally, the two research teams involved in this project are the UCB Computer Systems Laboratory and the UCB Computer Architecture Laboratory. Both teams are interested in exploring how NPUs can contribute to scientific model acceleration. The key members of this project are:

Dr. Eric Keller: Advisor for the Computer Systems Laboratory.

Dr. Tamara Lehman: Advisor for the Computer Architecture Laboratory.

Dr. Miquel Moreto: Tutor at the Universitat Politècnica de Catalunya.

Dr. Alessandro Peri: Advisor from the Economics Department.

Victor Jimenez: Author and developer of this project.

1.6 State of the Art

The lack of high-performance computing expertise among macroeconomic researchers results in a limited number of studies on macroeconomic model acceleration. Aside from a few papers, such as those by *Duarte, Duarte, Fonseca, and Montecinos (2019)*[18] and *Peri (2020)*[19], there has been little research in this area.

The research has been pioneered by *Cheela, DeHon, Fernández-Villaverde, and Peri (2025)*[5], which implements the *incomplete markets, heterogeneous agent model with aggregate uncertainty of Krusell and Smith (1998)* [6] on an FPGA. In addition, it includes both single-core and multicore implementations, which are compared to the FPGA in terms of runtime and energy consumption.

	FPGA	CPU Cores		
N.		1	8	48
Exec. Time (min:s)	7:11	474:25	60:57	10:14
Energy (kJ)	13.70	227.62	233.90	235.56

Table 1: State of the Art Execution Comparison
Adapted from *Cheela, DeHon, Fernández-Villaverde, and Peri (2025)*[5]

Table 1 shows the results obtained. As it can be observed, a normal single-core execution can take upwards of 474min, while a multicore execution reduces this time to 60min. However, this is still much higher than the 7min achieved with an FPGA. It is clear that an FPGA implementation is orders of magnitude more efficient, both in terms of speed and energy. This is due to the reprogrammability of the FPGA. As explained in the original article, although the FPGA operates at a much lower frequency, its reprogrammability allows for a hardware implementation of the model, making the execution significantly faster.

1.7 Rationale & Motivation

There has been growing research into optimizing macroeconomic models through hardware accelerators. The previous research has investigated an FPGA implementation and a multicore implementation. Even though the results with a 48-core CPU on Table 1 are promising, it is noticeably slower than the FPGA implementation. Furthermore, the 48-core power consumption is more than 18 times that of the FPGA. The price of the CPU is also prohibitive, costing $10,000[1].

While the FPGA results are the most interesting, non-AI FPGAs are not well-suited for performing matrix multiplication operations, which are the computational bottleneck of economic models. This suggests that there is still significant room for optimization. In addition, FPGA compilation can take hours and consume a significant amount of energy, which would defeat the main purpose of the model acceleration. The runtime would be very low, yet the time and energy consumption during compilation would offset its benefits. Like the 48-core CPU, the FPGA is also expensive, costing $15,000[2].

In light of the aforementioned, I have decided to research the implementation on an NPU. NPUs are an emerging technology, with the latest Intel and AMD personal use CPUs incorporating them. This results in not needing to rent or purchase extremely expensive hardware. Additionally, they are designed for efficient matrix multiplication, which helps resolve the bottleneck of the economical model. Finally, AMD provides MLIR-AIE, a Python/C++ development framework for their AIE. This framework has a short compilation time — seconds or minutes — compared to that of the FPGA — hours or days. In addition, developing in plain Python/C++ makes development easier for non-experts in high performance computing.

[1]https://www.xbyte.com/products/cat-16056/?srsltid=AfmBOopmp_OTPisdjBclTyr-HYoXNNQC2U43jWQY2AZ1mdXexafVZgRH

[2]https://www.mouser.com/ProductDetail/BittWare/XUPP3R-0060?qs=vmHwEFxEFR%252BtrrHI%252BtF3XQ%3D%3D

2 Scope of the project

2.1 Project Objectives

As previously mentioned, macroeconomic models often lack optimization due to the high barrier of entry in high-performance computing. The overall objective of this project is to determine whether NPUs can serve as a viable solution for economic model deployment, in terms of power efficiency, performance, and accessibility. To achieve this, the following sub-objectives have been set:

1. Develop an understanding of the computational requirements of macroeconomic models.
2. Analyze NPU performance for economic model deployment. In specific, NPUs should maximize computing speed.
3. Investigate the feasibility of developing economic models on NPUs. Development on an NPU should have little to no overhead for economic researchers, enabling them to develop models without requiring specialized training.
4. Assess the broader implications of NPU-based model deployment by providing a holistic analysis of whether NPUs can foster the development of generic scientific models.

2.2 Requirements

2.2.1 Functional Requirements

Besides the specific project objectives, the final model implementation must meet the following functional requirements to be considered successful:

- **Correctness:** The NPU implementation of the model must work as expected, producing the expected output for any given input.
- **Compatibility:** The implementation should be compatible with a AMD's NPUs.

2.2.2 Non-Functional Requirements

The following are the non-functional requirements of the product. These are requisites that are not directly associated with how the system works, but they have to be taken into account during the development regardless:

- **Usability:** The process of implementing a macroeconomic model on an NPU should be feasible for economics researchers. As previously mentioned, the burden of having to learn high-performance computing prevents them from optimizing their models.
- **Performance:** The NPU implementation should demonstrate improvements over CPU and AVX+Multithreading implementations. In specific, it should reduce execution time and CPU load, achieving a balanced improvement across both metrics.

- **Reusability:** To maximize the project's impact, the NPU model implementation should be easily usable in other STEM fields. Ideally, researchers without expertise in high-performance computing should be able to deploy accelerated models efficiently.

2.3 Risks & Possible Roadblocks

Any technical project can be impacted by unforeseen events. However, there are some events we can expect to happen with a certain degree of probability. The following are some of the most likely setbacks that we may encounter:

- **Compatibility Issues:**
  - **Hardware Incompatibility:** Since NPU technology is relatively new, the hardware we are using may not yet be supported.
  - **Software Incompatibility:** Similarly, the software we are using, such as the operating system, might face compatibility issues with the NPU.
- **Implementation Errors:**
  - **Bugs:** There might be bugs in the model's implementation that could delay the development.
  - **Timing Violations:** Although we expect improved results, the NPU implementation might be worse than the AVX+Multithreading implementation, or even the CPU implementation altogether.
- **Planning Errors:** Incorrect planning can delay the development of certain aspects of the project. Therefore, it is advisable to set conservative goals and timelines.

2.4 Methodology

The first set of tasks will consist of setting up the software tools and resolving any compatibility issues. Additionally, understanding how to implement the macroeconomical model and use the NPU development tools will be key to the project's success. These initial tasks are primarily individual responsibilities and are not necessarily part of the project's main development phase.

2.4.1 Agile Methodology

Once the setup is complete, the actual development of the project will begin. For this, the agile development methodology will be followed. As shown in Figure 2, this methodology consists of:

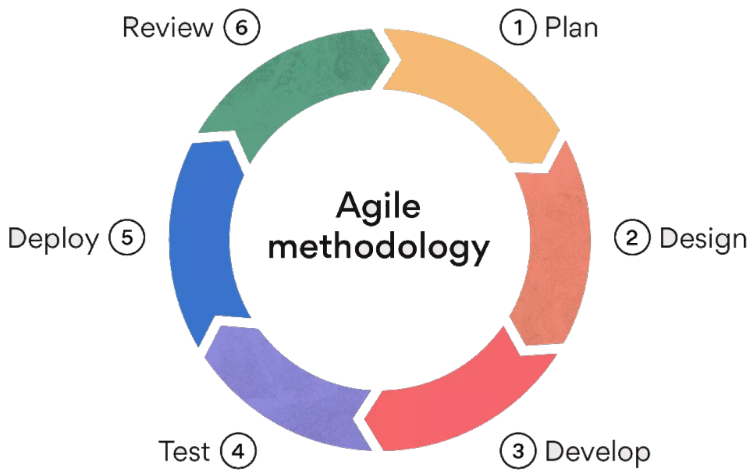


Figure 2: Agile Methodology
[20]

1. **Plan:** Define a list of tasks and establish an estimated timeline.
2. **Design:** We design the overall structure of the product and lay the foundation for implementation.
3. **Develop:** Implement the solution based on the previous design.
4. **Test:** Verify that the solutions function correctly. While final tests are typically conducted at the end of the project, testing and development often go hand in hand and are performed concurrently.
5. **Deploy:** Release the final product into a production environment.
6. **Review:** Evaluate the product's performance in the production environment to determine necessary changes, leading the team back to the planning stage.

In our case, the development will follow the aforementioned steps. To track progress, we will hold a weekly meeting with my advisors, Prof. Keller and Prof. Lehman. This meeting will provide an opportunity to review the project's progress, consider solutions to any challenges that arise, and explore possible directions to take. The research and meetings will take place at the UCB Computer Systems Laboratory.

2.4.2 Tools

The development of this project will require both hardware and software tools.
Hardware:

- **HP Victus Laptop with an AMD Ryzen™ 7 7840HS CPU:** The project will be developed on an HP Victus laptop equipped with an AMD Ryzen™ 7 7840HS CPU. This laptop will be used for both software development and NPU benchmarking.

- **AMD NPU:** The embedded NPU within the AMD Ryzen™ 7 7840HS CPU will serve as the primary hardware target for this project. This NPU is model *npu1* of AMD's AIE Version 2[21].

Software:

- **Git:** Open-source Version Control System tool, used to keep track of changes and manage code versions.
- **G++:** G++ is a C++ compiler, part of the GNU Compiler Collection (GCC). This software converts C++ code into an executable binary.
- **GProf:** GProf is a GNU profiler for C/C++ applications. This program allows for measuring the performance of programs, as well as how much each function contributes to the total runtime.
- **MLIR AIE:** AMD's NPU development toolkit, which provides the necessary development framework to build software for AMD's NPUs.
- **Xilinx RunTime (XRT):** A generic driver base for Xilinx products, serving as a common set of libraries for running code on Xilinx's FPGAs and NPUs.
- **XDNA Driver:** A specific driver required to run software on AMD Ryzen-AI NPUs, installed on top of XRT.

2.4.3 Verification

The development of the project will be verified through two main methods. First, the weekly meeting will serve as a checkpoint to evaluate the progress and ensure that milestones are met. Second, the correctness of the model's implementation will be validated through a dedicated test. This test will verify whether the NPU produces the expected output, as well as benchmark its performance.

Both correctness testing and benchmarking will be conducted iteratively throughout development. This process will continue until either the target performance is achieved or it is determined that the NPU cannot deliver the desired results.

3 Time Planning

The project will run for five months, from February 1 to June 27, 2025, with research conducted from February 1 to May 31, spanning 17 research weeks. The commitment is full-time (8 hours per day), though about 4 hours per week will be allocated to unrelated meetings and tasks. Additionally, some days will be unavailable due to holidays or other commitments. Accounting for these factors, the total estimated project time is approximately 570 hours.

As the thesis defense date has not yet been determined, June 27 has been set as a tentative date and will be updated once finalized. Since this project is conducted in a research laboratory, there are no clients, meaning there are no additional planning constraints or strict deadlines.

3.1 Resources

As with any project, certain resources are essential for its successful completion. The following resources have been identified as necessary to bring it to fruition:

- **Human [R1]:** The developer will be primarily responsible for completing the project. Additionally, the UCB Computer Systems Laboratory and Computer Architecture Laboratory will be involved. The directors of both research laboratories will serve as advisors for the project, assisting with guidance and planning, among other tasks.
- **Hardware [R2]:** An HP Victus Laptop has been chosen for its embedded AMD NPU and low power consumption. Additionally, a network connection will be required to conduct research.
- **Software [R3]:** The software required includes the G++ compiler, the MLIR-AIE toolkit for development, the XRT and XDNA drivers for testing and benchmarking, and Git for version control. As a plus, all the aforementioned software is free to use.

3.2 Task Description

In general, the tasks will focus on comparing the model's runtime and power consumption on an NPU with the previous research on FPGA and multicore implementations. Also, the difficulty of NPU development will be evaluated to determine whether advanced expertise in high-performance computing is required.

The following tasks collectively constitute the entire project. They are directly derived from the objectives outlined in Project Objectives Section. Each task has been assigned an estimated number of hours, with a brief explanation for each estimation. Table 2 provides a detailed breakdown of the tasks, associated resources, and estimated hours.

3.2.1 Project Management

Project management encompasses the non-technical tasks that are crucial to the project's success. These tasks constitute the *GEP* course and are primarily conducted at the beginning of the project. These tasks include planning, thesis preparation and documentation, as well as establishing clear objectives and tasks. Completing these tasks requires a computer for writing documentation and conducting research (resource R2) and a researcher to oversee and execute them (resource R1).

GP1 - Context and Scope
This is the foundation of the entire thesis. At this point, the premise for the project is set. First, the context is defined, with explanations of key technical concepts. The scope is then defined, including a clear statement of the objectives that will be pursued and the motivation behind the project, as well as an identification of the stakeholders. Since this is a fundamental part of the project and requires thoughtful consideration of the details, it translates to a significant amount of time, around 25 hours.

GP2 - Time Planning
This task is more straightforward, as it focuses on setting the tasks and timelines that will guide the project. In other words, it sets the milestones and time references for tracking progress throughout the development process. This task is preceded by GP1, since it requires first setting the fundamentals. While this task requires attention, it is not as time-intensive as GP1. Therefore, around 10 hours will be dedicated to this task.

GP3 - Budget and Sustainability
This task consists of establishing the required funding and sustainability metrics that justify the project's exploration. In a business environment, this would be crucial in assessing whether to fund a new project, while in a research setting, such as the one this project is based in, it would be used to determine whether the project qualifies for funding through a federal grant. This task is preceded by both GP1 and GP2, as sustainability metrics rely on the project's fundamentals, and the budget is calculated based on the time plan. This task is comparable to time planning in terms of workload. Therefore, around 10 hours will be dedicated to this task.

GP4 - Thesis Document
This will be the final thesis document. This document will include the aforementioned documentation, along with the results of the research and the conclusions. This task is preceded by GP1 through GP3, since all their information will be included inside the thesis document. As the central component of the thesis, this task requires substantial effort. However, a significant portion of this document will already be completed through the previous tasks. This task is estimated to take 70 hours, with 45 of them already accounted for.

GP5 - Thesis Defense
The thesis defense involves presenting the entire project in front of a panel of directors. This will be a brief presentation that provides the context and scope, explains the research findings, and concludes. Most of the work will already be completed with the thesis document. Therefore, it will primarily consist of summarizing the thesis docu-

ment into a presentation and rehearsing for the defense. As such, this task is preceded by GP4. This task is expected to require around 10 hours of work.

GP6 - Weekly Meetings
This task consists of the regular meetings with the thesis advisors to review milestones, address questions, and guide the project's progress. These are scheduled to be 30min every week, during the 17 weeks of research, which adds up to around 10 hours. This task does not have any precedence, but will be concurrent to all other tasks during the research period. In specific, this task would also require a human in the role of an advisor, which corresponds to resource R1.

3.2.2 Preliminary Tasks

The preliminary tasks consist of learning the basic concepts and familiarizing oneself with the development framework. This is the knowledge base that will enable the development of the project. These tasks can be conducted in parallel with project management tasks, as there are no dependencies between them. They will be carried out exclusively by the researcher, corresponding to resource R1.

PT1 - Model Study
Study the model and implementation of *The Economic Geography of Global Warming economic model by Cruz, José-Luis and Rossi-Hansberg, Esteban (2021)*[7]. This task can be significantly complex, especially for a developer lacking macroeconomic knowledge. Therefore, this task is set to be around 40 hours. To conduct the research, a computer will be required, corresponding to resource R2.

PT2 - Environment Setup
The environment setup will consist of installing and validating the MLIR-AIE development framework for AMD's AIE, specifically for deployment on the local AMD Ryzen-AI NPU. This will also include checking for hardware and software compatibility issues and fixing them. Even though this task is pretty simple, potential issues might delay it significantly. Therefore, around 80 hours of work are allocated to this task. This task requires a computer with an embedded NPU for setup (resource R2) and the MLIR-AIE development framework (resource R3).

PT3 - MLIR-AIE Framework Study
Explore NPU development methodologies and available resources, entailing learning how to use the MLIR-AIE development framework and understanding how each C++ and Python component contributes to the final NPU implementation. The MLIR-AIE framework provides an in-depth programming guide that takes around 40 hours to be complete[3]. This task requires a computer with an embedded NPU to follow the programming guide (resource R2) and the software development framework (resource R3). Additionally, this task is preceded by PT2, as the environment must be set up before it can be used.

PT4 - State of the Art Analysis
This task requires first understanding the basic concepts of the economic model and

[3]https://github.com/Xilinx/MLIR-AIE/tree/main/programming_guide

the NPU implementation, which means that PT1 and PT3 precede it. It consists of reviewing the FPGA and CPU implementations provided by *Cheela, DeHon, Fernández-Villaverde, and Peri (2025)* [5] and estimating the expected results on an NPU. This will need to be discussed with the thesis advisors. Given that much of the analysis is already covered in the referenced paper, this task is not expected to require significant effort. Thus, around 10 hours of work are estimated for this task. This task will require a computer to conduct the research, corresponding to resource R2.

3.2.3 Product Development

Once the preliminary tasks have been completed, the actual development of the product can begin, thus being preceded by PT. It needs to first be designed, then implemented, and finally verified for possible bugs. All these tasks require the human in the role of researcher (R1), the computer to design and implement the model (R2), and the software development framework to implement and verify it (R3). However, the design task does not require the software development framework (R3) since it will consist only of theoretical design. The tasks have been set as the following:

PD1 - Design
This is the design of the model implementation on the NPU. It will need taking into account how the model works and the previous implementations. Once the NPU's implementation design is ready, it will be discussed with the advisors to assess the feasibility. Since the model has been implemented in several different workflows previously, the design should be somewhat portable and not a workload burden. Therefore, a total of approximately 20 hours have been allocated for the design.

PD2 - Implementation
This task involves translating the previous design into functional MLIR-AIE code, thus being preceded by PD1. This code will then be compiled to be run on the local AMD Ryzen-AI NPU. As development progresses, Git will be used for version control, which is designated as resource R3. The implementation consists of two parallel subtasks:

- **PD2.1 - Dataflow:** Designing the dataflow — how data moves within, into, and out of the NPU. This task is not significantly complex and an estimated 40 hours will be dedicated.
- **PD2.2 - Algorithm:** Implementing the algorithm itself, which is the most complex part, requiring significant optimization for efficient execution. It is expected to require around 90 hours of work.

Together, these subtasks amount to a total of 130 hours, reflecting the substantial effort required for the project's core development. Although they may be executed in sequential order, there is no precedence between them and they may be executed concurrently.

PD3 - Verification
This task focuses on verifying the correctness of the implementation by comparing the NPU output with the golden model from previous implementations. Thus being preceded by a correct design (PD1). If the outputs match for a range of inputs, the

implementation is considered correct. Otherwise, debugging will continue until discrepancies are resolved. This task is comprised of two subtasks:

- **PD3.1 - General Verification:** Short task that consists of general debugging and verification. The estimate is around 20 hours.
- **PD3.2 - Corner-Case Verification:** Long task that consists of corner-case bugs that might become an obstacle and can take a few days to fix. The estimate is around 60 hours.

In total, verification is expected to take 80 hours and will be conducted concurrently with implementation, as debugging will directly drive development.

3.2.4 Product Assessment

The product assessment is the final step in the project and will depend on a correct implementation of the model, thus being preceded by PD. There will be two main aspects to be assessed: performance and feasibility. The assessment will be carried out by a human in the role of a researcher, requiring resource R1. These will be studied through the following two tasks:

PA1 - Benchmarking & Performance Metrics
The benchmarking task consists of measuring the model's performance metrics in terms of execution timing and power consumption. This task depends on a correct implementation of the model and is conducted once the model has been implemented and verified. This task requires a computer with an embedded NPU (resource R2) and the development framework (resource R3) to execute performance evaluations. This task is not expected to be complex or subject to unexpected delays, with an estimated 40 hours of dedication.

PA2 - Evaluation
This final task depends on the model implementation and benchmarking, thus being preceded by PA1. Once the results are obtained, an evaluation will assess whether the metrics are positive and whether the development process is accessible to scientific researchers, particularly in economics. The evaluation will include a holistic analysis of NPUs' potential to promote the development of generic scientific models. While this task requires significant effort, it is not expected to be time-intensive, with an estimated duration of 40 hours. The evaluation will require a computer for documentation and analysis, corresponding to resource R2.

3.2.5 Task Breakdown

The following Table 2, provides a detailed view of the tasks, their estimated duration, their dependencies, and the resources used.

Code	Task	Dedication	Dependencies	Resources
GP	**Project Management**	**90h**		
GP1	Context & Scope	25h		R1, R2
GP2	Time Planning	10h	GP1	R1, R2
GP3	Budget & Sustainability	10h	GP1, GP2	R1, R2
GP4	Thesis Document	25h	GP1, GP2, GP3	R1, R2
GP5	Thesis Defense	10h	GP4	R1, R2
GP6	Weekly Meetings	10h		R1, R2
PT	**Preliminary Tasks**	**170h**		
PT1	Model Study	40h		R1, R2
PT2	Environment Setup	80h		R1, R2, R3
PT3	MLIR-AIE Study	40h	PT2	R1, R2, R3
PT4	State of the Art Analysis	10h	PT1, PT3	R1, R2
PD	**Product Development**	**230h**	PT	
PD1	Design	20h		R1, R2
PD2	Implementation	130h	PD1	R1, R2, R3
PD2.1	Dataflow	40h		
PD2.2	Algorithm	90h		
PD3	Verification	80h	PD1	R1, R2, R3
PD3.1	General Verification	20h		
PD3.2	Corner-Case Verification	60h		
PA	**Product Assessment**	**80h**	PD	
PA1	Benchmarking & Performance	40h		R1, R2, R3
PA2	Evaluation	40h	PA1	R1, R2
Total		**570h**		

Table 2: Task Breakdown
Author's own work

3.3 Risk Management

As outlined in Possible Risks & Roadblocks Section, we have described the course of action for the most likely challenges. This includes alternative tasks, how it would affect the duration of the project, and the additional resources required:

- **Compatibility Issues:**
  - **Hardware Incompatibility:** The HP Victus Laptop used for this project may not yet be compatible with the MLIR-AIE development framework, which would affect PT2. However, this would cause no deviation from the original plan (0h), since we have a backup computer available that has already been tested to run MLIR-AIE (resource R2).
  - **Software Incompatibility:** Given incompatible software, the solution would be switching to a compatible version. While some software modifications, such as changing the operating system, may be complex and cause minor delays, the adjustment should take no more than 16h of development on PT2. This would require the specific resources related to the software incompatibility, be it another operating system or specific libraries (resource R3).
- **Implementation Errors:**
  - **Bugs:** Implementation bugs, especially corner-case, can be challenging to resolve. To address this, we have allocated a conservative amount of time to PD3.2 and PD2, since the bugs have to be verified and the fix implemented. We expect this task to be a deviation of at most 80h, which would not require additional resources. However, if a bug takes too long to diagnose, we would continue with the implementation as if the model were functioning properly. In such cases, we would note in the thesis results that the findings are contingent on resolving the bug and verifying whether the results remain consistent. This approach ensures that the thesis will still be delivered on time.
  - **Timing Violations:** Similar to corner-case bugs, timing violations can be time-consuming, taking as much as 80h to resolve but no additional resources. To mitigate this, we have allocated a significant amount of time to PD2, allowing for necessary optimizations. If the desired timing results are not achieved by the end of the research, we will assess whether this is due to deadline constraints or fundamental limitations.

Planning errors can cause a project's timeline to capsize. To mitigate this risk, we have set conservative estimates for critical tasks, buffering against unforeseen delays. In the worst-case scenario, there might be an increment of up to 176h to the project's duration. If the development is short on time, we may opt to re-plan or reduce the implementation to a simpler model. Thus, we believe that the project will be completed on time, absent major unrelated events.

4 Preliminary Study

Before beginning to implement the model and obtain results, I would like to conduct a preliminary study. This preliminary study will consist of an analysis of the actual model and an analysis of the characteristics of the NPU we are using.

4.1 The Economic Geography of Global Warming Model

The model presented in *The Economic Geography of Global Warming economic model by Cruz, José-Luis and Rossi-Hansberg, Esteban (2021)*[7] is a SIAM designed to study the long-term economic impacts of climate change. It incorporates geographic detail with a global grid, modeling how temperature changes affect local productivity, amenities, and natality rates. The model includes responses to climate change through migration, trade, innovation, and fertility. The model predicts welfare losses of up to 20% in some tropical regions and gains in colder areas like Siberia and Canada — thus increasing global spatial inequality.

The model also considers emissions from fossil fuel use and their impact on global and local temperatures. Policy analysis using the model shows that clean energy subsidies alone are insufficient. While carbon taxes can delay emissions, they just “flatten the temperature curve”, mitigating short to medium term warming, but having little impact over the long-run.

4.2 Banach fixed-point theorem

The critical part of the model finds a fixed point in a matrix using the Banach fixed point theorem. The theorem states the following:

Let (X, d) be a non-empty complete metric space with a contraction mapping $T : X \longrightarrow X$. Then T admits a unique fixed-point x^* in X, such that $T(x*) = x*$. The fixed-point can be found with the following procedure:

Start with an arbitrary guess $x_0 \in X$. Define a sequence $x_n = T(x_{n-1})$ for $n \geq 1$. Then $\lim_{n\to\infty} T(x_n) = x^*$[22].

4.3 Model Analysis

The Economic Geography of Global Warming economic model by Cruz, José-Luis and Rossi-Hansberg, Esteban (2021)[7] has already been implemented in single-core scalar CPU execution in the C++ language by Professor Alessandro Peri, from the Department of Economics at the University of Colorado Boulder. With this initial CPU implementation, an analysis on the computational characteristics of the model can be conducted.

4.3.1 Critical Function

The critical function of the model is *loop_uhat*. This function receives a matrix M as input and tries to find a fixed-point x^*. That is, it tries to find a vector x^* for which $Mx^* = x^*$. In order to do so, it follows the aforementioned Banach fixed-point theorem.

Algorithm Description: The function starts with an arbitrary guess (vector). A coefficient and exponential — *R*, *R_exp* – is applied to each element of the guess, which produces the input vector. The input vector is multiplied by the matrix — trmult_reduced — and produces the output vector. A different coefficient and exponential — *L*, *L_exp* — is now applied to the output vector of the matrix-vector multiplication, which produces the result. The distance between the result and the original guess is computed. If the distance is less than the tolerance, the function assumes it has found a sufficiently good result and exits. Otherwise, this result is taken as the guess for the next iteration. The following pseudo-code describes the algorithm:

```
guess_new = [1, ..., 1]
dist = tol
while dist >= tol:
    input_vector = R * guess_new^R_exp
    output_vector = Matrix * input_vector
    result = L * output_vector^L_exp
    dist = dist_funct(result, guess_new)
    guess_new = result
```

The original C function can be found in Appendix E.1.1.

This function follows the iterative procedure described in equations *I.37* and *I.38* of the supplemental material of the *The Economic Geography of Global Warming by Cruz, José-Luis and Rossi-Hansberg, Esteban (2021)*[7].

Parameter Description: The function takes the following 6 arguments:

```
const real tolerance, const real *trmult_reduced, const real *L,
↪  const real L_exp, real *uhat_i, const real *R, const real R_exp
```

- **tolerance:**
  - **Type:** Double-precision floating point.
  - **Size:** 1 element.
  - **Description:** This is the distance, between the guess and the result, at which we assume we have found the fixed point.
- **trmult_reduced:**
  - **Type:** Double-precision floating point.
  - **Size:** 17048x17048 elements.

  - **Description:** This is the matrix by which we are going to multiply the input vector.

- **L:**
  - **Type:** Double-precision floating point.
  - **Size:** 17048 elements.
  - **Description:** Each coefficient applied to each element of the output vector.
- **L_exp:**
  - **Type:** Double-precision floating point.
  - **Size:** 1 element.
  - **Description:** Exponential applied to all elements of the output vector.
- **uhat_i:**
  - **Type:** Double-precision floating point.
  - **Size:** 17048 elements.
  - **Description:** Fixed point that the algorithm has found.
- **R:**
  - **Type:** Double-precision floating point.
  - **Size:** 17048 elements.
  - **Description:** Each coefficient applied to each element of the input vector.
- **R_exp:**
  - **Type:** Double-precision floating point.
  - **Size:** 1 element.
  - **Description:** Exponential applied to all elements of the input vector.

4.3.2 CallGraph Analysis

If we want to put the *loop_uhat* function into perspective, we can use the CallGraph found in Figure 3. As it can be seen, the *loop_uhat* function is called from four different functions: *model_initial_period*, *migration_costs*, *forward_climate*, *backward_climate*. Out of *loop_uhat*'s 6 parameters, the tolerance is fixed to 0.01 across different calls. In addition, the matrix — *trmult_reduced* — is loaded at the beginning of the program and not modified during runtime. The other parameters can change on each call.

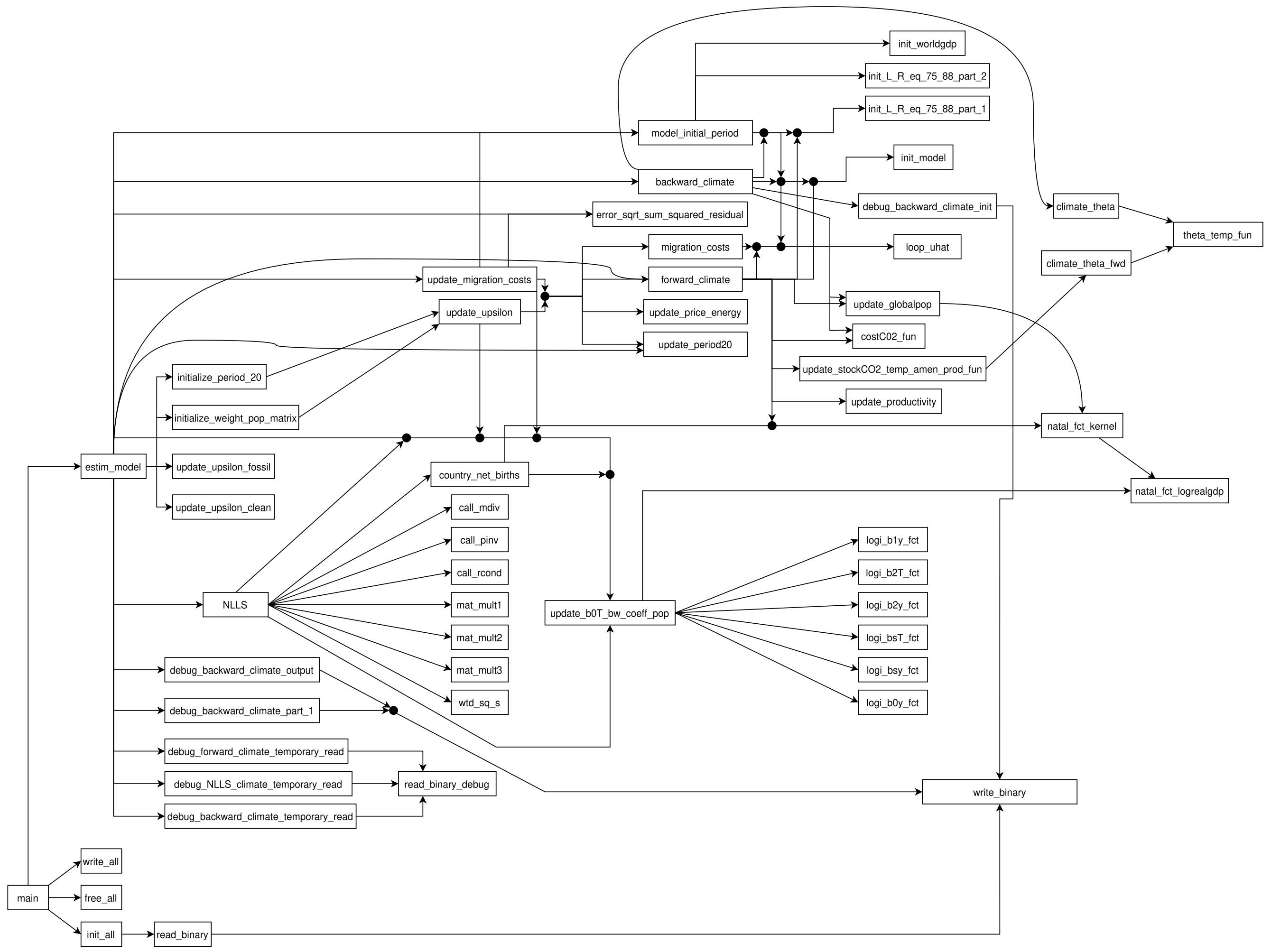


Figure 3: Model CallGraph
Author's own work

4.3.3 Value Analysis

The distribution of matrix values plays an important role in determining feasible optimizations. These are summarized in Table 3:

Filter	Amount	% of total	Total Amount
$\geq$1	7	2.4e8	
= 1	0	0	17408^2
$\leq$1	290634297	99.99...%	
= 0	0	3.9e-03	

Table 3: Economic Model Matrix's Values
Author's own work

As shown, the vast majority of matrix elements are less than 1. To be specific, the minimum value is 2.3508e-23, while the maximum value is 2.9163. This means that most matrix values will attenuate the input during multiplication, effectively reducing its magnitude. It is also important to note that, while the matrix has very small values, the guess' magnitudes vary across iterations. In some cases, its elements can reach values exceeding 10^{45}.

The matrix is also characteristic for being positive-definite, that is, all of its values are positive. Therefore, any positive-definite input vector will yield a positive-definite output vector. Given that coefficients *L* and *R* are also positive, both the initial guess and the resulting vectors also remain positive-definite to maintain consistency with the algorithm's structure.

4.3.4 Performance Analysis

After profiling the application, the first thing we notice is that *loop_uhat* takes 97.93% of the runtime. The function is structured as a while loop containing 4 nested for loops, as outlined in Table 4.

Loop	# Iterations	Iteration	Parent %	Total %
Error Minimizing While-Loop	42	818.7ms	99.94%	97.87%
Integral Computation	17048	10.15ns	0.02%	0.02%
Matrix-Vector Multiplication	17048	48us	99.95%	97.82%
Row Multiplication	17048	2.76ns	98%	95.86%
Guess Update	17048	13.37ns	0.03%	0.03%

Table 4: For-Loops in *loop_uhat* function
Author's own work

As it can be observed, the matrix-vector multiplication takes most of the application's runtime, around 97.82%. This implies that, if we were able to make this matrix-vector multiplication infinitely fast, the application would run 45.87 times faster than the original. For instance, if a specific execution of the model takes 7h, the newer version would take less than 10 minutes.

4.4 NPU Analysis

In order to implement the model on the NPU, we first need to understand the different floating points available in computing. Then, we need to understand the architecture of the NPU, as well as the supported operations.

4.4.1 Floating Point Types

Before analyzing the NPU, it is important the bear in mind the details of the different floating-point types used in mathematical models. Floating-point formats represent real numbers in computing with varying precision and range, depending on their bit-width. These formats follow a standard structure but differ in size and dynamic range. The relevant types used in this work are summarized in Table 5:

Name	Min/Max Values	Precision Error %*
Double-Precision Floating Point (FP64)	±2.23e−308, ±1.80e308	1.11e-16
Single-Precision Floating Point (FP32)	±1.18e−38, ±3.4e38	5.96e-08
Half-Precision Floating Point (FP16)	±5.96e-08, ±65504.0	4.88e-04
Brain Floating Point (BF16)	±9.2e-41, ±3.39e38	3.9e-03

Table 5: Floating Point Types
Author's own work

****Note:*** The precision error % is the maximum possible rounding error when rounding to the nearest value[23].

4.4.2 Supported Types

As the NPU is designed primarily for machine learning workloads, it is optimized for 16 and 8-bit data types. While FP32 operations are supported, they are not accelerated; such operations are executed as unoptimized, scalar code and are emulated by the NPU[24]. In contrast, the only natively supported floating-point format is BF16. Although BF16 introduces a slightly higher numerical error compared to FP16, the NPU supports native accumulation in FP32, which helps mitigate this loss in precision. Furthermore, because BF16 shares a similar dynamic range with FP32 — unlike FP16 — it enables efficient storage of large-magnitude values without significant loss of representation.

4.4.3 Value Scaling

Even though BF16 provides a wide range of representation, it still falls short of the FP64 format used in the model. As it can be seen in the Value Analysis Section, the guess can hold several elements that exceed the representable range of BF16. To address this, we must find a way to represent such values in BF16 without incurring significant precision loss.

We can leverage the linearity of matrix-vector multiplication, which satisfies the properties of additivity and homogeneity. This allows us to rescale the input vector, perform the multiplication in the lower-precision format, and then rescale the result. Specifically, for a matrix *M*, input vector *x*, arbitrary vector *z*, and arbitrary scalar *s*, the following identity holds: $Mx = \frac{1}{s} * (M(xs + z) - z)$.

Therefore, each input vector can be scaled down with a linear transformation. Given an input vector *x*, and the minimum and maximum representable values of BF16 — denoted as $BF16_{min}$ and $BF16_{max}$ — we can define the following map:
$f(x) : \mathbb{R} \to \mathbb{R} \mid f(min(x)) = BF16_{min}, f(max(x)) = BF16_{max}$.

Following previous notation, the transformation parameters can be computed as:

$$s = \frac{BF16_{max} - BF16_{min}}{max(x) - min(x)}, z = (-s * min) + BF16_{min}$$

Here, the scaling factor *s* compresses the value range into the BF16 domain, while the offset *z* aligns the minimum value with $BF16_{min}$.

However, there is still an issue. Since the values are mapped as far as the extremes of $BF16$, they may still overflow and become $\pm\infty$. As expressed in the Value Analysis Section, all values are positive-definite. Therefore, instead of mapping to $[BF16_{min}, BF16_{max}]$, we will map to $[0, BF16_{max}]$. This transformation enables computation within the BF16 range while preserving the result's accuracy and avoiding undefined values such as $\pm\infty$.

4.4.4 Supported Operations

As previously mentioned, the NPU is designed primarily for machine learning workloads, with native hardware acceleration focused on matrix–matrix multiplication. However, it also supports vector–vector dot-product operations, which can be composed to implement matrix–vector multiplication efficiently. Specifically, Table 6 outlines the floating-point operations natively accelerated by the NPU used in this project.

Name	**Operation Type**	**Matrix A**	**Matrix B**
aie::accumulate<8>	Vector Dot Product	8xN*	Nx1*
aie::mmul<4, 8, 4, ...>	Matrix Multiplication	4x8	8x4
aie::mmul<4, 16, 4, ...>	Sparse Matrix Multiplication	4x16	16x4

Table 6: NPU1 Supported Operations
Author's own work[24][25]

***Note:** *N* is an arbitrary number.

4.4.5 Architecture

To optimize the model's implementation, it is essential to understand the architecture of the target NPU. In our case, we use *NPU1* based on AIE Version 2. The architecture is illustrated in Figure 4. As shown, each column — except the first — contains four compute cores, one *MemTile*, and one *ShimDMA* tile. The first column lacks a *ShimDMA*, which can limit bandwidth and potentially degrade performance if memory access becomes a bottleneck.

External L3 memory refers to the host's DRAM, where buffers allocated by the host application reside. *ShimDMA* tiles fetch data directly from DRAM and route it either to compute cores or to *MemTiles*. Each *MemTile* offers 2.5MB of storage, while each compute core has 64KB of local high-speed memory[26][27].

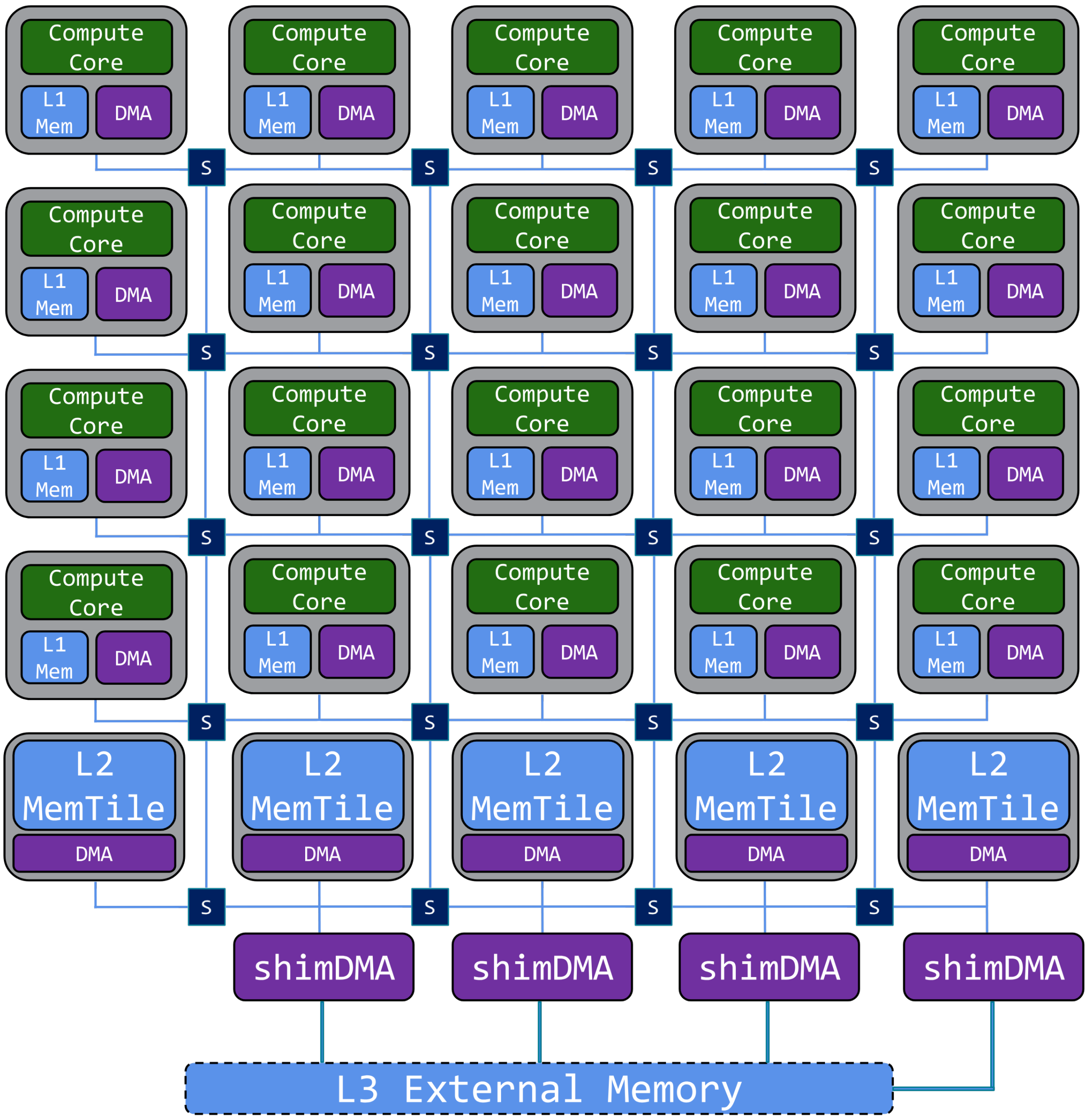


Figure 4: NPU1 Architecture
Author's modification of [14].

4.4.6 Data Transfer

Another important aspect to consider is data transfer within the NPU. *ShimDMA* tiles have direct access to DRAM and can arbitrarily read from any region for which a buffer has been allocated. However, they are constrained in the number of total memory accesses they can perform, particularly as input size increases.

In contrast, MemTiles and Compute Cores operate on data streams. They receive and send data linearly, as a continuous flow. Although some reshaping can be performed on the incoming data, it is limited by the size of individual stream elements. Streams are expected to remain small, typically consisting of only 1 or 2 buffers.

5 Matrix-Vector Acceleration

As established in the preliminary study, the primary bottleneck of the model lies in the matrix–vector multiplication within the *loop_uhat* function. Consequently, accelerating this function alone is sufficient to achieve a substantial overall speed-up. Given that it accounts for nearly 100% of the runtime, improvements in its execution time translate almost directly to equivalent improvements in the total model runtime.

To maximize performance, the accelerated implementation must be highly efficient. To avoid data dependencies and potential slowdowns, the design must ensure that each computational unit processes independent regions of data, enabling parallel execution without synchronization overhead.

To enable a fair comparison with CPU performance, we also implement an AVX + Multithreaded version of the algorithm (AVX-M). This allows us to evaluate the NPU not only against a naive scalar CPU baseline, but also against an optimized CPU implementation that leverages both thread-level parallelism and SIMD vector extensions.

5.1 Data Reshaping Algorithm

To ensure correct results, the data must be reshaped before processing. The reshaping algorithm implemented on the host mirrors the logic used by the NPU.

Given a set of dimension pairs $P_1 ... P_k$, where $k \in \mathbb{N} \mid k \geq 1$, each pair $P_i = (x_i, y_i) \mid x_i, y_i \in \mathbb{R}$, $i \in [1, k] \cap \mathbb{N}$, and matrix M, the reshaped matrix M^* is constructed as follows:

```
pos = 0
for $i_k$ in range($x_k$):
  ...
    for $i_1$ in range($x_1$):
        M*[pos++] = M[$i_1$ * $y_1$ + ... + $i_k$ * $y_k$]
```

In this context, each y represents the stride of a dimension, while x corresponds to the number of iterations along that dimension.

5.2 NPU Implementation

Given a matrix M, input vector v_i, and output vector $v_o = Mv_i$, each element of v_o can be computed independently. The only shared input is v_i, which is multiplied by each row of M. Since each row-wise computation is independent, the NPU implementation assigns each compute core a distinct set of consecutive rows. This ensures that no core depends on intermediate results from others, allowing for fully parallel execution.

However, each compute core is limited to 64KB of local memory, and a single row of M already exceeds half of this capacity. Therefore, the matrix–vector multiplication must be split into multiple steps. Initially, each compute column is assigned a subset of consecutive rows, its designated portion of M. For this subset, the compute column iterates over M's columns in blocks, processing partial dot products until all columns are covered.

Figure 5 illustrates this data distribution strategy. Each compute column is responsible for a unique set of rows (represented by different colors), and processes them iteratively across column blocks (represented by different shades).

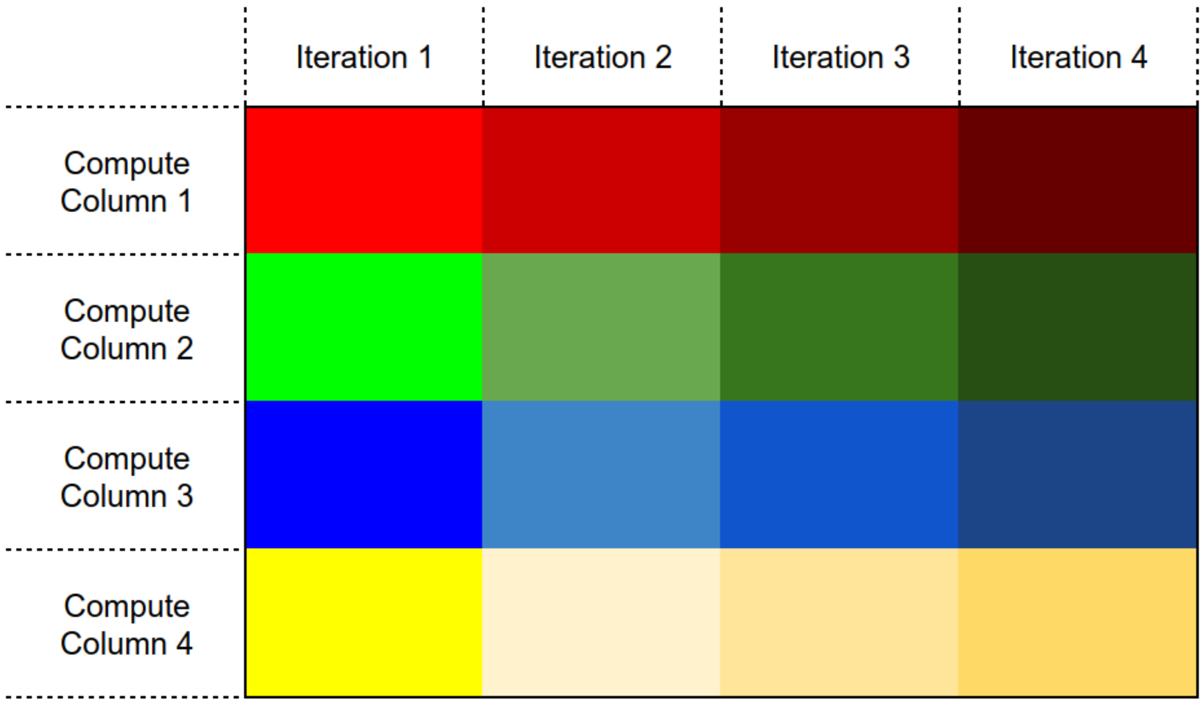


Figure 5: NPU MatVec General Design
Author's own work.

As previously discussed, each compute column processes a portion of M, which is further divided into subsets of rows and columns. Each compute column consists of several compute cores. Within a single compute core, each group of rows is referred to as a row iteration, and each group of columns as a column iteration. For every row iteration, the core performs all corresponding column iterations before moving on to the next set of rows.

To maximize performance and leverage hardware acceleration, each row iteration processes multiple rows simultaneously, and each column iteration processes multiple columns at once.

Figure 6 illustrates this data distribution strategy for a single compute tile. The assigned portion of M is divided into row iterations, with each row iteration processed across column blocks (represented by different shades).

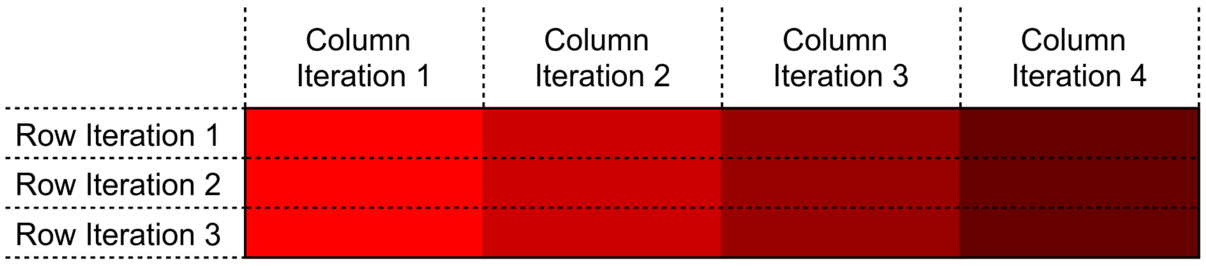


Figure 6: NPU MatVec Subtile Reshape
Author's own work.

5.2.1 Hardware Acceleration:

To accelerate the matrix–vector multiplication, we cannot leverage the NPU's native matrix–matrix multiplication. However, as discussed in the NPU Supported Operations Section, the hardware does support native vector dot-product acceleration, which we exploit in this implementation. This operation can process 8 rows of the matrix in parallel with an arbitrary number of columns. Based on experimentation, we perform computations using square tiles of size 8×8.

Additionally, each compute core has a strict 64KB memory limit, which constrains the size of the data it can process at once. Through empirical testing, subtiles of size $(m, k) = (32, 32)$ were found to provide optimal performance. In this configuration, m denotes the number of rows processed per row iteration, while k denotes the number of columns processed per column iteration.

5.2.2 Data Reshaping:

Since the matrix is stored in row-major order but each compute core requires access to specific tiled regions, a data reshaping step is necessary. This reshaping can be performed either on the host or on the NPU. While the NPU is optimized for such operations, testing during this research has shown that it lacks the capacity to handle reshaping for large matrices. Consequently, reshaping must be performed on the host, despite its relative inefficiency for this task.

However, as noted in the Callgraph Analysis Section, the matrix remains constant throughout execution. This means the most computationally expensive reshape — due to the matrix's size — only needs to occur once, at startup. By reshaping the matrix on the host at initialization, we can avoid repeated reshaping costs during runtime.

Input Matrix:
Figure 7 illustrates how subtiles are sent to the NPU across multiple compute tiles, in a configuration with two compute columns, two compute rows per compute column, and one row iteration assigned to each compute core. Each number in the figure indicates the order in which a subtile is transferred to the NPU.

As shown, one subtile is sent to each compute core within a compute column before proceeding to the next column iteration. Once all subtiles have been sent to the first compute column, the process is repeated for the next compute column. This sequential ordering simplifies data access for the *ShimDMA*, allowing it to simplify its data transfer to a single linear transfer, instead of multiple different transfers.

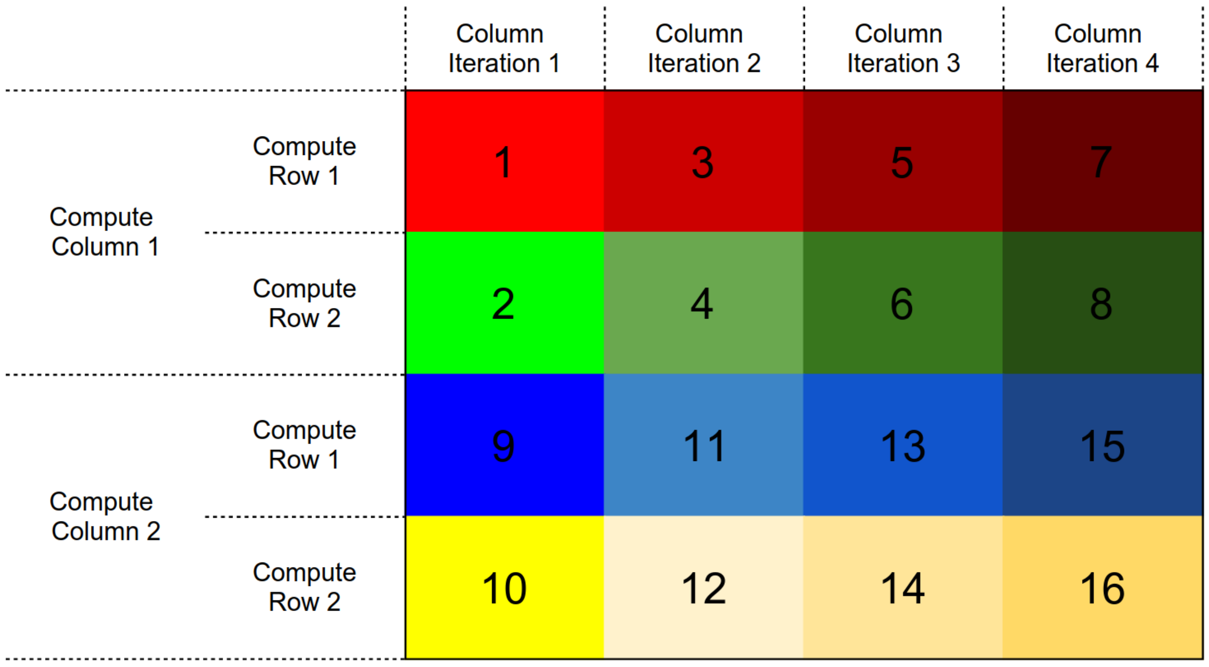


Figure 7: NPU MatVec Matrix Reshape
Author's own work.

Figure 8 provides a detailed view of how subtiles are sent to the NPU within a specific compute column. In this configuration, each NPU column uses two compute cores, and each core processes two row iterations. Each number in the figure indicates the order in which a subtile is transferred.

As shown, all subtiles for the first row iteration are sent before moving to the next. Within each row iteration, all subtiles for a column iteration are sent before proceeding to the next. Once all column iterations have been completed, the process repeats for the next row iteration.

Note: Each subtile is sent in column-major order to match the input format expected by the NPU's hardware acceleration functions.

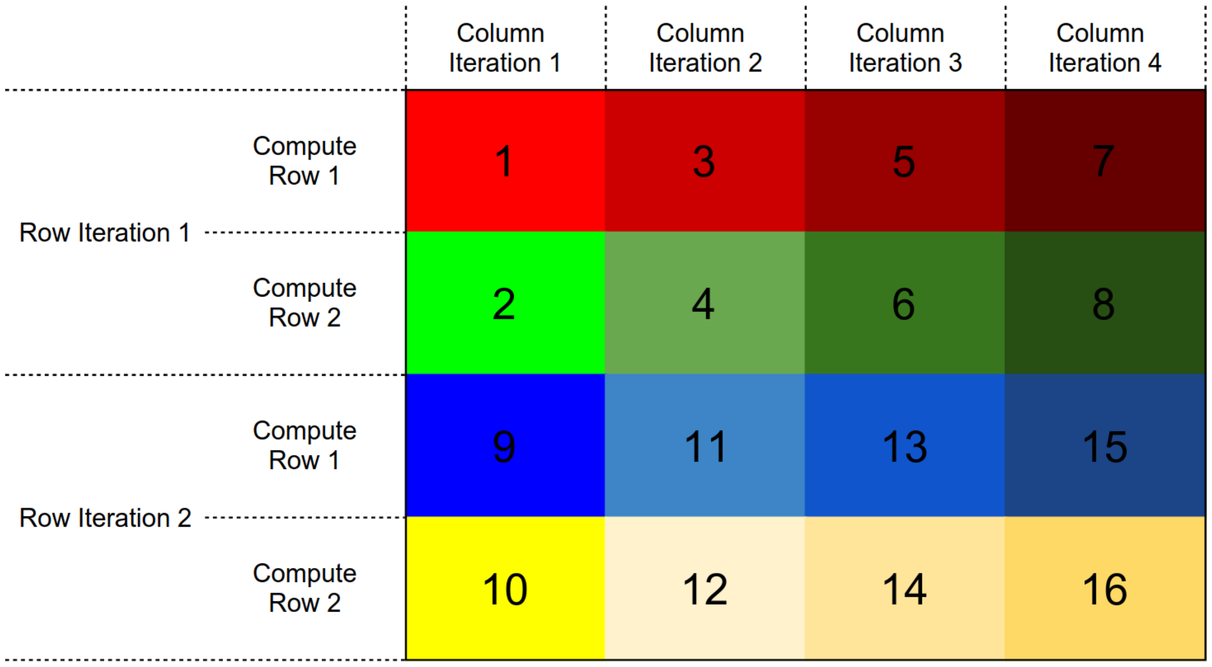


Figure 8: NPU MatVec Matrix Column Reshape
Author’s own work.

Finally, the previously described data transfer pattern, when expressed in terms of reshaped dimensions, corresponds to the following pairs:

$$
\begin{aligned}
P_6 &= (n\_aie\_columns,\ \tfrac{rows\_matrix \times columns\_matrix}{n\_aie\_columns}) \\
P_5 &= (\tfrac{rows\_matrix}{m \times n_{cores}},\ m \times n\_aie\_rows \times columns\_matrix) \\
P_4 &= (\tfrac{columns\_matrix}{k},\ k) \\
P_3 &= (n\_aie\_rows,\ m \times columns\_matrix) \\
P_2 &= (k,\ 1) \\
P_1 &= (m,\ columns\_matrix)
\end{aligned}
$$

Input Vector:

The input vector must be sent once for each row iteration assigned to a compute core. Beyond this, no reshaping is necessary, as the input vector will be accessed linearly.

$$P_1 = (\tfrac{rows\_matrix}{m \times n_{cores}},\ 0)$$

Output Vector:

The output vector does not require reshaping, as each compute column produces its assigned portion in order. Because the matrix row distribution is contiguous across compute columns and each core writes its results sequentially, the final output is already correctly laid out in memory.

5.2.3 Data Transfer:

It is also important to consider how the data will be transferred. The simplified data layout resulting from reshaping will allow the data to be consumed and produced linearly. The following holds true:

Input Matrix:

- **ShimDMA:** For its aie column, it will transfer $\frac{rows \times columns}{n\_aie\_columns}$ contiguous elements from the host to the corresponding compute column.
- **MemTile:** For every $m \times k \times n\_aie\_rows$ elements received from the *ShimDMA*, it will divide them into segments of $m \times k$ elements and send one segment to each compute row in its compute column.

Input Vector:

- **ShimDMA:** For its aie column, it will transfer the entirety of the input vector, including its repetitions, to its respective MemTile. This will be equivalent to $\frac{rows}{m \times n\_cores} \times columns$ elements.
- **MemTile:** For every k elements received from the *ShimDMA*, it will broadcast them into each compute core in its aie column.

Output Vector:

- **MemTile:** For every m elements received from each compute core in its aie column, it will join them to produce $m \times n\_aie\_rows$ elements and send them to its corresponding *ShimDMA* tile.
- **ShimDMA:** For its aie column, it will transfer the elements received from the MemTile into a contiguous $\frac{rows \times columns}{n\_aie\_columns}$ elements to the host.

5.2.4 Size Rounding:

According to the previously described data distribution, the matrix must satisfy the conditions: $rows\ mod(m \times n_{cores}) == 0$ & $columns\ mod\ k == 0$. The original matrix has dimensions 17048×17048, which do not meet these requirements. Therefore, we expand both the matrix and the input vector by padding with zeros to match $(m, k) = (32, 32)$ and $n_{cores} = 16$. In this case, the matrix and vector are expanded from 17048×17048 to 17408×17056.

5.3 AVX + Multithreading Implementation

Following the same data-independence principle as the NPU implementation, the AVX + Multithreading (AVX-M) approach leverages the fact that each row-wise computation is independent, thereby eliminating inter-thread dependencies. Each thread is assigned a distinct set of consecutive rows to process.

We structure the computation similarly to the NPU design: the algorithm first computes an entire set of rows by iterating across all columns, then proceeds to the next set of rows. Unlike the NPU, however, the CPU already has access to DRAM. As a result, the implementation simply reshapes the data in memory without requiring explicit transfers.

Figure 9 illustrates the data distribution strategy for the AVX-M implementation. As shown, the column iterations are narrower to reflect the limited width of each AVX dot-product instruction, which can only process a small number of columns at a time. Each thread is assigned a unique set of rows (indicated by different colors) and processes them iteratively across column blocks (represented by different shades).

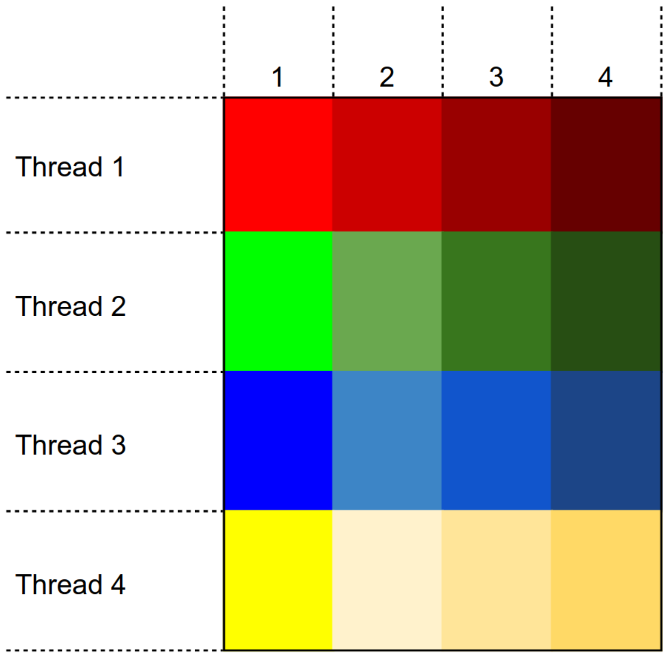


Figure 9: AVX MatVec General Design
Author's own work.

Within each thread, the computation process mirrors that of a compute core in the NPU, as illustrated previously in Figure 6. Each thread processes one row iteration — a group of consecutive rows — by iterating across all columns before moving on to the next row iteration. To maximize performance and utilize AVX hardware acceleration effectively, each column iteration processes two columns simultaneously.

5.3.1 Hardware Acceleration:

Due to the nature of AVX, it does not provide dedicated matrix operations. However, it does support dot-product instructions, which can be leveraged to implement matrix–vector multiplication. In this context, we use the dot-product to compute the inner product between a set of rows of the matrix and the input vector. The following function demonstrates this approach, where the result C is incremented by the product of A and B:

```
__m512 _mm512_dpbf16_ps(__m512 __C, __m512bh __A, __m512bh __B)
```

The parameters are the following:

- **A:**
  - **Type:** BF16.
  - **Size:** 16x2 elements.
  - **Description:** The input matrix.
- **B**
  - **Type:** BF16.
  - **Size:** 2 elements.
  - **Description:** The input vector.
- **C**
  - **Type:** FP32.
  - **Size:** 16 elements.
  - **Description:** The output vector against which the result is accumulated.

$C+ = A \times B$
As with the NPU implementation, accumulation is performed in FP32 rather than BF16 to improve numerical accuracy and minimize precision loss.

5.3.2 Data Reshaping:

As with the NPU, each thread's access to specific tiled regions of the matrix requires a data reshaping step. However, since the matrix remains constant throughout execution, this reshaping can be performed once at startup, avoiding repeated overhead. Moreover, because AVX operations allow the input and output vectors to be accessed linearly without reshaping, the overall reshaping cost is effectively non-existent.

Input Matrix:
Figure 10 illustrates how subtiles are reshaped across multiple threads, in a configuration where each of the two threads is assigned two row iterations.

As shown, each row iteration is sent in full — covering all associated column iterations — before proceeding to the next row iteration. Once all row iterations for a given thread have been sent, the process continues with the next thread's assigned tiles. Notably, if each subtile is viewed as an element of a higher-level matrix, the reshaping process effectively follows a standard row-major order.

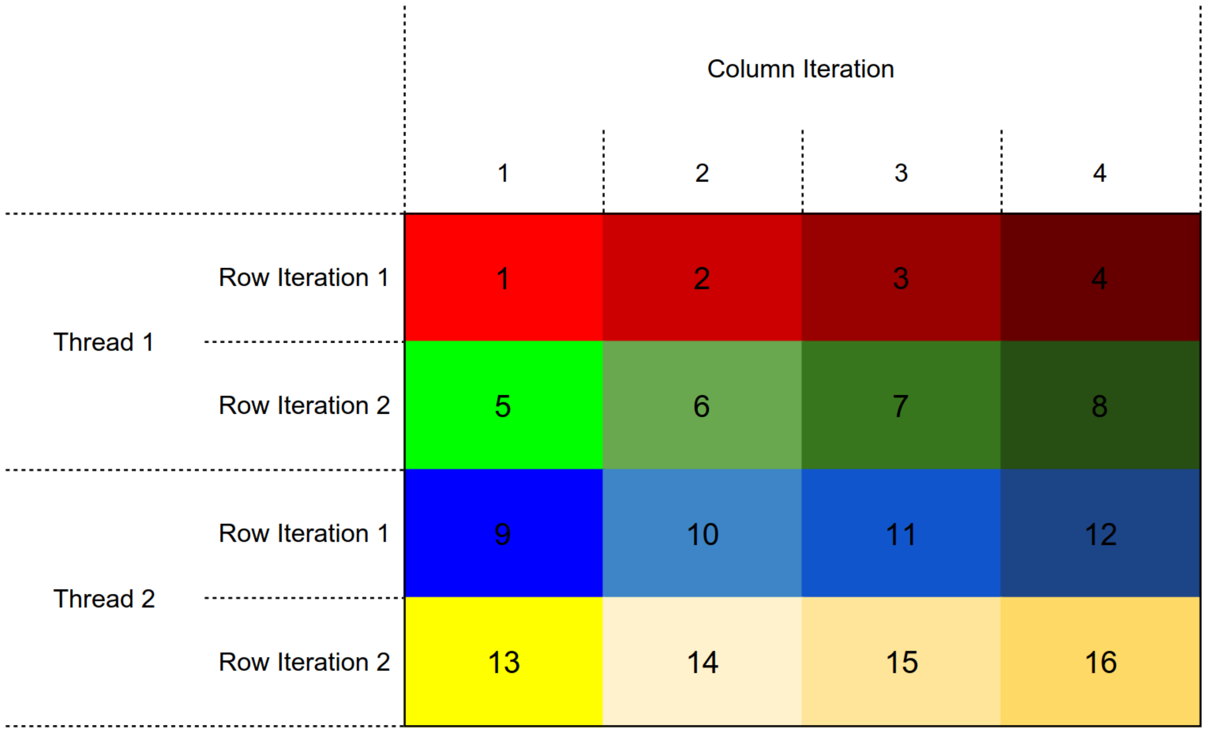


Figure 10: AVX MatVec Matrix Reshape
Author's own work.

Figure 11 provides a detailed view of how each subtile is transmitted. In this example, two column iterations are shown for a single row iteration, with each color representing a different subtile. As illustrated, subtiles are sent in row-major order, while respecting the requirement that each subtile spans two columns in width, in accordance with the constraints of the vectorized function.

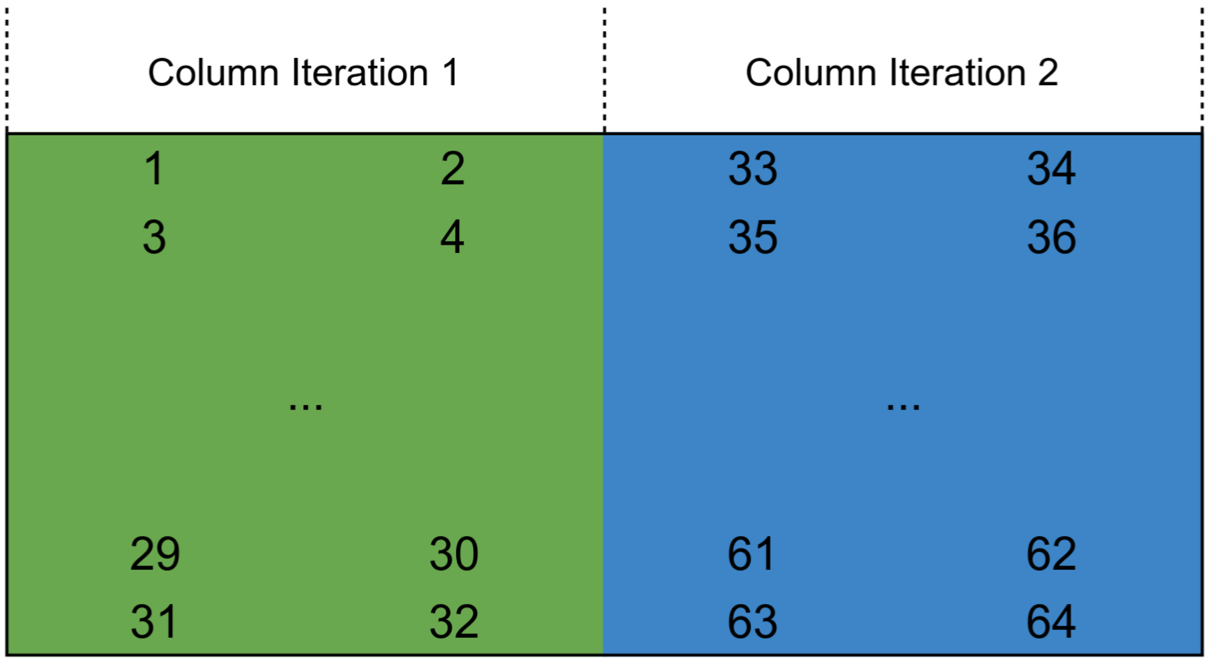


Figure 11: AVX MatVec Matrix Thread Reshape

Author’s own work.

Finally, the previously described data transfer pattern, when expressed in terms of reshaped dimensions, corresponds to the following pairs:

$$P_4 = (\tfrac{rows}{16},\ 16 \times columns)$$
$$P_3 = (\tfrac{columns}{2},\ 2)$$
$$P_2 = (16,\ columns)$$
$$P_1 = (2,\ 1)$$

Note: The value 2 represents the number of columns in each subtile, while 16 corresponds to the number of rows.

Input Vector:
Since the input vector is accessed linearly and each thread can freely access any memory location as needed — unlike the NPU, which receives data as a stream — no reshaping is required for the input vector.

Output Vector:
The output vector does not require reshaping. Because the matrix row distribution is contiguous across threads, and each thread writes its results sequentially, the final output is already correctly laid out in memory.

5.3.3 Size Rounding:

As with the NPU implementation, the data distribution requires the matrix to satisfy the following: $rows\ mod(16 \times n_{threads}) == 0$ and $columns\ mod\ 2 == 0$. The original matrix, with 17,048 rows, does not meet this requirement. Therefore, we expand the matrix by padding with zeros to reach the nearest multiple of $16 \times n_{threads} = 16 \times 16 = 256$. In this case, the matrix is reshaped from 17048×17048 to 17152×17048.

6 Matrix-Matrix Acceleration

Matrix–vector multiplication is the critical operation in the entire economic model. However, during the course of this research, matrix–matrix acceleration emerged as a potentially valuable optimization strategy. Specifically, instead of computing one guess at a time, we propose evaluating multiple well-chosen random guesses in parallel, which might converge faster. In this case, the input is a matrix rather than a vector, with each column representing a distinct guess. By running several guesses simultaneously, the total number of iterations required to reach a fixed point could be significantly reduced.

If the runtime of matrix–matrix operations remains comparable to that of matrix–vector operations, this approach could further reduce overall execution time. As a result, providing an accelerated matrix–matrix implementation may yield greater performance gains than focusing solely on accelerating matrix–vector multiplication.

As with matrix–vector acceleration, the matrix–matrix implementation should aim to eliminate data dependencies between compute units by assigning each one an independent region of data. This ensures parallel execution without contention or synchronization overhead, thereby avoiding potential slowdowns. To enable a fair comparison with CPU performance, we also implement an AVX-M version of the matrix–matrix operation.

6.1 NPU Implementation

We extend the NPU matrix–vector implementation to support matrix–matrix multiplication. Given a matrix A, an input matrix B, and an output matrix $C = A \times B$, each row of C can be computed independently. In this setup, B is the shared input across all compute cores, as the full matrix B is required to compute each row of A.

Formally, let A_i denote the i-th row of A, and C_i the corresponding row of C. Then:

$$C_i = A_i \times B$$

To exploit this independence, each compute column is assigned a distinct set of contiguous rows from A. This parallel strategy ensures that cores operate without data dependencies, enabling efficient and scalable execution.

As with the matrix–vector implementation, execution must be divided into smaller subtiles to meet memory capacity constraints. Each compute core follows a similar access pattern: it processes a subset of rows from A and iterates over the columns. However, instead of multiplying by a single input vector, the core now multiplies its assigned subtile of A by a submatrix of B, processing multiple columns of B at a time.

Once a core has processed all its assigned rows of A for the current block of columns in B, it returns to the beginning of its assigned rows of A to compute the next set of output columns in C.

Matrix A:

The data distribution strategy illustrated in Figure 5 for the matrix–vector implementation also applies to the matrix–matrix case. However, in this scenario, each core loops back to the beginning of its assigned rows of A after reaching the end, repeating the process until all columns of B have been processed.

As in the matrix–vector implementation, computation within each core is further divided into row iterations and column iterations, as previously illustrated in Figure 6.The execution order remains the same: for each row iteration, all corresponding column iterations are completed before proceeding to the next row iteration.

Matrix B:

The data distribution for matrix B follows a different pattern. As previously noted, all compute cores require access to the entirety of matrix B. For each compute core, to traverse B, a transposed iteration order is used: instead of processing all column iterations before advancing to the next row iteration, all row iterations are processed first, followed by the next column iteration. This allows each row of A to accumulate contributions for one block of columns from B before proceeding to the next block, aligning with the tiling strategy used in matrix–matrix multiplication.

6.1.1 Hardware Acceleration:

To accelerate matrix–matrix multiplication, we leverage the NPU's native support for this operation. As discussed in the NPU Supported Operations Section, the hardware provides native acceleration for matrix–matrix products. Specifically, it supports operations for matrices of sizes 4×8 and 8×4. This native capability is utilized in our implementation to achieve optimal performance.

Additionally, due to memory constraints, the size of each subtile must be carefully selected. Empirical testing identified subtiles of size $(m, k, n) = (32, 32, 16)$ as providing optimal performance. In this configuration, m represents the number of rows of matrix A processed per row iteration, and k denotes the number of columns processed per column iteration. For matrix B, n corresponds to the number of columns processed per column iteration.

6.1.2 Data Reshaping:

As with the previous implementations, a data reshaping step is required. However, due to the increased complexity of matrix–matrix multiplication, this reshaping process is significantly more involved. Unlike the matrix–vector implementation — where reshaping is performed almost entirely on the host — a portion of the reshaping in the matrix–matrix case is offloaded to the NPU. This division reduces overall complexity and alleviates some of the workload from the host system.

As noted in the previous implementations, matrix A only needs to be reshaped once at startup, thereby avoiding significant reshaping overhead during execution. In contrast to the matrix-vector case, matrix B requires repeated reshaping. This introduces a substantial overhead not present in the matrix–vector implementation, primarily due to matrix B's larger size and higher dimensionality.

Input Matrix A:

The general data reshaping strategy for Matrix A will follow the same pattern as in the matrix–vector implementation, as illustrated in Figure 7. The traversal remains analogous; however, a key difference arises due to the nature of matrix–matrix multiplication. Specifically, after Matrix A is sent to a given AIE compute column, it must be resent multiple times — once for each column iteration of Matrix B. This is because, as previously mentioned, each row in Matrix A must be multiplied by every column in Matrix B.

As shown in Figure 12, each compute column receives all its required subtiles repeated as many times as necessary before moving on to the next compute column. This example depicts a case where each subtile needs to be sent twice — Matrix B has two column iterations. This approach ensures that each compute column, which is associated to a distinct *ShimDMA*, receives its full data in a single, contiguous transfer. By doing so, the *ShimDMA* can perform a single fetch without jumping across memory regions. This not only simplifies the *ShimDMA* transfer logic but also avoids generating overly complex code that the MLIR-AIE compiler might fail to process.

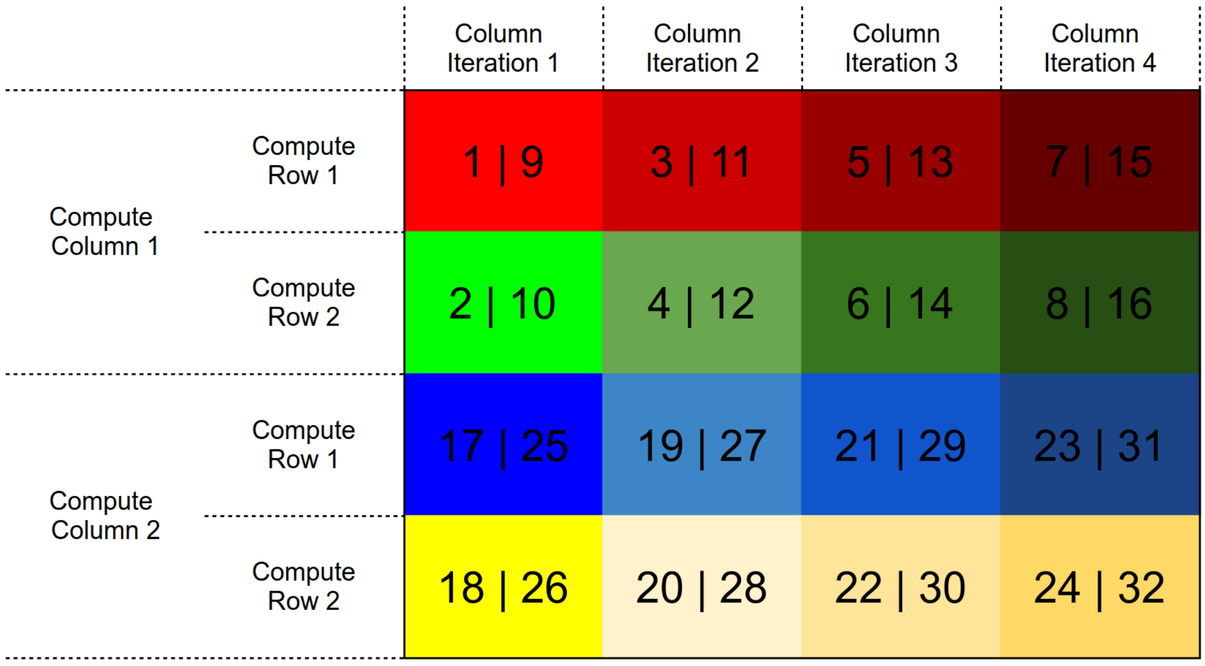


Figure 12: NPU MatMat Matrix A Reshape
Author's own work.

In matrix–matrix multiplication, the subtile must also be reshaped. In contrast to the matrix-vector implementation, where the subtile is stored in row-major order, we must consider that the hardware acceleration function expects input matrices of shape $(r, s) = (4, 8)$. In our reshaping strategy, the dimension r is irrelevant, as we compute all matrix multiplications for a group of s columns before advancing to the next group of s columns.

Specifically, we operate on subtiles of size $(m, k) = (32, 32)$, and traverse them in column-major order with respect to the internal 4×8 blocks expected by the hardware. Figure 13 provides a detailed view of this internal subtile reshaping. As shown, the $m \times k$ subtile is internally tiled into multiple $m \times s$ matrices to match the required input format.

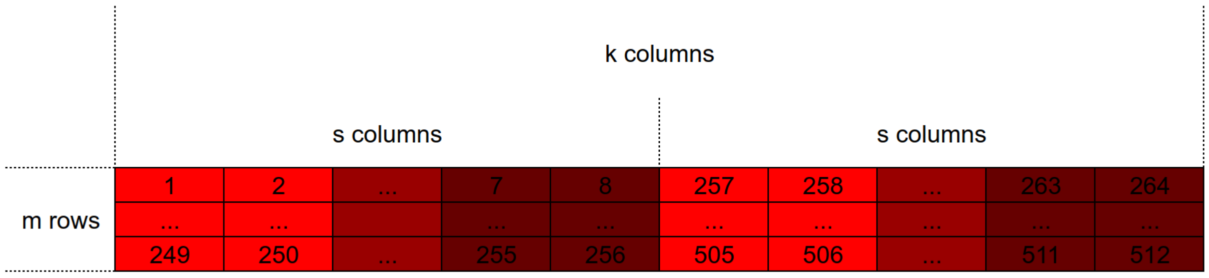


Figure 13: NPU MatMat Matrix A Subtile Reshape
Author's own work.

Note: Only 16 columns are displayed for k for simplicity, instead of 32.

The previous data pattern, when expressed in terms of reshaped dimensions, corresponds to the following pairs:

$$P_8 = (n\_aie\_columns,\ \frac{rows_A \times columns_A}{n\_aie\_columns})$$
$$P_7 = (\frac{columns_B}{n},\ 0)$$
$$P_6 = (\frac{rows\_A}{m \times n\_cores},\ m \times n\_aie\_rows \times columns_A)$$
$$P_5 = (\frac{columns\_A}{k},\ k)$$
$$P_4 = (n\_aie\_rows,\ m \times columns_A)$$
$$P_3 = (\frac{k}{s},\ s)$$
$$P_2 = (m,\ columns_A)$$
$$P_1 = (s,\ 1)$$

Input Matrix B:

The data reshaping strategy for Matrix B is similar to that of Matrix A, with the key difference being that Matrix B is transposed. As previously mentioned, we first send all row iterations corresponding to a single column iteration. This is then repeated as many times as needed — as many as different row iterations of Matrix A each compute core must process. Once all required repetitions for the current column iteration are finished, we proceed to the next column iteration.

Figure 7 illustrates an example in which each subtile of Matrix B must be sent twice — two row iterations of matrix A per compute core — in a configuration with 4 column iterations and 4 row iterations.

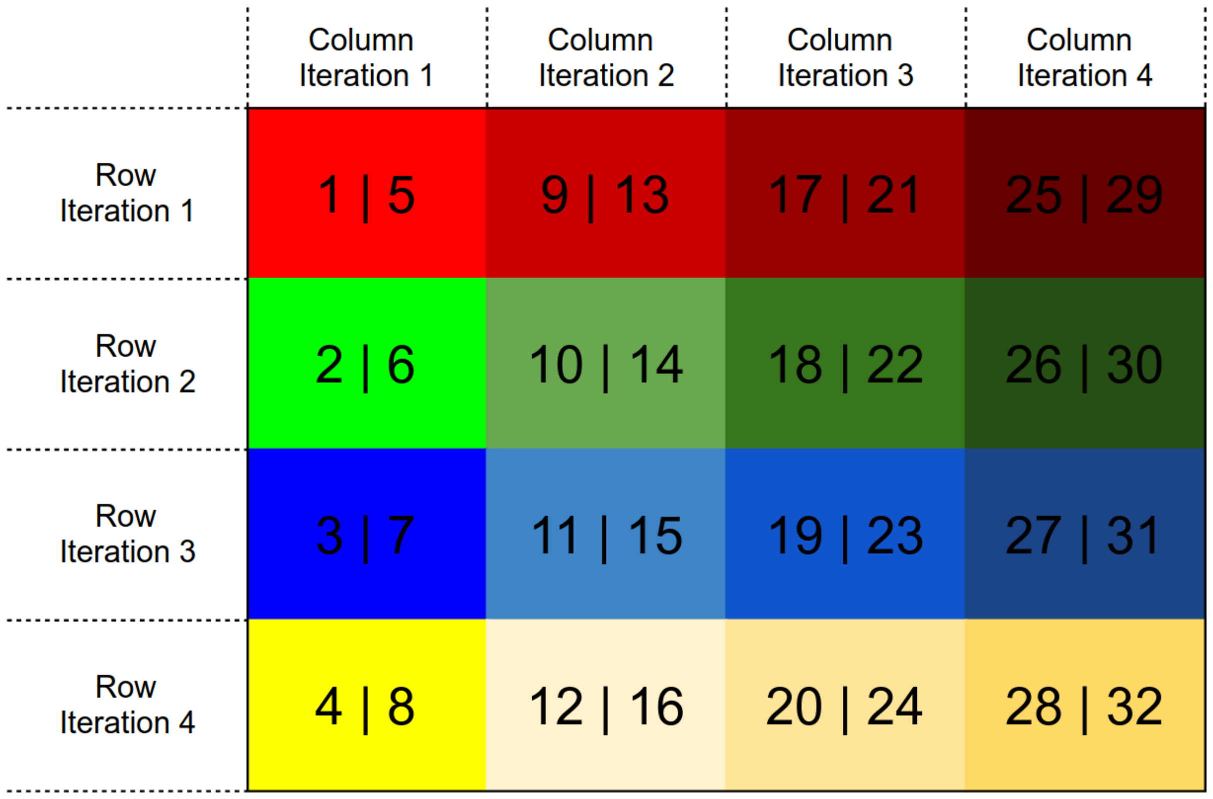


Figure 14: NPU MatMat Matrix B Reshape
Author's own work.

As with matrix A, the subtile of matrix B must also be reshaped. The hardware acceleration function expects input matrix of shape $(s, t) = (8, 4)$. In this reshaping strategy, the s dimension is not relevant, as all matrix multiplications are performed for a group of t columns before moving to the next group.

Specifically, we operate on subtiles of size $(k, n) = (32, 16)$, which are traversed in column-major order relative to the internal 8×4 blocks expected by the hardware. Figure 15 provides a detailed view of this internal subtile reshaping. As shown, the $k \times n$ subtile is internally tiled into multiple $k \times t$ matrices to match the required input format.

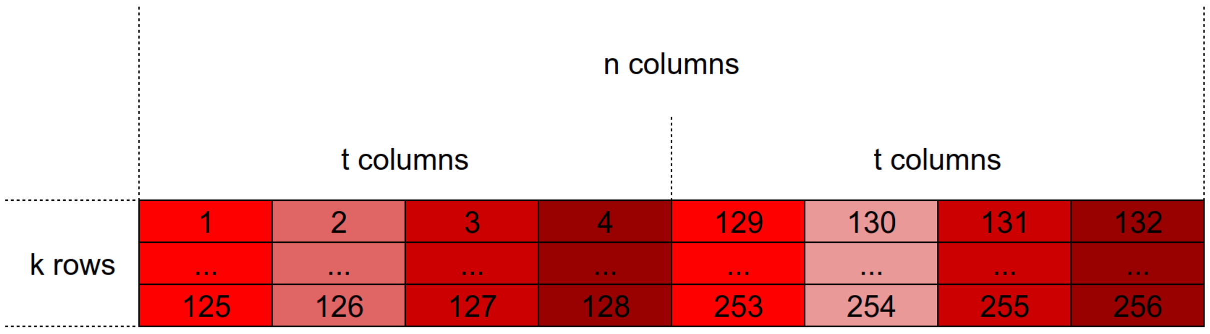


Figure 15: NPU MatMat Matrix B Subtile Reshape
Author's own work.

Note: Only 8 columns are displayed for n for simplicity, instead of 16.

The previous data pattern, when expressed in terms of reshaped dimensions, corresponds to the following pairs:

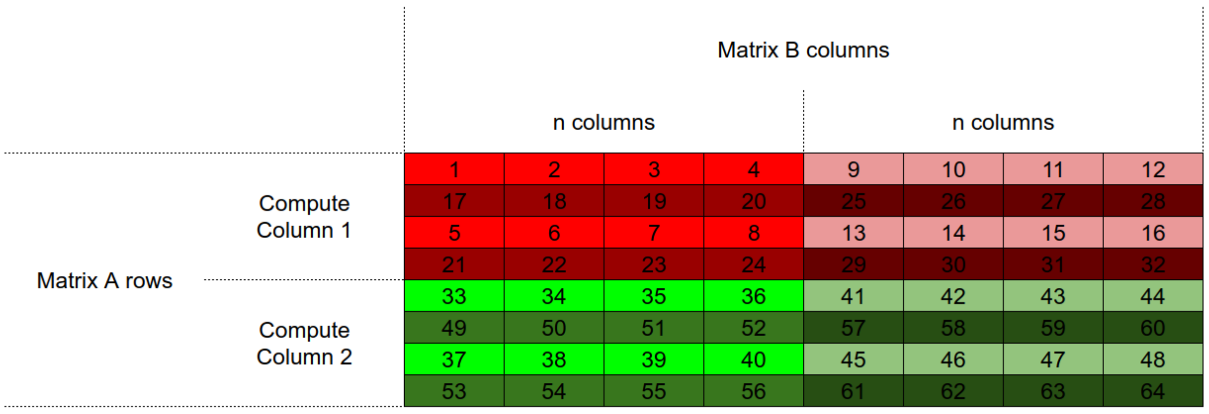
$$\begin{aligned}
P_6 &= (\tfrac{columns_B}{n},\ n)\\
P_5 &= (\tfrac{rows_A}{m \times n_{cores}},\ 0)\\
P_4 &= (\tfrac{columns_A}{k},\ k \times columns_B)\\
P_3 &= (\tfrac{n}{t},\ t)\\
P_2 &= (k,\ columns_B)\\
P_1 &= (t,\ 1)
\end{aligned}$$

Output Matrix:
Since the output is produced in subtiles that do not match the output matrix, reshaping of the output is needed. As previously mentioned, there is reshaping on both the host and the NPU.

The host will see the NPU produce a matrix of size $rows_A \times columns_B$. However, this matrix will have been produced in subtiles of size $(m, n) = (32, 16)$. Therefore, the host will need to reshape it from the multiple 32×16 blocks into a contiguous $rows_A \times columns_B$ matrix, laid out in a row-major order.

Each compute column will first produce all the rows for a set of n columns before moving on to the next set of n columns. Therefore, for a specific row, we will have to get the first n columns, go over to the next set of n columns for that specific row, and so on until all columns have been fetched. This process is repeated for every row, taking into account the data layout across different aie columns.

Figure 16 illustrates this in a simplified example. In this example, $(rows_A, columns_B) = (8, 8)$, $n\_aie\_columns = 2$, and $n = 4$.



Compute Column 1	1	2	3	4	9	10	11	12
	17	18	19	20	25	26	27	28
	5	6	7	8	13	14	15	16
	21	22	23	24	29	30	31	32
Compute Column 2	33	34	35	36	41	42	43	44
	49	50	51	52	57	58	59	60
	37	38	39	40	45	46	47	48
	53	54	55	56	61	62	63	64

Figure 16: NPU MatMat Matrix C Reshape
Author's own work.

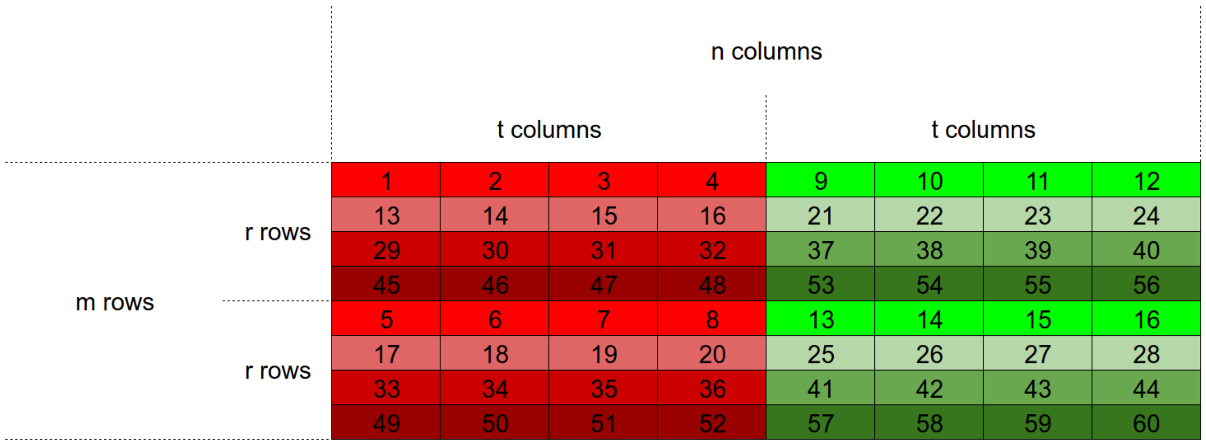
The previous data pattern, when expressed in terms of reshaped dimensions, corresponds to the following pairs:

$$
\begin{aligned}
P_4 &= (n\_aie\_columns,\ \tfrac{rows_A \times columns_B}{n\_aie\_columns}) \\
P_3 &= (\tfrac{rows_A}{n\_aie\_columns},\ n) \\
P_2 &= (\tfrac{columns_B}{n},\ \tfrac{rows_A}{n\_aie\_columns} \times n) \\
P_1 &= (n,\ 1)
\end{aligned}
$$

Analogous to the host, each compute core will have to reshape the output of the hardware acceleration function into its corresponding subtile. Specifically, the compute core will produce a subtile of size $(m, n) = (32, 16)$. However, this subtile will have been produced in chunks of $(r, t) = (4, 4)$, according to the matrix-matrix hardware acceleration function. Therefore, each compute core will have to reshape the produced tile when being sent to the MemTile. The reshape will take the multiple 4×4 blocks into a contiguous 32×16 subtile, laid out in a row-major order.

Figure 17 illustrates this in a simplified example. The matrix output by the hardware acceleration function is of size $(r, t) = (4, 4)$ and each subtile of Matrix C is of size $(m, n) = (8, 8)$.

Note: Each color represents a different row of the subtile.



1	2	3	4	9	10	11	12
13	14	15	16	21	22	23	24
29	30	31	32	37	38	39	40
45	46	47	48	53	54	55	56
5	6	7	8	13	14	15	16
17	18	19	20	25	26	27	28
33	34	35	36	41	42	43	44
49	50	51	52	57	58	59	60

Figure 17: NPU MatMat Matrix C Subtile Reshape
Author's own work.

The previous data pattern, when expressed in terms of reshaped dimensions, corresponds to the following pairs:

$$
\begin{aligned}
P_3 &= (m,\ t) \\
P_2 &= (\tfrac{n}{t},\ m \times t) \\
P_1 &= (r,\ 1)
\end{aligned}
$$

6.1.3 Size Rounding:

As in the matrix-vector implementation, matrix A has to meet the following conditions: $rows\ mod(m \times n_{cores}) == 0$ & $columns\ mod\ k == 0$. In addition, matrix B will have to fulfill the following condition $columns\ mod\ n == 0$. Following the previous implementation, the matrices are reshaped from 17048×17048 and 17048×16 to 17408×17056 and 17056×16.

6.1.4 Data Transfer:

Similar to the matrix-vector implementation, the matrix-matrix implementation's host reshape will allow data to be consumed and produced linearly. The following holds true:

Matrix A:

- **ShimDMA:** For its aie column, it will transfer $\frac{columns_B}{n} \times \frac{rows_A \times columns_A}{n\_aie\_columns}$ contiguous elements from the host to the corresponding compute column.
- **MemTile:** For every $m \times k \times n\_aie\_rows$ elements received from the *ShimDMA*, it will divide them into segments of $m \times k$ elements and send one segment to each compute row in its compute column.

Matrix B:

- **ShimDMA:** For its aie column, it will transfer the entirety of Matrix B, including its repetitions, to its respective MemTile. This will be equivalent to $\frac{rows_A}{m \times n\_cores} \times columns_B \times rows_B$ elements.
- **MemTile:** For every $k \times n$ elements received from the *ShimDMA*, it will broadcast them into each compute core in its aie column.

Output Matrix:

- **MemTile:** For every $m \times n$ elements received from each compute core in its aie column, it will join them to produce $m \times n \times n\_aie\_rows$ elements and send them to its corresponding *ShimDMA* tile.
- **ShimDMA:** For its aie column, it will transfer the elements received from the MemTile into a contiguous $\frac{rows_A \times columns_B}{n\_aie\_columns}$ elements to the host.

6.2 AVX + Multithreading Implementation

The AVX-M matrix–matrix implementation extends from the AVX-M matrix–vector approach. Unlike the NPU implementation, whose reshaping complexity increases significantly, the AVX-M version simply replaces the input vector with a matrix. The strategy remains the same: leveraging the independence of row-wise computations in Matrix A. Each thread is assigned a distinct set of consecutive rows of A to process.

Building upon the previous matrix–vector implementation, the next logical step is to compute matrix A for each column of matrix B sequentially. That is, once all rows of matrix A have been processed for one column of B, the computation proceeds to the next column of B. This approach is optimal, preserving the thread independence built on the matrix-vector implementation.

However, due to the lack of native matrix–matrix acceleration, we will still need to use AVX dot-product instruction. This will limit us to process one column of matrix B at a time. As a result, the total computation time will grow linearly with the number of columns in B. There is a positive side-effect, which is the aforementioned lack of increase in reshape complexity.

Within each thread, the computation process works as an extended matrix-vector computation. In this sense, the computation process repeats the matrix-vector process for each column of B. The data distribution for A follows the matrix-vector implementation, as outlined in Figure 9.

6.2.1 Hardware Acceleration:

As previously mentioned, AVX extensions lack support for matrix-matrix acceleration. Therefore, we will need to reuse the matrix-vector acceleration approach. We will be using the same vector dot product acceleration:

```
__m512 _mm512_dpbf16_ps(__m512 __C, __m512bh __A, __m512bh __B)
```

6.2.2 Data Reshaping:

As previously mentioned, processing one column of matrix B at a time prevents an increase in reshaping complexity. In specific, by extending the matrix-vector multiplication, we will be generating one column of the output matrix at a time. This will have two implications:

- **Matrix B:** It will need to be stored in column-major.
- **Output Matrix:** It will be produced in column-major order, requiring a transposition to row-major.

Specifically, the transposition consists of the following dimension pairs:

$$P_2 = (columns,\ 1)$$
$$P_1 = (rows,\ columns)$$

6.2.3 Size Rounding:

Since the matrix-matrix implementation is just an extension of the matrix-vector implementation, it must satisfy the same two conditions: $rows_A mod(16 \times n_{threads}) == 0$ & $columns_B mod 2 == 0$. Thus, matrix B is expanded from 17048×17048 to 17152×17048.

7 Results

7.1 Implementation Verification

To verify the correctness of both the NPU and AVX-M implementations, we generate pseudo-random matrices, vectors and compare the output against a baseline scalar CPU implementation. This process is repeated across various matrix sizes and shapes.

For the NPU, we test different configurations by varying the number of compute cores — adjusting the number of compute columns and rows — and exploring multiple subtile sizes. Similarly, for the AVX-M implementation, we evaluate different thread counts and matrix dimensions. If all tests produce results consistent with the baseline, we assume the implementation is correct.

7.2 Benchmarks

After completing the various implementations, we measured several metrics to compare their performance. These measurements include compilation time, reshape time, and runtime — each providing insight into different aspects of efficiency and practicality.

Before the actual model evaluation, we have ran tests focused on the matrix-vector and matrix-matrix multiplication operation itself. This has allowed us to analyze the computational characteristics of the critical operation in isolation. Afterwards, we have put these results into context by benchmarking the performance of the operation within the critical function of the model.

7.2.1 Compilation Time

As discussed in the background section, compilation time is an important factor in the development process. While it is often overlooked, compilation on platforms like FPGAs can take several hours or even days, whereas the same program might compile in minutes for a CPU. For this reason, we include compilation time as a key performance metric in our evaluation.

Table 7 presents the compilation times for different implementations and devices. It is worth noting that matrix size can influence the NPU's image compilation time. In this evaluation, matrix A has dimensions 17408×17408, and matrix B is 17408×16.

	CPU	**AVX**	**NPU MatVec**	**NPU MatMat**
Program	0.807s	1.2s	2.66s	2.66s
Image	-	-	4.350s	4.315s

Table 7: Compilation Time
Author's own work

As shown in the table, although the compilation time is comparatively higher for the NPU and AVX implementations, it remains under 10 seconds in all cases. Furthermore, this increase in compilation time is expected to remain constant and not scale significantly with program complexity. Therefore, compilation time is not a limitation for either of the accelerated implementations.

7.2.2 Run Time

To analyze the potential speed-up across the different implementations, we have performed the matrix-matrix multiplication with varying numbers of columns for matrix B. All the measurements in this section are solely of the computational time, they do not include reshape or buffer load time. Table 8 provides the results for these tests. It is important to consider the specific characteristics of each implementation, as discussed in the size rounding section of each implementation.

- **CPU:** Matrix-matrix multiplication of size 17048×17048 by $17048 \times N$. Elements are of FP64 Type.
- **AVX-M:** Matrix-matrix multiplication of size 17408×17048 by $17048 \times N$. We are using 16 threads. Elements are of BF16 Type.
- **NPU:** Matrix-matrix multiplication of size 17408×17056 by $17056 \times N$. Since the matrix-matrix hardware acceleration requires $n \geq 4$, an extension of the matrix-vector implementation was used for $N = 2$. Subtiles were defined with dimensions $(m, k, n) = (32, 32, N)$. We are using 4 AIE compute columns and 4 AIE compute rows for each column, totaling 16 compute cores. Elements are of BF16 Type.

N	**1**	**2**	**4**	**8**	**16**	**32**
CPU	2800ms	5650ms	11315ms	22620ms	45195ms	90650ms
NPU	36.5ms	37ms	37.5ms	38.5ms	40.5ms	45ms
AVX-M	11.5ms	23.5ms	46ms	96ms	187.5ms	394ms

Table 8: Matrix-Matrix Operation Runtime
Author's own work

As observed in the table, both the NPU and AVX-M implementations provide a significant speed-up over the scalar CPU baseline. In specific, the AVX-M implementation achieves speed-ups of $\times 243$ and $\times 230$ for $N = 1$ and $N = 32$, respectively. The NPU achieves speed-ups of $\times 78$ and $\times 2014$ for the same values of N.

These results highlight the fact that AVX-M excels at matrix-vector multiplication, outperforming the NPU by a factor of $\times 3.17$. However, AVX-M runtime grows linearly with N, while the NPU runtime grows almost logarithmically, particularly for $N \leq 16$. At $N = 32$, the NPU provides a speed-up of $\times 8.75$ over the AVX-M implementation.

Subtile Shape:
As shown in Table 9, the NPU's performance is also affected by the subtile sizes. For specific matrices of size 17408×17056 and $17056 \times N$, the configuration of subtiles

— defined as $(32, 32, n)$ — has a significant impact on runtime. In particular, the relationship between p (matrix B subtile columns) and P (matrix B columns) greatly affects the runtime.

N	(32, 32, 16)	(32, 32, 32)	(32, 32, 64)
16	40.5ms	—	—
32	81.5ms	45ms	—
64	—*	90.5ms	54.5ms

Table 9: Subtile Shape Runtime
Author's own work

***Note:** Due to matrix A's repetition, the required buffer size exceeds the maximum buffer size. More information on matrix A's repetition can be found on the NPU Matrix-Matrix Implementation | Matrix A Reshape Section.

The results clearly indicate that keeping n constant causes runtime to increase linearly with N. Specifically, the increase is 41ms for $N = 16 \to N = 32$ and 45.5ms for $N = 32 \to N = 64$. In contrast, if we match $n = N$, the runtime increase is significantly smaller — 4.5ms and 9.5ms, respectively.

This information helps us understand that the subtile shape is almost as important as the actual matrices' sizes. With poorly chosen subtile configurations, the operation could be almost twice as slow.

Computational Parallelism:
When parallelizing the computation across different NPU columns, the parallelism appears to be ideal. Table 10 shows the runtime for matrix-matrix multiplications with dimensions 17408×17056 and $17056 \times N$, using only one AIE compute row per column. This configuration allows us to effectively analyze the parallelism across different compute columns.

AIE Columns	1	2	4	8	16	32	64
1	145ms	243ms	145ms	145ms	210ms	382ms	710ms
2	72.5ms	120ms	72.5ms	73ms	105ms	191ms	355ms
4	37.5ms	61ms	41ms	45.5ms	54.5ms	95.5ms	178ms

Table 10: NPU Column Parallelism
Author's own work

We have also parallelized the computation across different NPU rows. Using only one compute column, we have varied the amount of compute rows, analogous to the AIE column parallelization. Table 11 shows the runtime for matrix-matrix multiplications with dimensions 17408×17056 and $17056 \times N$.

AIE Rows	1	2	4	8	16	32	64
1	145ms	243ms	145ms	145ms	210ms	382ms	710ms
2	145ms	145ms	145ms	145ms	145ms	190ms	356ms
4	145ms	145ms	145ms	145ms	145ms	145ms	175ms

Table 11: NPU Row Parallelism
Author's own work

Contrary to column parallelism — which is always ideal — there seems to be no row parallelism until $N = 16$. Specifically, no significant row parallelism is observed until $N = 64$. It is also interesting that some row parallelism is present at $N = 2$, but not at $N = 4$. This difference is is probably related to different implementations. For $N = 2$, we are using the extended matrix-vector implementation — Appendix D —, while for $N \geq 4$ we are using the matrix-matrix implementation.

Computational Analysis:

With all the previous results into perspective, we can now assess the performance characteristics of the NPU. First of all, the AVX-M implementation outperforms the NPU in matrix-vector multiplications. As previously mentioned, AVX-M provides a speed-up of $\times 3.17$ over the NPU. However, in matrix-matrix multiplications, the NPU outperforms AVX-M with a speed-up of $\times 8.75$.

These results align with the general purpose of both architectures. AVX extensions are optimized for vector operations, as their name suggests, making them well-suited for tasks such as matrix-vector multiplication. In contrast, the NPU is optimized for matrix-oriented workloads, providing hardware acceleration for matrix-matrix computation.

As shown in Table 8, for values of $N<16$, runtime grows almost logarithmically. In addition to this, it seems that the runtime is not row parallelizable for $N \in \{1, 4, 8\}$. To better understand this behavior, we must examine the computational characteristics of the NPU. Given a subtile $(m, k, n) = (32, 32, n)$, table 12 provides us the data-movement and computational cost per output element.

- $\frac{IN}{OUT}$: Number of input elements consumed per output element.

$$\frac{m \times k + k \times n}{m \times n}$$

- $\frac{OP}{OUT}$: Total operations per output element.

$$\frac{ADD}{OUT} + \frac{MULT}{OUT}$$

- $\frac{ADD}{OUT}$: Number of additions per output element.

$$\frac{m \times k \times n}{m \times n}$$

- $\frac{MULT}{OUT}$**:** Number of multiplications per output element.

$$\frac{m \times (k-1) \times n}{m \times n}$$

	$\frac{IN}{OUT}$	$\frac{OP}{OUT}$	$\frac{ADD}{OUT}$	$\frac{MULT}{OUT}$
Matrix-Vector	33	31.5	15.5	16
Matrix-Matrix	2			

Table 12: Matrix-Vector vs Matrix-Matrix Characteristics
Author's own work

As it can be observed in the table, the only difference between a matrix-vector and a matrix-matrix multiplication lies in the amount of data consumed per output element. The computational cost itself stays the same. For the example subtile, there is a $\times 16.5$ increase in the amount of elements that need to be consumed. Furthermore, $\frac{IN}{OUT}$ grows linearly with k in the matrix-vector operation, whereas it grows linearly with $\sqrt{k}$ for the matrix-matrix operation.

Since data transfer imbalance is the primary bottleneck at lower values of N, increasing N just serves to even out the data movement across inputs. As the subtile multiplication approaches a square matrix-matrix multiplication, the data needed to be consumed from each input is balanced. As this balance is achieved, the total computation time barely increases.

These results also explain the "strange" behavior displayed by the row and column parallelization. The behavior can be better understood through the NPU's architecture, as shown in Figure 4. In specific, all rows within a compute column share the same *ShimDMA* and MemTile, meaning they rely on a the same path to fetch data. In contrast, different compute columns have their own *ShimDMA* and MemTile, eliminating contention for data access across columns. This architectural characteristic has the following implications:

- As N increases, $\frac{IN}{OUT}$ decreases substantially, reducing the impact of data transfer. As the data movement is no longer a bottleneck, the workload becomes parallelizable across compute rows.
- Since compute columns do not share memory access resources, there is no competition to fetch data. This makes column parallelization always ideal.

The spike in row parallelism at $N = 2$ can be attributed to the use of the extended matrix-vector implementation. This implementation increases computational time linearly with N. The elevated computational load allows row parallelism to provide some performance improvement. However, when switching to the hardware accelerated matrix-matrix multiplication at $N = 4$, the computational time is substantially reduced. This shifts the bottleneck back to the data transfer, which prevents row parallelism from providing speed-up.

Smaller Matrix:
For smaller matrices, the observed computational speed-ups differ from those seen in larger matrices. Table 13 provides an overview of performance results for these smaller matrices. In specific, we have run tests for matrix-vector multiplications. In these tests, the NPU uses a subtile of size $(m, k) = (32, 32)$.

	512x512	1024x1024	1536x1536	2048x2048	4096x4096
CPU	2530us	10135us	22810us	40550us	162310
AVX-M	283us	290us	300us	335us	764us
NPU	120us	215us	380us	600us	2140us

Table 13: Small Matrix-Vector Runtime
Author's own work

For the smaller matrices, it can be observed that AVX-M runtime increases only slightly, while the NPU runtime grows linearly. Interestingly, it seems the NPU is faster at sizes 512×512 and 1024×1024. The CPU, as expected, remains significantly slower than both the AVX-M and NPU implementations.

We do not attribute this difference to the NPU being faster, but to AVX-M being slower. In this sense, AVX-M appears to require a larger size to begin to pipeline and parallelize execution. For smaller matrices, the overhead of starting the threads and computation outweights the potential performance benefits. In contrast, the NPU has no added computational overhead, so it can start processing data at full speed as soon as it is available.

7.2.3 Reshape Time

Reshape time overhead must also be taken into account, as it can significantly degrade performance due to its complexity, even exceeding the actual computational time. Since the reshape required for matrix-vector and matrix-matrix implementations is different, we have analyzed both separately.

Matrix-Vector:
Table 14 presents the reshape time for the matrix-vector implementation on both, AVX-M and NPU. The matrix dimensions were adjusted to meet the requirements of each specific implementation, as highlighted in the respective *Size Rounding* sections. The sizes are the following:

- **AVX-M:** 17152×17048 and 17048×1.
- **NPU:** 17408×17056 and 17056×1. Subtiles of size $(m, k) = (32, 32)$

	AVX-M	NPU
Matrix	1630ms	1030ms
Input Vector	0.055ms	0.085ms

Table 14: Reshape Matrix-Vector Time
Author's own work

As observed, reshaping the matrix has a much higher cost than the input vector. This is expected, as the input vector reshape only consists of sending it multiple times. The matrix's cost, however, does not affect the actual computational time. As previously mentioned, the input matrix only needs to be reshaped once at the beginning, since it does not change throughout the model's runtime. The input vector's cost is negligible, representing an added 0.48% and 0.23% to the computational time of AVX-M and NPU, respectively.

Matrix-Matrix:
Table 15 presents the reshape time for the matrix-matrix implementation. As with the matrix-vector implementation, the matrix dimensions were adjusted to meet the requirements of each specific implementation. The sizes are the following:

- **AVX-M:** 17152×17048 and 17048×16.
- **NPU:** 17408×17056 and 17056×16. Subtiles of size $(m, k, n) = (32, 32, 16)$

	AVX-M	NPU
Matrix A	1635ms	970ms
Matrix B	0.870ms	0.935ms
Output Matrix	0.910ms	1.670ms

Table 15: Reshape Matrix-Matrix Time
Author's own work

As observed, reshaping A still has a much higher cost than reshaping B, due to A's higher complexity reshape. In this case, however, the cost of reshaping the output and B has increased. Combined, the output and B represent an added 0.95% and 6.4% to computational time of the AVX-M and NPU implementations, respectively. Even though the increase on the NPU does not significantly affect runtime, it would now become noticeable.

7.2.4 Computing Resources Usage

The usage of computational resources is another important aspect to consider, particularly in terms of memory and CPU utilization. These two aspects can significantly impact the requirements and scalability of the implementation.

Memory Usage:

Regarding memory, the utilization for each implementation is as follows:

- **CPU:** 2.35GB.
  - **Matrix:** 17048×17048 8B elements $\longrightarrow$ 2.35GB.
  - **Vector:** 17048 8B elements $\longrightarrow$ 136.4KB.
  - **Result:** 17048 8B elements $\longrightarrow$ totaling 136.4KB.
- **AVX-M:** 585MB.
  - **Matrix:** 17152×17048 2B elements $\longrightarrow$ totaling 585MB.
  - **Vector:** 17048 2B elements $\longrightarrow$ totaling 34.1KB.
  - **Result:** 17152 4B elements $\longrightarrow$ totaling 68.6KB.
- **NPU:** 595MB.
  - **Matrix:** 17408×17056 2B elements $\longrightarrow$ totaling 594MB.
  - **Vector:** 34 repetitions — reshape for 16 compute cores — of 17056 2B elements $\longrightarrow$ 1.16MB.
  - **Result:** 17408 4B elements $\longrightarrow$ 69.63KB.

Due to the reduced data type, both the AVX-M and NPU implementations use $\times 4$ less memory than the CPU, despite holding slightly more elements. This is the expected result, since BF16 takes $\frac{1}{4}$ of the space of FP64.

However, the NPU implementation's memory usage is highly sensitive to the number of columns in matrix B. This is due to matrix A's reshape repetition requirements. In specific, matrix A would be repeated $\frac{columns_B}{columns_{subtileB}}$. Since memory usage is effectively dictated by matrix A, the memory usage would be increased as many times as A would be repeated. For example, having $\frac{64}{16} = 4$ would require $\times 4$ the memory for matrix A. This would see an NPU usage of 2.38GB, higher than that of the CPU with FP64.

CPU Usage:

To analyze the CPU usage, we ran 10,000 iterations of the matrix-vector multiplication implementation. Based on the collected data, we constructed Table 16. This table reports the average CPU usage, the maximum available CPU usage ($max_{cores} \times threads_{core}$), the total execution time, and the total CPU usage.

Although total CPU usage is not a standard metric, it offers a better perspective of the overall CPU usage. By combining both CPU usage and time, this metric allows for a more meaningful comparison of runs with different durations. In specific, this metric is computed as the product of the average CPU usage and the total execution time.

	CPU Usage %	Max CPU Usage %	Total Time	Total CPU Usage
CPU	99%	1600%	28179s	2789721%s
AVX-M	1245%		268.17s	333872%s
NPU	1%		383.74s	383.74%s

Table 16: CPU Usage
Author's own work

As expected, the baseline CPU implementation utilizes a single core thread, while the NPU implementation uses only about 1% of a single core thread. Interestingly, the AVX-M implementation, despite using 16 threads, does not fully saturate the CPU. It only reaches around 1245%, instead of the maximum 1600%.

This discrepancy is likely due to the program having to yield to other processes. Since the operating system kernel and background applications are also running, they prevent the AVX-M implementation from occupying the entire CPU. It may be possible, under different system conditions, for AVX-M to achieve a higher CPU utilization.

When analyzing the total CPU usage, we can see the previous information into perspective. Although the NPU takes $\times 1.43$ longer than the AVX-M implementation, its total CPU usage is $\times 870$ lower.

In contrast, the baseline CPU implementation takes $\times 104$ longer than AVX-M and $\times 73$ longer than the NPU. Since it uses an entire core thread for its extended duration, it has a total CPU usage $\times 8.36$ higher than AVX-M and $\times 7824$ higher than the NPU.

Overall, the NPU implementation is the most effective at offloading computation. This makes it ideal when maintaining multitasking and system responsiveness is a priority, as it has no impact on the performance of other processes. In comparison, the AVX-M implementation is highly CPU intensive, potentially degrading the performance of other concurrent processes. While the baseline CPU implementation would provide a balanced approach, its significantly longer execution time makes it a less practical implementation.

7.3 Model Evaluation

Once the performance analysis of each implementation is complete, we can integrate them into the actual model. For a fair comparison, each implementation is integrated with the model according to its computational characteristics.

For the **CPU**, we preserve the original implementation using FP64. This decision is based on the lack of hardware support for BF16 operations, which require emulation and thus run significantly slower. Additionally, FP64 operations run as fast as FP32 operations, making FP64 a fairer point of comparison due to its higher precision. The implementation can be found in Appendix E.1.1.

The **AVX-M** implementation is integrated into the original CPU version by replacing the matrix-vector multiplication step with an AVX-M accelerated version using BF16. This change enables higher parallelism and increases the overall throughput. The implementation can be found in Appendix E.1.2.

For the **NPU**, we do not directly integrate it into the original CPU version. Instead, we redesign the fixed point search algorithm to take advantage of the NPU's strength in matrix-matrix operations. Rather than using a single initial guess, we generate 16 concurrent random guesses, allowing the model to leverage the NPU's strength. The implementation can be found in Appendix E.1.3.

However, the random generation of the guesses needs careful construction. Analysis of the fixed point elements revealed significant variation in their orders of magnitude. To address this, we implemented a bounded randomization strategy: each guess is scaled relative to the magnitude of a known fixed point. Specifically, we use the values from the first fixed point, which will be computed using the CPU FP64 implementation.

The generation rule is defined as follows:

Given $g_j, p \in \mathbb{R}^{17048}$, where $j \in \mathbb{N} \cap [1, 16]$, p is the first fixed point, and g_j is j-th guess, the values are drawn as:

$$g_i^j = \begin{cases} \text{rand}(0, p_i) & \text{if } j \leq 8 \\ \text{rand}(0, 2 \times p_i) & \text{if } j > 8 \end{cases}$$

This strategy maintains variability in the values while ensuring they stay within reasonable bounds. This should promote faster convergence during model execution.

7.3.1 Execution Results

Table 17 provides an overview of the model execution performance. Specifically, we profiled a call to function *loop_uhat* and measured the time required to reach the fixed point. The table provides the total number of iterations, the total execution time, and the average time per iteration.

	Total Iterations	**Total Time**	**Time Per Iteration**
CPU	29	23.33s	804ms
AVX-M	24	0.389s	16.21ms
NPU	20	1.219s	60.95ms

Table 17: Model Execution
Author's own work

As observed, the amount of iterations required to reach the fixed point varies across implementations. We attribute the reduced iterations in AVX-M compared to the CPU to the loss of precision when transitioning from FP64 to BF16. The NPU also achieves a lower iteration count than AVX-M, which is attributed to computing multiple guesses.

Specifically, the NPU shows a 20% decrease in total iterations over the AVX-M implementation.

When examining average iteration duration, AVX-M provides the most substantial improvement, achieving a $\times 49.6$ speed-up over the CPU. The NPU, while also faster than the CPU, only provides a $\times 13.19$ improvement. As a result, each NPU iteration is $\times 3.76$ slower than an AVX-M iteration.

Overall, the AVX-M implementation clearly provides the fastest execution. Although the NPU benefits from fewer iterations, its slower speed results in longer execution time compared to AVX-M — though it still outperforms the baseline CPU implementation.

7.3.2 Development Complexity

Throughout the development process, it has become evident that developing for the NPU involves a high level of complexity. One of the primary challenges was in learning the new MLIR-AIE framework, which requires both *Python* and *C++* development. The particularly complicated aspect was the dataflow construction and the data reshaping.

Getting started with MLIR-AIE took only a few days. I was able to grasp the C++ kernel programming relatively quickly due to its straightforward nature. However, gaining a deeper understanding of the MLIR-AIE framework took around four weeks. Particularly, understanding the dataflow design and data reshape proved to be significantly more complex.

To fully understand this dataflow design and data reshape, I needed to use visual aid tools. Tools such as *André Rösti's Data Layout Visualizer[28]* proved crucial in understanding how the data was being transferred, which was the most complicated aspect.

The integration of the NPU implementation into the model was the easiest part. The operations can be conveniently wrapped within a class, exposing simple interaction methods, such as *InitializeNPU*, *LoadMatrix*, or *RunIteration*. Due to the generic nature of matrix-vector and matrix-matrix operations, this structure could be packaged as a reusable library for future applications.

Overall, while learning NPU development was not prohibitively difficult, it was far from an easy task. I believe that it might become somewhat challenging for those who are not proficient in high performance computing or computer science. However, if a reusable library were built, integration of the NPU into different models would become much more accessible.

In contrast to the NPU, the AVX-M implementation was considerably more approachable. The implementation primarily involved picking the correct AVX instructions, understanding their behavior, and designing a simple reshape strategy. For someone with a moderate experience in high performance computing, implementing AVX-M could be done in approximately 8 hours, with time varying based on the developer's familiarity with AVX and multithreading.

8 Conclusions

This project has covered an in-depth analysis of AMD's NPU1, through the acceleration of an economic model. In specific, the project has focused on evaluating the computational constraints of the NPU, while comparing its performance to an AVX + Multithreading (AVX-M) implementation.

On the one hand, the NPU offers a much faster runtime than the CPU. Additionally, it fully offloads computation from the CPU, ensuring that other processes' performance remain unaffected. Furthermore, the NPU can be encapsulated within a C++ wrapper, which would enable the creation of an easy-to-use library for seamless integration with other models.

On the other hand, the NPU faces a significant data transfer bottleneck in matrix-vector multiplication scenarios, which renders it considerably slower than AVX-M. Furthermore, the current version of the MLIR-AIE compiler does not support reshape for larger matrices[29], which forces most of the reshape to occur on the host. This host-side reshaping increases the latency and, most notably, the memory demands. The buffer size required can quickly grow, even exceeding the CPU buffer's size.

In addition, NPU development requires a significant learning process. This learning process can become a development burden, especially for non high performance computing programmers. Finally, the NPU only supports the BF16 floating point type, which may introduce numerical error when downcasting from FP64, especially in precision-sensitive applications.

Overall, the NPU is a solid option in cases where BF16 precision is acceptable and an NPU acceleration library for the operation is available. However, there is still some room for improvement. In particular, addressing the data transfer bottlenecks in future NPU hardware versions and enhancing the MLIR-AIE compiler for large-scale data reshaping would significantly broaden the NPU's advantadges.

For economic and, more broadly, scientific models, the NPU could be a viable acceleration platform — provided that it does not require custom kernel or dataflow development. In this sense, if the acceleration task is limited to a simple operation like matrix-matrix or matrix-vector multiplication, then an easy-to-use library could make the NPU integration both practical and efficient. Fortunately, most scientific models heavily rely on matrix-matrix multiplication, making the NPU a promising fit for these workloads. That said, for workloads dominated by matrix-vector operations, an AVX-M implementation would still yield significantly faster performance — at the cost of higher CPU usage.

8.1 Future Work

After the extensive research conducted throughout this project, we believe the findings are interesting enough to merit an article at a workload characterization conference. In specific, due to the lack of existing research studying the computational limits and architectural characteristics of the NPU, this research could provide valuable insight for those interested in NPU development. Specifically, we plan on submitting the paper to *IEEE's International Symposium on Workload Characterization (IISWC)*.

In addition to submitting an article, we believe it would be interesting to explore faster and next-generation versions of the NPU. In specific, it would be interesting to evaluate the performance of NPU2, which is available in newer CPUs and features 7 *ShimDMA* units — a significant increase that could almost double the performance of the current NPU1.

Furthermore, integrating the model with an FPGA platform such as Xilinx's V70 board could be another interesting direction. The V70 features an NPU2 with 12 *ShimDMA* units, along with the added benefit of reprogrammable logic. This could offer opportunities for even greater performance gains.

Another promising research direction could be a comparative study between different commercial NPUs, namely Intel's integrated NPU found in the newer Intel Core Ultra processors. Examining the architectural characteristics, performance profiles, and target workloads of each NPU could reveal whether one offers significant advantages over the other or if each is better suited to different use cases.

Finally, another interesting research venue could be the development of a user-friendly NPU library. Such a library could offer precompiled kernel images optimized for different NPU versions, along with C++ wrappers that simplify integration into scientific models. By taking away the complexity of NPU development and providing an easy-to-use interface, such a library could significantly boost productivity and promote broader adoption of NPUs within the scientific community.

Appendices

A Budget Management

An economic study is necessary to identify and estimate the costs associated with the project. This will provide an estimated budget and help assess the project's economic viability. The costs have been categorized into staffing and generic costs, as well as contingency and risk & roadblock budgets. Also, the management of these costs throughout the project is explained.

A.1 Staffing Costs

In order to determine the staffing costs of the personnel involved in this project, we have to identify their roles and associated salaries. In this case, there are two roles: Assistant Professor, who will advise the developer, and Research Visitor, who will be developing the project. The standard salary for an Assistant Professor has been taken from the official UCB website[30], while the Research Visitor salary has been obtained from the UCB Europe-Colorado Mobility Program's website[31].

In addition to the gross salary, the employer has to pay an extra 6.2% per employee, according to the official Social Security Administration's website[32]. However, Research Visitors are not eligible for Social Security benefits, so there is no extra expense for the employer in this case. Table 18 provides the gross hourly wage, along with the employer's social security contribution.

Role	Gross Salary	Social Security	Expense
Assistant Professor (AP)	\$64.6/h	\$4/h	\$68.6
Research Visitor (RV)	\$7.5/h	\$0/h	\$7.5

Table 18: Staffing Cost
Author's own work

Once the hourly cost per employee has been defined, we can proceed to calculate the cost per activity (CPA). The CPA is calculated based on the Gantt Chart by multiplying the number of hours per task by the hourly wage for each role involved. Table 19 provides a detailed view of the employees involved in each task and its total cost.

Code	Task	Time	Roles	Cost
GP	**Project Management**	**90h**		
GP1	Context & Scope	25h	RV	$187.5
GP2	Time Planning	10h	RV	$75
GP3	Budget & Sustainability	10h	RV	$75
GP4	Thesis Document	25h	RV	$187.5
GP5	Thesis Defense	10h	RV	$75
GP6	Weekly Meetings	10h	RV, AP	$761
PT	**Preliminary Tasks**	**170h**		
PT1	Model Study	40h	RV	$300
PT2	Environment Setup	80h	RV	$600
PT3	MLIR-AIE Study	40h	RV	$300
PT4	State of the Art Analysis	10h	RV, AP	$761
PD	**Product Development**	**230h**		
PD1	Design	20h	RV	$150
PD2	Implementation	130h	RV	$975
PD3	Verification	80h	RV	$600
PA	**Product Assessment**	**80h**	RV	
PA1	Benchmarking & Performance	40h	RV	$300
PA2	Evaluation	40h	RV	$300
Total		**570h**		**$5647**

Table 19: Total Staffing Cost Breakdown
Author’s own work

A.2 Generic Costs

The generic costs (GC) are calculated as overall expenses, since they do not directly depend on individual project tasks but rather on the project as a whole. These include asset depreciation, workplace rent, electricity bill, and internet bill.

A.2.1 Asset Depreciation

Hardware
The hardware costs of the project will only consist of a mid-range computer, with an

approximate cost of $700[4], and two screens, with a cost of $150 each[5]. Assuming a lifespan of 60 months — 5 years — for these 3 devices, the depreciation during the 4 research months will be:
$\frac{4}{60} \times (700 + 2 \times (150)) =$ **$66.7**.

Software
As it has been previously mentioned, all of the software used throughout the project is free to use. Therefore, the software depreciation is $0.

A.2.2 Rent

The project will be developed in the UCB Computer Systems Lab, located in the Engineering Center of UCB's main campus. The average office cost per person in Boulder is $329[6], which adds up to $987 for the 3 people involved. Considering the 4-month research period in Boulder, the total cost adds up to a total of **$3948**.

A.2.3 Electricity Bill

The price of electricity in Boulder is 15¢/kWh[7]. Using this rate, we can estimate the total cost of running the laptop and the monitors. Table 20 provides an overview of the total costs associated with each device. The laptop has a power consumption of 230W[8], while the monitors consume 40W each[9].

Device	Power	Hours	Energy	Cost
HP Victus Laptop	230W	570	131kWh	$19.65
Screen Monitor	40W	570	23kWh	$3.45
Screen Monitor	40W	570	23kWh	$3.45
Total				$26.55

Table 20: Power Consumption by device
Author's own work

A.2.4 Internet Bill

A high-speed internet connection costs $35/mo[10]. Considering 8 hours of research per day over 4 months, the proportional internet cost would be calculated as:
$4mo * \frac{\$35}{mo} * \frac{8h}{24h} =$ **$46.67**.

[4]https://www.hp.com/us-en/shop/pdp/victus-by-hp-gaming-laptop-16t-s100-161-91k72av-1
[5]https://www.bestbuy.com/site/asus-vg248qg-24-widescreen-lcd-elmb-sync-adaptive-sync-snd-freesync-compatible-fhd-gaming-monitor-displayport-hdmi-black/6373927.p
[6]https://liquidspace.com/us/co/boulder
[7]https://www.energysage.com/local-data/electricity-cost/co/boulder-county/
[8]https://www.hp.com/us-en/shop/pdp/victus-by-hp-gaming-laptop-16t-s100-161-91k72av-1
[9]https://files.bbystatic.com/Z1iPp5if3w4FHb7AWmpNzQ%3D%3D/7138DF68-9240-4F7A-9071-BB0C6E6CEB24.pdf
[10]https://www.t-mobile.com/home-internet

A.2.5 Total Generic Costs

Table 21 summarizes the total generic costs associated with the project, as previously detailed.

Device	**Cost**
Asset Depreciation	\$66.7
Rent	\$3948
Electricity Bill	\$26.55
Internet Bill	\$46.67
Total	\$4087.92

Table 21: Generic Costs
Author's own work

A.3 Contingency Budget

As with time planning, potential unexpected expenses must be accounted for. To avoid running short on funding, there has to be a buffer known as the contingency budget. In software development projects, the contingency budget usually represents between a 10% and 20% percent of the CPA and GC. In this case, we have opted for the average of 15%. Below is the contingency budget for the project:

Contingency Budget $= (Total\ CPA + Total\ GC) * 0.15 = (\$5647 + \$4087.92) * 0.15 =$ **\$1460.24**

A.4 Risks & Roadblocks Budget

As outlined in Possible Risks & Roadblocks Section, potential roadblocks may cause delays to the development of the project. These delays can result in increased costs. Below is the quantification of each risk and the corresponding cost increase:

- **Compatibility Issues:**
  - **Hardware Incompatibility:** As previously mentioned, the overhead of a hardware incompatibility would be negligible. While this issue is likely to occur, it is not highly probable, so a probability of around 20% has been assigned.
  - **Software Incompatibility:** The solution to software compatibility issues would take at most 16 hours of development. Therefore, the added cost would be $16h * \frac{\$7.5}{h} =$ **\$120**. Since software incompatibility is quite likely, the assigned probability is 40%.
- **Implementation Errors:**

- **Corner-Case Bugs:** Due to the difficulty of solving corner-case bugs, up to 80 additional development hours may be required. This would add up to $80h * \frac{\$7.5}{h} =$ **$600**. While corner-case bugs are likely to occur, it is unlikely that they will take the full 80 hours of development. Therefore, the assigned probability is 35%.
- **Timing Violations:** Similar to corner-case bugs, the difficulty of optimizing the algorithm could require up to 80 additional development hours. This would add up to $80h * \frac{\$7.5}{h} =$ **$600**. Although this may happen, it is not highly likely that timing violations will be due to an unoptimized algorithm, so the probability would be of 10%.

Event	**Cost**	**Probability**	**Cost**
Hardware Incompatibility	$0	20%	$0
Software Incompatibility	$120	40%	$48
Corner-Case Bugs	$600	35%	$210
Timing Violations	$600	10%	$60
Total			$1320

Table 22: Risks & Roadblocks Budget
Author's own work

A.5 Total Project Costs

Table 23 shows the final cost of the project, which is derived from adding all the aforementioned costs.

Concepte	**Cost**
CPA	$5647
GC	$4087.92
Contingency Budget	$1460.24
Risks & Roadblocks Budget	$1320
Total	$12515.16

Table 23: Total Project Costs
Author's own work

A.6 Management Control

Monthly evaluations of costs will be conducted to monitor the estimated versus actual expenditures, as this timeframe allows for identifying significant discrepancies. The following formulas will be used to assess where the deviations arise from:

- **Hourly Consumption Deviation:**
  (Estimated Hours - Real Hours) × Estimated Cost
- **Cost Deviation by Hourly Consumption:**
  (Estimated Hours - Real Hours) x Real Cost
- **Cost Deviation in Human Resources per Task:**
  (Estimated Cost − Real Cost) x Real Hours
- **Cost Deviation of Assets:**
  Estimated Asset Cost - Real Asset Cost
- **Staffing Costs Deviation:**
  Estimated Staffing Cost - Real Staffing Cost
- **Total Deviation of General Costs:**
  Estimated General Costs - Real General Costs
- **Hourly Deviation:**
  Estimated Hours - Real Hours
- **Cost Deviation:**
  Total Estimated Cost - Total Real Cost

If a noticeable deviation occurs, we will analyze expenditures to determine its source. The formulas provided will help identify where the deviation stems from. Once the root cause is identified, we will assess whether it is linked to one of the previously identified risks. If so, the risks & roadblocks budget will be used to cover the additional cost. If the cost exceeds this budget or arises from an unforeseen event, the contingency budget will be utilized.

In the worst-case scenario — where costs surpass both the contingency and risks & roadblocks budgets — we will need to adjust the project scope and redesign the plan to remain as close to the original budget as possible.

As a preventive measure, we will strictly adhere to the use of the resources described in this document. This will help avoid any unexpected cost. For example, we will limit ourselves to the specific computers mentioned to prevent unexpected power consumption expenses.

B Sustainability Report

B.1 Sustainability Self-Assessment

Throughout my education, from elementary school to university, there has been a strong emphasis on raising awareness about the economic, societal, and environmental impacts of our actions. Alongside these efforts, my life experiences have also helped me develop greater awareness.

Growing up in a typical household, I had to be mindful of conserving water, limiting the use of A/C or heating, and avoiding wasteful spending. Inadvertently, these habits also helped minimize my environmental footprint.

I have been conscious of my environmental and economic sustainability. However, my understanding has been more personal and practical than theoretical. While I understand the basics, I am not familiar with the specific laws, regulations, and measures designed to achieve societal sustainability goals.

Taking the survey[11] made me realize my lack of knowledge about theoretical metrics and regulations. There were questions about specific policies and standards that I was not familiar with. However, when it came to practical questions, I was able to answer positively.

One reason for this gap is that I have not yet conducted any project that required further research or impact measurement. At university, I have learned about sustainability through various lectures, but I have never had the opportunity to apply this knowledge in a technical setting.

Regarding the social aspect, I have always been personally invested in helping those in need or marginalized communities. However, I feel that this focus has been lacking in sustainability discussions at university. In retrospect, there has been a strong emphasis on environmental and economic issues, while the social aspect has been treated as more incidental.

In conclusion, I am generally aware of the environmental, economic, and social impacts of my actions, but I am not entirely familiar with the specifics. I believe that through this bachelor's thesis, as well as with future projects, I will acquire a deeper theoretical understanding of sustainability.

B.2 Environmental Dimension

The development of this project is conducted on a used laptop that was donated by AMD to the UCB Computer Systems Laboratory. Also, the monitors used for this project are part of the UCB Computer Systems Laboratory inventory. Thus, no new material has been acquired in order to conduct it. This prevents further environmental

[11] https://docs.google.com/forms/d/e/1FAIpQLSfVgBxcxZfh7pB_OVRUNGQmRpFDFlhAskukNcpQBowLRF4-sA/viewform

impact of disposing of older hardware, while producing new.

The state of the art for economic model acceleration is not very efficient. While the latest research shows certain improvement, most research is still focused on the old development flows. Therefore, there is a great margin for improvement in this project. As such, one of the project's main goals is to significantly reduce the time and energy consumption, thus having a much better environmental impact.

A more efficient model implementation will certainly lead to higher productivity. This increased productivity allows for the deployment of more models, which in turn results in greater energy consumption. However, the time required for other areas of research, such as studying and developing the model, will offset this rise in energy consumption. Therefore, the implementation of economic models on NPUs will still have a significant positive environmental impact.

B.3 Economic Dimension

The cost of the project has been explained in detail in Section 4 - Budget Management. All costs have been properly broken down and optimized to the fullest extent. Since every element is essential to the development of the project, no cost is expendable. Therefore, we believe the estimated cost is proper and it could not be more optimized.

As previously mentioned, the current state of the art addresses economic model implementation in a highly inefficient manner. This inefficiency leads to both environmental and economic costs. The amount of time required to run an economic model results in a hardware depreciation, while high power consumption adds to overall energy costs.

Moreover, existing solutions come with significant hardware costs. As noted in Section 1.7 - Rationale & Motivation, the CPUs and FPGAs specifically used for this purpose cost $10,000 and $15,000, respectively. In contrast, NPUs are integrated into end-user CPUs, resulting in little to no additional cost.

By eliminating the need for costly high-end hardware, such as workstation CPUs or FPGAs, and by reducing the amount of time and energy consumed by running an economic model, this project represents a major cost-saving improvement over the current state of the art.

B.4 Social Dimension

Personally, I hope that this project will help me transition from my bachelor's studies to a PhD. Up until now, success in coursework has largely depended on following the guidelines set by professors. However, both the bachelor's thesis and future graduate research require me to become an independent learner and researcher. Without structured daily tasks assigned by an instructor, I must learn to autonomously advance toward my research objectives.

As previously mentioned, the mainstream solution requires up to 7 hours to run an economic model on a standard single thread of a CPU. Although researchers can parallelize their work by focusing on other tasks, waiting an entire workday for simulation results significantly slows down progress.

Aside from economic research, we believe that an accessible development framework for generic scientific models can enhance the overall research productivity. Upon a successful project, the results could be extended to other fields, such as biology and environmental sciences, where accelerating model runtime could similarly boost productivity.

This project aims to provide a major quality-of-life improvement by significantly reducing simulation times. Currently, an unsuccessful simulation means a loss of several hours — our goal is to reduce this to a negligible time frame. This acceleration, alongside the environmental and economic improvements, is why we believe there is a real need for this project.

C Gantt Chart Representation

The Gantt Chart available in Figure 18 provides a graphical representation of the previous Table 2. The figure shows the task durations, dependencies, and overall project timeline. In addition, the development of the project has been divided into different *sprints*.

C.1 Sprints

Following the agile methodology, the timeline has been subdivided into different sprints. Sprints are a span of time with a conceptual goal. Typically, they last between 1 and 4 weeks and conclude with a review of all the research that has been conducted, followed by planning for the next sprint. This iterative process allows for adjustments based on progress and timeline deviations. The project is organized into the following sprints:

Sprint 1: The first sprint lasts approximately 2 weeks and focuses on the preliminary tasks. It accommodates setting up the environment and settling into the UCB Computer Systems Laboratory. During this period, the hardware will be configured, the necessary frameworks will be installed, and the MLIR-AIE study will begin.

Sprint 2: This is the longest sprint, comprising the entire month of the *GEP* course. During this period, the preliminary tasks will be conducted alongside the course. These include the MLIR-AIE study, the Model Study, and the State of the Art Analysis.

Sprint 3: This sprint marks the start of actual development. It includes the entire design of the NPU implementation, as well as the initial work on the thesis document.

Sprint 4: This is the second product development sprint, focusing on the dataflow implementation and some preliminary verification.

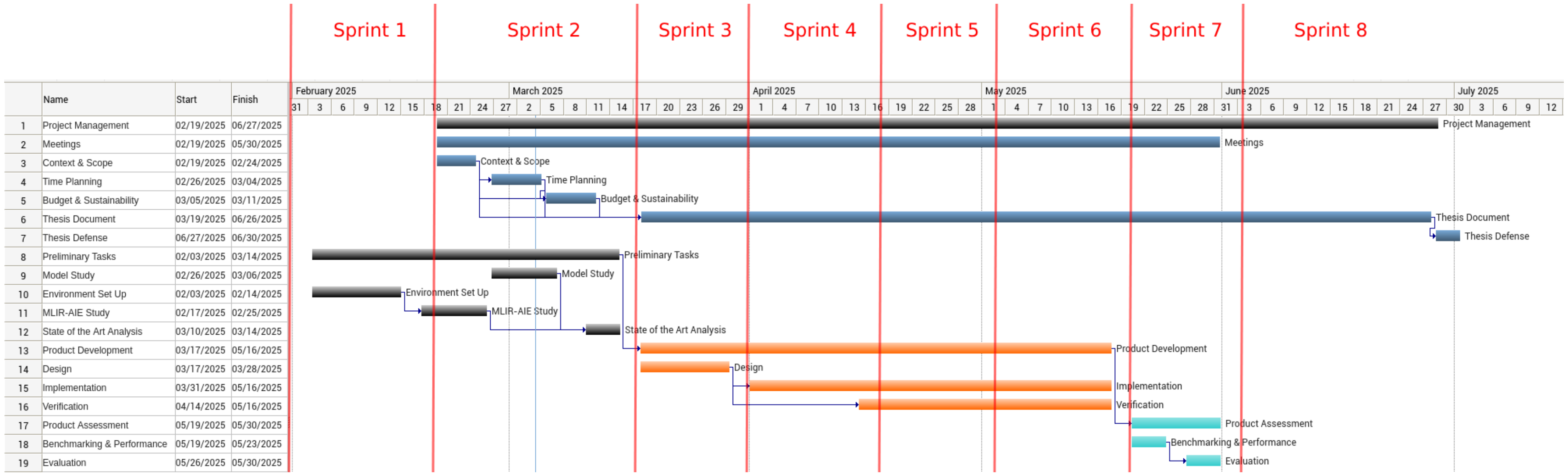
Sprint 5: This is the third and central developmental sprint, where the core of the code will be implemented, focusing on algorithm implementation and optimization. Verification will also be conducted in parallel.

Sprint 6: This is the fourth and final implementation sprint, which entails finishing up and polishing the code, as well as fixing minor bugs. The objective of the sprint is to ensure that the entire model is functional.

Sprint 7: This constitutes the final research sprint, focused on benchmarking the model and obtaining performance and power metrics. The obtained metrics will be evaluated, and conclusions will be drawn. During this sprint, the last research information will be added to the thesis document.

Sprint 8: This final sprint focuses on polishing the documentation details of the project. At this point, the research will be complete, and most information will be on the document. The only work required during this sprint will be preparing the document for submission and preparing the thesis defense.

As shown in Figure 18, the meetings task is a recurring and will only take place throughout the research months. The thesis document task begins after finishing the *GEP* course and will not end until the thesis is finished. The actual preparation of the thesis defense is assigned to the 8th sprint and will take place during the final week before the defense.



	Name	Start	Finish
1	Project Management	02/19/2025	06/27/2025
2	Meetings	02/19/2025	05/30/2025
3	Context & Scope	02/19/2025	02/24/2025
4	Time Planning	02/26/2025	03/04/2025
5	Budget & Sustainability	03/05/2025	03/11/2025
6	Thesis Document	03/19/2025	06/26/2025
7	Thesis Defense	06/27/2025	06/30/2025
8	Preliminary Tasks	02/03/2025	03/14/2025
9	Model Study	02/26/2025	03/06/2025
10	Environment Set Up	02/03/2025	02/14/2025
11	MLIR-AIE Study	02/17/2025	02/25/2025
12	State of the Art Analysis	03/10/2025	03/14/2025
13	Product Development	03/17/2025	05/16/2025
14	Design	03/17/2025	03/28/2025
15	Implementation	03/31/2025	05/16/2025
16	Verification	04/14/2025	05/16/2025
17	Product Assessment	05/19/2025	05/30/2025
18	Benchmarking & Performance	05/19/2025	05/23/2025
19	Evaluation	05/26/2025	05/30/2025

Figure 18: Gantt Chart
Author’s own work with the Gantter tool

D Extended Matrix-Vector Implementation

Since the matrix-matrix implementation requires matrix B to have a minimum of four columns, an alternative approach is to extend the matrix-vector implementation for $N < 4$. This is achieved by performing multiple matrix-vector multiplications, which effectively construct the result column by column.

While this extended matrix-vector method works properly, its computation time scales linearly with the number of columns in matrix B and is less efficient than the dedicated matrix-matrix implementation. In addition, this implementation requires more complex reshape than matrix-vector, both in the host and in the NPU.

This implementation results in the following matrix reshapes on the host:

Matrix A:

$$P_7 = (n\_aie\_columns,\ \tfrac{rows_A \times columns_A}{n\_aie\_columns})$$
$$P_6 = (\tfrac{columns_B}{n},\ 0)$$
$$P_5 = (\tfrac{rows_A}{m \times n\_cores},\ m \times n\_aie\_rows \times columns_A)$$
$$P_4 = (\tfrac{columns_A}{k},\ k)$$
$$P_3 = (n\_aie\_columns,\ m \times columns_A)$$
$$P_2 = (k,\ 1)$$
$$P_1 = (m,\ columns_A)$$

Matrix B:

$$P_5 = (\tfrac{columns_B}{n},\ n)$$
$$P_4 = (\tfrac{rows_A}{m \times n_{cores}},\ \tfrac{rows_A \times columns_B}{n\_aie\_columns})$$
$$P_3 = (\tfrac{rows_B}{K},\ k \times columns_B)$$
$$P_2 = (n,\ 1)$$
$$P_1 = (k,\ columns_B)$$

Output Matrix:

$$P_4 = (n\_aie\_columns,\ \tfrac{rows_A \times columns_B}{n\_aie\_columns})$$
$$P_3 = (\tfrac{rows_A}{n\_aie\_columns},\ n)$$
$$P_2 = (\tfrac{columns_B}{n},\ \tfrac{rows_A}{n\_aie\_columns} \times n)$$
$$P_1 = (n,\ 1)$$

Performing matrix-matrix multiplication column by column results in the output subtile being generated in column major format. Therefore, we transpose the result as the subtile exits the compute core and heads to the MemTile. Given subtiles $m \times k$ and $k \times n$, the following are the reshape dimensions on the NPU:

Output Subtile:

$$P_2 = (m,\ 1)$$
$$P_1 = (n,\ m)$$

E Source Code

This section of the appendix contains the most important excerpts of code. For access to the full code, please review the attached zip folder.

E.1 loop_uhat function

E.1.1 Original CPU Implementation

```
void loop_uhat(const real tolerance, const real *trmult_reduced,
 ↪  const real *L, const real L_exp, real *uhat_i, const real
 ↪  *R,const real R_exp)
/*
    input:
    - tol: level of tolerance, scalar, real
    - trmult_reduced: 17048x17048, real
    - L,R: 17048x1, real
    - L_exp,R_exp: scalar
    output:
    - uhat_i: 17048x1, real
 */
{
    real integral[NPOSLAND];
    real rhs[NPOSLAND];

    real error = tolerance;
    int iter = 0;
    while (error >= tolerance)
    {
        error = 0;

        // Compute the iLOOP_UHATnput vector.
        for (int i = 0; i < NPOSLAND; i++)
            integral[i] = R[i] * pow(uhat_i[i], R_exp);

        // Perform the Matrix-Vector multiplication. This will be
         ↪  accelerated by the NPU.
        for (int i = 0; i < NPOSLAND; i++)
        {
            rhs[i] = 0;
            for (int j = 0; j < NPOSLAND; j++)
                rhs[i] += trmult_reduced[i * NPOSLAND + j] *
                 ↪  integral[j];
        }
        // Compute the relative error and update the guess.
        for (int i = 0; i < NPOSLAND; i++)
        {
            real uhat_f = L[i] * pow(rhs[i], L_exp);
            real deviation = (uhat_i[i] - uhat_f) / uhat_i[i];
            error += deviation * deviation;
            // Update the Guess
            uhat_i[i] = uhat_f;
        }
        iter = iter + 1;
    } // end while
}
```

E.1.2 Multithreaded + AVX CPU Implementation

```
void loop_uhat(const real tolerance, const real * trmult_reduced,
 ↪  MatMatAVX * mat_avx,
                const real *L, const real L_exp, real *uhat_i,const
                 ↪  real *R, const real R_exp)
/*
    input:
    - tol: level of tolerance, scalar, real
    - trmult_reduced: 17048x17048, real
    - L,R: 17048x1, real
    - L_exp,R_exp: scalar
    output:
    - uhat_i: 17048x1, real
 */
{
    real integral[NPOSLAND];
    std::bfloat16_t integral_AVX[NPOSLAND];
    float rhs_avx[NPOSLAND];
    real rhs[NPOSLAND];

    static constexpr bool verify = false;

    real error = tolerance;
    int iter = 0;
    while (error >= tolerance)
    {
        error = 0;

        // Compute the input vector.
        for (int i = 0; i < NPOSLAND; i++)
            integral[i] = R[i] * pow(uhat_i[i], R_exp);

        auto [s, z] =
         ↪  GetScalingParameters<std::bfloat16_t>(integral,
         ↪  NPOSLAND);

        for (int i = 0; i < NPOSLAND; i++)
            integral_AVX[i] =
             ↪  static_cast<std::bfloat16_t>(integral[i] * s + z);

        // Load matrix B into the object.
        mat_avx->LoadMatrixB(integral_AVX);
        // Perform the Matrix-Vector multiplication. This is
         ↪  accelerated with AVX and multithreading.
        mat_avx->Compute(rhs_avx);

        // Scale the result back.
        for (int i = 0; i < NPOSLAND; i++)
            rhs[i] = (static_cast<real>(rhs_avx[i]) - z) / s;

        // Verify the result if the verify parameter is set to
         ↪  true.
        if (verify)
        {
            for (int i = 0; i < NPOSLAND; i++)
            {
                real expected = 0;
                for (int j = 0; j < NPOSLAND; j++)
                    expected += trmult_reduced[i * NPOSLAND + j] *
                     ↪  integral[j];

                if (!double_equal(expected, rhs[i]))
                {
                    std::cout << "(" << expected << ", " << rhs[i]
                     ↪  << ")" << std::endl;
                    throw std::runtime_error("Incorrect result");
                }
            }
        }

        // Compute the relative error and update the guess.
        for (int i = 0; i < NPOSLAND; i++)
        {
            real uhat_f = L[i] * pow(rhs[i], L_exp);
            real deviation = (uhat_i[i] - uhat_f) / uhat_i[i];
            error = std::max(error, abs(deviation));
            // Update the Guess
            uhat_i[i] = uhat_f;
        }

        iter = iter + 1;
    } // end while
}
```

E.1.3 NPU Implementation

```
void loop_uhat(const real tolerance, const real *trmult_reduced,
 ↪  const real *L, const real L_exp, real *uhat_i, const real
 ↪  *R,const real R_exp)
/*
    input:
    - tol: level of tolerance, scalar, real
    - trmult_reduced: 17048x17048, real
    - L,R: 17048x1, real
    - L_exp,R_exp: scalar
    output:
    - uhat_i: 17048x1, real
 */
{
    // Is verification enabled?
    static constexpr bool verify = true;

    // Initialization will be run once.
    static bool initialized = false;

    // Reference values used to generate a random initial guess.
    static real reference_uhat_i[NPOSLAND];

    // If the reference values have not been initialized yet, run
     ↪  the original CPU implementation and
    // store the result uhat_i as the reference values.
    if (!initialized)
    {
        loop_uhat_cpu(tolerance, trmult_reduced, L, L_exp, uhat_i,
         ↪  R, R_exp);
        memcpy(reference_uhat_i, uhat_i, NPOSLAND * sizeof(real));

        initialized = true;
        return;
    }
```

```
// Allocate memory for the different values used. Each guess
↪  will have its own set of values.
real * integral = static_cast<real*>(calloc(N_GUESSES *
↪  NPOSLAND, sizeof(real)));
std::bfloat16_t * integral_NPU =
↪  static_cast<std::bfloat16_t*>(calloc(NPOSLAND * N_GUESSES,
↪  sizeof(real)));
float * rhs_npu = static_cast<float*>(calloc(NPOSLAND *
↪  N_GUESSES, sizeof(real)));
real * uhat_i_parallel = static_cast<real*>(calloc(N_GUESSES *
↪  NPOSLAND, sizeof(real)));
real * rhs = static_cast<real*>(calloc(NPOSLAND * N_GUESSES,
↪  sizeof(real)));

// Define a scale and offset for each guess.
double scale[N_GUESSES], offset[N_GUESSES];

// Initialize the values for each guess randomizing between 0
↪  and the reference uhat_i values.
// The first half of the guesses will be capped at the value,
↪  while the second half will be capped at twice the value.
for (int i = 0; i < NPOSLAND; ++i)
{
    const double current_value = reference_uhat_i[i];

    for (int j = 0; j < N_GUESSES; ++j)
    {
        if (j < N_GUESSES / 2)
            uhat_i_parallel[j * NPOSLAND + i] = double_rand(0,
            ↪  current_value);
        else //if (j >= N_GUESSES / 2)
            uhat_i_parallel[j * NPOSLAND + i] = double_rand(0, 2
            ↪  * current_value);
    }
}
```

```
// Iterate until the fixed point is found.
real error = tolerance;
int best_guess = 0;
int iter = 0;
while (error >= tolerance)
{
    error = std::numeric_limits<real>::max();

    // Compute the input vector for each guess.
    for (int j = 0; j < N_GUESSES; ++j)
    {
        for (int i = 0; i < NPOSLAND; i++)
            integral[j * NPOSLAND + i] = R[i] *
             ↪  pow(uhat_i_parallel[j * NPOSLAND + i], R_exp);

        auto [s, o] =
         ↪  GetScalingParameters<std::bfloat16_t>(&integral[j *
         ↪  NPOSLAND], NPOSLAND);
        scale[j] = s;
        offset[j] = o;

        for (int i = 0; i < NPOSLAND; i++)
            integral_NPU[i * N_GUESSES + j] = integral[j *
             ↪  NPOSLAND + i] * scale[j] + offset[j];
    }

    // Load matrix B into the object.
    npu->LoadMatrixB(integral_NPU);

    // Perform the Matrix-Vector multiplication. This is
     ↪  accelerated with AVX and multithreading.
    npu->Compute(rhs_npu);

    // Scale the result back for each guess.
    for (int j = 0; j < N_GUESSES; ++j)
    {
        for (int i = 0; i < NPOSLAND; i++)
            rhs[i * N_GUESSES + j] =
             ↪  (static_cast<real>(rhs_npu[i * N_GUESSES + j]) -
             ↪  offset[j]) / scale[j];
    }
    // Verify the result if the verify parameter is set to
    ↪  true.
    if (verify)
    {
        for (int j = 0; j < N_GUESSES; ++j)
        {
            for (int i = 0; i < NPOSLAND; i++)
            {
                real expected = 0;
                for (int k = 0; k < NPOSLAND; k++)
                    expected += trmult_reduced[i * NPOSLAND + k]
                    ↪  * integral[j * NPOSLAND + k];

                if (!double_equal(expected, rhs[i * N_GUESSES +
                ↪  j]))
                {
                    std::cout << std::endl << "Incorrect result
                    ↪  [" << j << ", " << i <<"]: (" <<
                    ↪  expected << ", " << rhs[i * N_GUESSES +
                    ↪  j] << ")" << std::endl;
                    exit(1);
                    throw std::runtime_error("");
                }
            }
        }
    }
```

```
        // Update the guess for each of the guesses.
        for (int j = 0; j < N_GUESSES; ++j)
        {
            double guess_error = 0;

            // Compute the relative error and update the guess.
            for (int i = 0; i < NPOSLAND; i++)
            {
                real uhat_f = L[i] * pow(rhs[i * N_GUESSES + j],
                 ↪  L_exp);
                real deviation = (uhat_i_parallel[j * NPOSLAND + i]
                 ↪  - uhat_f) / uhat_i_parallel[j * NPOSLAND + i];
                guess_error = std::max(guess_error, abs(deviation));

                // Update the Guess
                uhat_i_parallel[j * NPOSLAND + i] = uhat_f;
            }

            // Save the informaiton on the guess with the lowest
             ↪  error.
            if (guess_error < error)
            {
                best_guess = j;
                error = guess_error;
            }
        }

        iter = iter + 1;
    } // end while

    // Copy the best guess into the output.
    memcpy(uhat_i, &uhat_i_parallel[best_guess * NPOSLAND], NPOSLAND
     ↪  * sizeof(real));

    free(integral);
    free(integral_NPU);
    free(rhs_npu);
    free(uhat_i_parallel);
    free(rhs);
}
```

References

[1] CDW. *CPU vs. GPU: What's the Difference?* www.cdw.com, Dec. 2022. URL: https://www.cdw.com/content/cdw/en/articles/hardware/cpu-vs-gpu.html (visited on 02/26/2025).

[2] Pure Storage. *CPU vs. GPU for Machine Learning*. Pure Storage Blog, Sept. 2022. URL: https://blog.purestorage.com/purely-educational/cpu-vs-gpu-for-machine-learning/ (visited on 02/26/2025).

[3] IBM, Josh Schneider, and Ian Smalley. *NPU vs. GPU*. Ibm.com, Oct. 2024. URL: https://www.ibm.com/think/topics/npu-vs-gpu (visited on 02/26/2025).

[4] UST Production Engineering Team. *Limitless Future of Custom ASIC Design*. Ust.com, 2025. URL: https://www.ust.com/en/insights/the-limitless-future-of-custom-asic-design (visited on 02/23/2025).

[5] Bhagath Cheela et al. "Programming FPGAs for economics: An introduction to electrical engineering economics." In: *Quantitative Economics* 16 (2025), pp. 49–87. DOI: 10.3982/qe2344. (Visited on 02/26/2025).

[6] Per Krusell and Anthony Smith. "Income and Wealth Heterogeneity in the Macroeconomy." In: *Journal of Political Economy* 106 (1998). URL: http://www.econ.yale.edu/smith/250034.pdf (visited on 02/27/2025).

[7] José-Luis Cruz and Esteban Rossi-Hansberg. *The Economic Geography of Global Warming*. Working Paper 28466. National Bureau of Economic Research, Feb. 2021. DOI: 10.3386/w28466. URL: http://www.nber.org/papers/w28466 (visited on 05/08/2025).

[8] Wikipedia Contributors. *Integrated assessment modelling*. Wikipedia, Oct. 2019. URL: https://en.wikipedia.org/wiki/Integrated_assessment_modelling (visited on 05/08/2025).

[9] Vani Rajasekar, J. Premalatha, and Rajesh Kumar Dhanaraj. "Security analytics." In: *System Assurances* (2022), pp. 333–354. DOI: 10.1016/b978-0-323-90240-3.00019-9. (Visited on 05/08/2025).

[10] Corsair Gaming. *CPU vs GPU vs NPU: What's the difference?* Corsair.com, Aug. 2024. URL: https://www.corsair.com/us/en/explorer/diy-builder/power-supply-units/cpu-vs-gpu-vs-npu-whats-the-difference/?srsltid=AfmBOopdRNHhHmzo_q6S7kRpYtUrwNG7ipgCBHx007OtuYgDwiBwaBxg (visited on 02/27/2025).

[11] IBM. *Neural processing unit*. Ibm.com, Sept. 2024. URL: https://www.ibm.com/think/topics/neural-processing-unit (visited on 02/26/2025).

[12] Wikipedia Contributors. *AMD XDNA*. Wikipedia, Aug. 2024. (Visited on 02/26/2025).

[13] Advanced Micro Devices. *AMD XDNA™ Architecture*. AMD, Feb. 2025. URL: https://www.amd.com/en/technologies/xdna.html (visited on 02/26/2025).

[14] Xilinx. *MLIR-AIE Programming Guide*. GitHub, 2021. URL: https://github.com/Xilinx/mlir-aie/tree/main/programming_guide (visited on 02/26/2025).

[15] Oleksandr Zinenko et al. "MLIR: A Compiler Infrastructure for the End of Moore's Law." In: (Feb. 2020). (Visited on 02/26/2025).

[16] Wikipedia Contributors. *C++*. Wikipedia, Mar. 2019. URL: `https://en.wikipedia.org/wiki/C%2B%2B` (visited on 02/26/2025).

[17] Wikipedia Contributors. *Python (programming language)*. Wikipedia, May 2019. URL: `https://en.wikipedia.org/wiki/Python_(programming_language)` (visited on 02/26/2025).

[18] Victor Fonseca Duarte et al. *Modern Frameworks for Quantitative Economics*. English (US). WorkingPaper. Illinois Experts, June 2019. (Visited on 02/26/2025).

[19] Alessandro Peri. "A hardware approach to value function iteration." In: *Journal of Economic Dynamics and Control* 114 (May 2020), p. 103894. DOI: `10.1016/j.jedc.2020.103894`. (Visited on 02/27/2025).

[20] Sarah Laoyan. *What is agile methodology? (A beginner's guide)*. Asana, Feb. 2025. URL: `https://asana.com/resources/agile-methodology` (visited on 02/26/2025).

[21] Xilinx. *Device support*. Xilinx AIEngine MLIR Dialect, 2024. URL: `https://xilinx.github.io/mlir-aie/Devices.html` (visited on 05/09/2025).

[22] Wikipedia Contributors. *Banach fixed-point theorem*. Wikipedia, July 2021. URL: `https://en.wikipedia.org/wiki/Banach_fixed-point_theorem` (visited on 05/10/2025).

[23] Wikipedia Contributors. *Machine epsilon*. Wikipedia, Oct. 2022. URL: `https://en.wikipedia.org/wiki/Machine_epsilon` (visited on 05/10/2025).

[24] Xilinx. *AI Engine API User Guide: Matrix Multiplication*. Xilinx.com, 2023. URL: `https://www.xilinx.com/htmldocs/xilinx2023_2/aiengine_api/aie_api/doc/group__group__mmul.html` (visited on 05/10/2025).

[25] Xilinx. *AMD Technical Information Portal*. Amd.com, Nov. 2024. URL: `https://docs.amd.com/r/en-US/ug1603-ai-engine-ml-kernel-graph/accumulate` (visited on 05/12/2025).

[26] Linux Kernel. *AMD NPU — The Linux Kernel documentation*. Kernel.org, 2024. URL: `https://docs.kernel.org/accel/amdxdna/amdnpu.html` (visited on 05/10/2025).

[27] AMD. *AMD Technical Information Portal*. Amd.com, May 2024. URL: `https://docs.amd.com/r/en-US/am020-versal-aie-ml/AIE-ML-Tile-Architecture` (visited on 05/12/2025).

[28] André Rösti. *Storing Matrices in Memory — Data Layout Visualizations*. Andreroesti.com, 2025. URL: `https://andreroesti.com/data-layout-viz/data_layout.html` (visited on 05/21/2025).

[29] Xilinx and Joseph Melber. *NPU enters corrupt state with large amount of SHIM data transfers*. GitHub, Apr. 2025. URL: `https://github.com/Xilinx/mlir-aie/issues/2224#issuecomment-2831014349` (visited on 05/2025).

[30] University of Colorado Boulder. *CU Salary Database*. University of Colorado, Dec. 2024. URL: `https://www.cu.edu/budget/cu-salary-database` (visited on 03/05/2025).

[31] University of Colorado Boulder. *Europe-Colorado Program*. International Programs, 2016. URL: `https://www.colorado.edu/engineering-international/europe-colorado-program#accordion-249196496-1` (visited on 03/05/2025).

[32] U.S. Social Security Administration. *How is Social Security financed? | Press Office | Social Security Administration*. Ssa.gov, 2018. URL: `https://www.ssa.gov/news/press/factsheets/HowAreSocialSecurity.htm` (visited on 03/05/2025).