CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Vehicle Motion Control on SIMD: Traditional and AI based models on the edge

Master's thesis in Embedded Electronic System Design

MADHU SURESH
SAURUBH SUDARSHAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Vehicle Motion Control on SIMD: Traditional and AI based models on the edge

MADHU SURESH
SAURUBH SUDARSHAN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Vehicle Motion Control on SIMD: Traditional and AI based models on the edge
MADHU SURESH, SAURUBH SUDARSHAN

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Company advisor: Henok Fessehatsion, Thyagaraja Naidu, CEVT AB
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

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

Typeset in LaTeX
Gothenburg, Sweden 2022

Vehicle Motion Control on SIMD: Traditional and AI based models on the edge
MADHU SURESH
SAURUBH SUDARSHAN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Recent advancements in technology such as Artificial Intelligence (AI) and Non-Linear Model Predictive Control (NMPC) have led to its use in the field of motion control in vehicles. When it comes to the implementation of the models related to these technologies, they are expected to be executed within hard timing deadlines as they are performance critical. Further, due to their high computational cost, coupled with the strict deadlines, they are usually deployed on accelerators like the Graphics Processing Unit (GPU). However, resource-constrained embedded platforms cannot afford to have such accelerators. Therefore considering these limitations, it's crucial to thoroughly investigate the implementation of these models entirely on CPU without any dedicated accelerator, while meeting the strict requirements. This thesis investigates the method by analyzing two different models, viz. AI and NMPC models, in which the Single Instruction Multiple Data (SIMD) component of an Arm processor is exploited. The SIMD units are commonly used for vector operations in a modern CPU. By using these models, various Arm's SIMD implementation techniques such as Arm Neon intrinsics, Ne10 library and Auto-vectorization are investigated. When compared to the traditional approach of sequential computing implementation, the proposed method implemented with Neon Intrinsics was found to be more efficient and gave an execution time reduction of 61.9% for an AI model, while the NMPC model gave an increase in execution time of 8.3%.

Keywords: AI, NMPC, Graphical Processing Unit, Internet of Things, SIMD, Neon Intrinsics, Neon enabled library, CUDA.

Acknowledgements

We would like to express our sincere gratitude to our supervisors Henok Fessehatsion, Thyagaraja Naidu, and technical expert Stefan Carlsson at CEVT for allowing us to do our Master's thesis and for their encouragement and willingness to assist in any situation. We would also like to thank our supervisor Pedro Petersen Moura Trancoso at Chalmers University who has guided us on the right path throughout the project. Finally, we would like to thank our respective families for the continuous support and encouraging words that have helped us in finishing this thesis.

Madhu Suresh, Saurubh Sudarshan, Gothenburg, December 2022

Contents

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis:

AI	Artificial Intelligence
MPC	Model Predictive Control
GPU	Graphics Processing Unit
TPU	Tensor Processing Unit
SIMD	Single Instruction, Multiple Data
GCC	GNU Compiler Collection
CNN	Convolution Neural Network
FPGA	Field Programmable Gate Array
ML	Machine Learning
DSP	Digital Signal Processing
SLP	Superword-Level Parallelism
PERF	performance analyzing tool in Linux
SISD	Single Instruction, Single Data
MIMD	Muliple Input, Multiple Data
MISD	Muliple Input, Single Data
LLVM	Low Level Virtual Machine
NMPC	Non-linear Model Predictive Control
DP	Dynamic Programming
SOC	State Of Charge
IPOPT	Interior Point OPTimizer
FNN	Feedforward Neural Network
RMSE	Root Mean Squared Error
ReLU	Rectified Linear Unit
CUDA	Compute Unified Device Architecture
SSE	Streaming SIMD Extensions

1

Introduction

The recent advancements in automobiles and information technology have transformed conventional vehicles into smart commuting machines such as hybrid vehicles and self-driving vehicles. These features and characteristics are offered by both cutting-edge communication and computing technologies which pose many challenges for the design of autonomous driving edge computing systems. Major technologies such as AI (Artificial Intelligence) and MPC (Model Predictive Control) are currently being used for improving the functioning and responsiveness of these vehicles [1] [2]. These applications are often complex and resource-intensive, hence they were generally facilitated on a cloud platform [3]. In general, it is more advantageous or required to have these inferences close to the source of data or action requests [4], avoiding the need to send the data to a cloud service and wait for a response. In many scenarios, data transmission to the cloud is unreliable, if not impossible, or has a high latency with uncertainty about the communication's round-trip delay, which is unacceptable for latency-sensitive applications that require real-time decisions. Other considerations, such as data security and privacy, force data to remain on edge devices.

However, existing edge devices pose challenges in themselves, as some of them are not capable of parallel computation. There is a need to opt for multi-core, parallel computing edge devices [5]. Lately, to provide the edge devices with the necessary computation capability to handle complex algorithms and AI, unconventional approaches such as tensor processing unit (TPU) by Google and new graphics processing unit (GPU) architectures by Nvidia are being developed [6]. These devices have multi-core processors which can process many pieces of data simultaneously, making them useful for complex algorithms. Embedded platforms with limited resources, such as Internet of Things (IoT) devices, cannot afford such accelerators as they operate with batteries with limited capability. For specific applications such as vehicle motion control in the automotive sector, Arm processors are widely adopted as control units, and hence shifting to other hardware inflicts inconvenience. Therefore it is important to investigate an efficient method for implementation of AI or similar complex functions on an Arm processor by exploiting resources such as Arm Neon gaining an advantage of parallel computation with SIMD support.

SIMD is one architecture that was used as a base for early supercomputers [7] and that is capable of parallel computation. Recent developments in technology have led SIMD to become a generic feature in high-end processors. Many mobile appli-

cations such as multimedia, graphics, and signal processing have appeared which require better performance, and output quality needs parallel computation capabilities which SIMD has. It also achieves a performance speed-up that is needed for high computational load applications such as computer vision, digital signal processing and other applications that require performing the very same operation over large amounts of data which can achieve significant performance benefits from the use of SIMD extensions. Many commonly used microprocessors have recently improved their architectures to allow specific SIMD extensions, such as Neon for Arm microprocessors [8], AVX for Intel [9] and 3DNow! for AMD microprocessors [10] respectively. These add-ons are available in specific co-processors that enable vectorization.

1.1 Problem Statement

Complex algorithms like the MPC model and AI model are two of the major technologies which are used to improve the functionality of the vehicle. These algorithms are computationally expensive and resource intensive. As a result, their performance must be improved for them to meet the system's stringent deadlines. Hence, edge devices with multi-core and vectorized computing capabilities would be advantageous for these applications. GCC compiler offers the auto-vectorization feature, but it does not always give optimal results. Thus, it is necessary to manually implement a vectorized operation. Arm compatible edge devices such as Raspberry Pi zero 2W has the necessary capabilities such as Arm Neon registers to vectorize the operations. This thesis investigates Arm Neon's capabilities on these models by manually vectorizing the operations by using Neon Intrinsics, Ne10 library, and Auto vectorization on Raspberry pi zero 2W. The findings will be used to answer the following research questions:

- How well can both AI and MPC models be implemented on Arm processor using Neon technology?

1.2 Related Work

Recently, specifically in the autonomous driving field, extensive research is being carried out to deploy AI and similar complex algorithms efficiently on edge devices due to their challenges and concerns of centralized cloud computing. Here are a few articles that help in understanding the issue. To start with, Lee et al. [11] proposed a method to accelerate Convolution Neural Network (CNN) with LeNet network by using SIMD architecture. Here, performance was evaluated using Raspberry Pi 3 MODEL B by utilizing Arm Neon, the SIMD processing unit inside Arm CPUs. In comparison to the traditional implementation, the proposed implementation achieved a speed-up of up to 2.66 in execution time and a 3.55-fold decrease in energy consumption. Similarly, a study on the implementation of six different digital signal processing (DSP) algorithms on an A15 architecture used SIMD to optimize performance [12]. These implementations were then compared to those that are automatically produced by the compiler's auto-vectorization feature. The execution

times of the SIMD implementations achieved much lower execution time compared to that produced by the compiler and the speed-up ranged from 2.47 to 5.11. In another research project, to address the problem of slower inference on edge devices, researchers have focused on accelerating edge inference by both hardware and software means. An FPGA System-on-Chip based architecture to speed up the ML computations on an edge environment was proposed in [13]. In [14], Gaurav Mitra et al. considered and compared the different hardware with SIMD architecture, namely the Neon SIMD instruction set used on the Arm Cortex-A series of RISC processors with the SSE2 SIMD instruction set found on Intel platforms within the context of the Open Computer Vision (OpenCV) library. The performance of compiler auto-vectorization was compared to that of hand-tuned script across five different benchmarks and ten different hardware platforms. Hand-tuned Neon benchmarks on Arm platforms were 1.05x to 13.88x faster than auto-vectorized code, while hand-tuned SSE benchmarks on Intel platforms were 1.34x to 5.54x faster. Also, Liu et al. [15] proposed a computer architecture for a self-driving vehicle that is based on heterogeneous hardware. The authors identified bottlenecks in autonomous driving, which were found to be localization and perception, and matched them with suitable accelerators such as CPU, GPU, and DSP, which enabled them to obtain high performance and energy efficient results. Hence, these papers suggest that SIMD has some advantages in that has simple, repetitive arithmetic operations of enormous amounts of data.

1.3 Aim and Objectives:

The main research question that is addressed in this thesis work is to analyze how capable the Arm Neon is, compared to other hardware, when employed for edge computing of vehicle motion control models. Based on this overall aim, the following objectives are formulated:

- Identify the best implementation method by deploying the existing MPC, i.e., the non-linear MPC (NMPC) [16], on the selected Arm processor by making use of its SIMD component. Evaluate the performance, considering computation time and amount of memory used.
- Similarly, investigate and implement an AI model on Arm Neon using SIMD architecture and compare it with the traditional approach of sequentially computing using only CPU. This will enable us to recommend the best way for implementation on Arm devices with Neon technology. The model will be further implemented on other hardware such as Nvidia GPU for comparison.

1.3.1 Limitation:

Since the main focus of the thesis is to optimize the models using SIMD architecture, a limited machine-independent optimization is done and the execution time measurement which is necessary for the benchmark is constrained to use two main techniques which are PERF [17] and Chronos [18]. The individual model also has its limitations and complications to be used in the thesis for SIMD optimization.

Limitation of NMPC model:

In the case of the NMPC model, it is limited to optimizing evaluation functions which are used to calculate cost functions, that can be vectorized, the rest are not chosen due to limited time. Since these evaluation functions take double precision to store their value, the Neon enabled library which is developed for single precision cannot be used. Hence this limits the model to an investigation with Arm Neon intrinsics.

1.3.2 Thesis Outline:

For ease of understanding, the thesis report is divided into following chapters:

2. **Theory:** This chapter starts with the necessary background literature. Further, it aims to provide the reader a technical insight on important topics such as SIMD, Arm Neon technology and performance metrics used in the thesis.
3. **Implementation models:** This describes the models which are used for implementation in the thesis.
4. **Methods** This section describes the research methodology of all the implementation models, and how the implementation was developed and tested in the thesis.
5. **Results:** This chapter presents and discusses the project results and concludes with a possible solution to the benchmark.
6. **Conclusion:** This chapter concludes with the limitations of the thesis.
7. **Future work:** This describes how the project can be expanded in the future

2

Theory

This section aims to provide technical insight on topics that are salient to thesis, which include: SIMD, Arm Neon technology and its usage, and performance metrics used in the thesis.

2.1 Flynn's Classification

Based on the number of instructions and data streams that can be processed concurrently, in 1966 Michael J Flynn, in his highly cited paper [19], classified computing systems into following four major categories:

1. SISD: A single-instruction, single-data-stream computing system (SISD) is a single processor machine that executes single instruction to operate on a single data stream.
2. SIMD: A single-instruction, multiple-data-stream system is a multiprocessor machine that performs the same operation on multiple pieces of data concurrently.
3. MISD: Out of this four categories, multiple-instruction, single-data-stream system is a rarely used class. But as we approach the limits of Moore's law, few researchers are proposing models based on this parallelism architecture [20].
4. MIMD: Multiple-instruction, multiple-data-stream system involves multiple processors independently executing several instructions on several data sources.

As the main focus of the thesis is to investigate on SIMD architecture in Arm processor, SIMD is emphasized in subsequent discussions.

SIMD

Machines executing vectors of data have been present for a while now. Early supercomputers such as the CDC Star-100 and the Texas Instruments ASC could operate on the vector of data with a single instruction [21]. SIMD machines can only exploit data level parallelism but not concurrency. The following figure 2.2 helps to understand the difference between the conventional scalar operation and SIMD operation. Using scalar operations, to acquire the sums, four add instructions have to be executed sequentially. SIMD operation, on the other hand, achieves the same outcome with only one additional instruction.

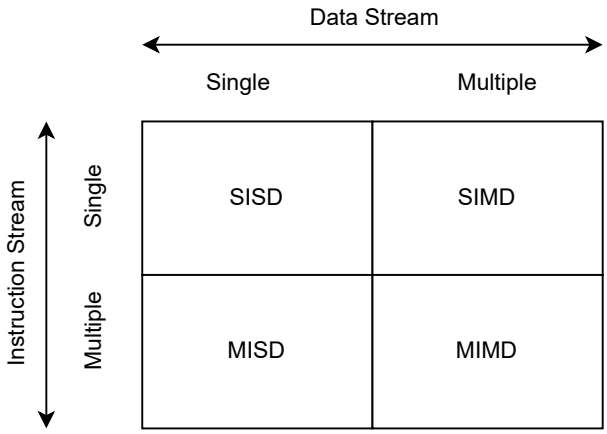

Figure 2.1: Flynn's Classification

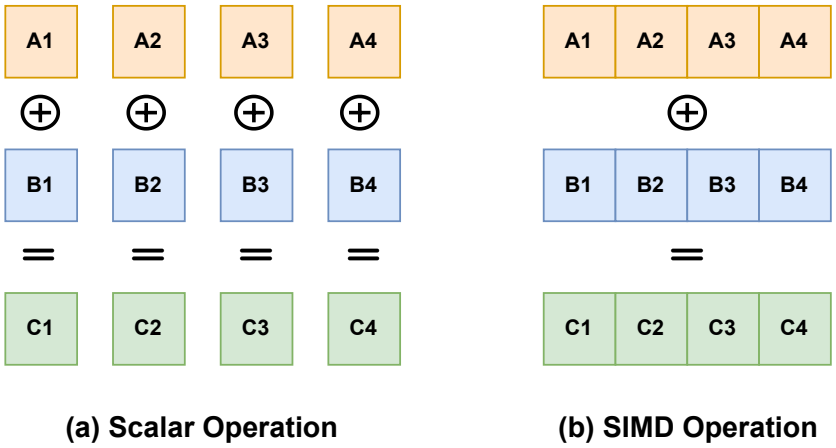

Figure 2.2: Scalar and SIMD operations illustration

It is also important to note in figure 2.2 that each rectangle represents registers. A regular register, as illustrated in figure 2.2, can accommodate only a single scalar value and aid in the fast retrieval of data for processing by the CPU. However, in the case of architecture employing SIMD capabilities, they contain a separate set of wide registers generally termed SIMD or vector registers. These registers are capable of containing multiple lanes which helps in storing multiple scalar values.

Depending on the architecture the width of the SIMD registers vary. If we assume the width of the SIMD register to be 64 bits, it would allow the user to operate with either eight 8-bit elements, sixteen 4-bit elements or two 32-bit elements.

Likewise, from the example provided in figure 2.2 b), if we assume the width of SIMD register to be 128 bits, then we are adding four 32-bit elements with another register having similar structure. The corresponding elements in the lanes of the two registers would be added and the result would be placed on the respective lane of the destination register.

2.2 Armv8-A Architecture

The processor used in the thesis is Cortex-A53 processor from Arm, which implements the Armv8-A architecture.

The Armv8-A architecture includes two execution states, 32-bit and 64-bit, and each of the execution states have their own instruction set.

1. AArch32: There will only be 16 128-bit wide and 32 64-bit wide registers in this instruction set. Maximum width for load and store operations is 64 bits. Completely different register aliases compared with AArch64.
2. AArch64: There are in total 32 128-bit registers in the AArch64 execution state of Armv8-A architecture.

2.3 Arm Neon

Arm's implementations of the Advanced SIMD architecture are referred to as Arm Neon. Armv8-A includes a separate register file, a set of 128-bit wide registers, for performing SIMD operations. The Neon unit is completely integrated into the CPU, sharing resources for integer operations, loop control, and caching. When compared to a hardware accelerator, this significantly saves space and power.

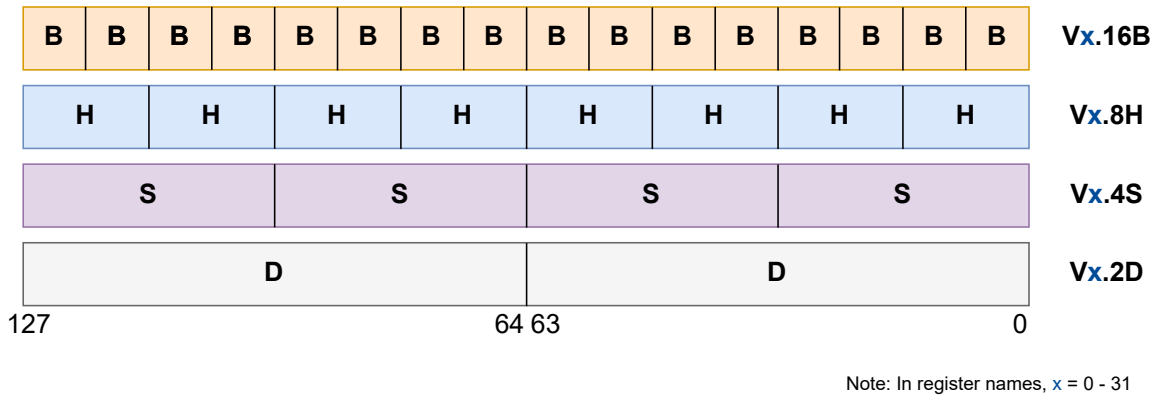

Figure 2.3: ARMv8-A Neon Register Packing

Figure 2.3 describes the packing of 128-bit Neon registers in Armv8-A architecture. Note, in the case of AArch32 execution state, where only 64-bit Neon register width is allowed, only the 64 least significant bits are used. As illustrated in figure 2.3, a 128-wide register can be packed as follows:

- Sixteen 8-bit elements, represented by the operand suffix .16B where B indicates byte.
- Eight 16-bit elements, represented by the operand suffix .8H where H indicates halfword.
- Four 32-bit elements, represented by the operand suffix .4S where S indicates word.

- Two 64-bit elements, represented by the operand suffix .2D where D indicates double word.

2.4 Utilizing Arm Neon

Neon technology can be used in the following ways:

1. Hand-coded Neon assembler
2. Auto-vectorization
3. Neon intrinsics
4. Neon-enabled libraries

2.4.1 Hand-coded Neon assembler

Hand-coded Neon assembler uses Neon Assembly instructions to vectorize the data. Neon assembler code will help in maximizing the optimal performance of the code using carefully hand-written assembler code that yields the best results from Neon, especially for performance-critical applications. The disadvantage is that it is difficult to maintain and write assembly code. However, with a limited time, this way of optimizing will not be investigated in the thesis.

2.4.2 Auto-vectorization

Auto-vectorization is the technique of allowing the compiler to automatically detect possibilities to use advanced SIMD instructions in the source code. Essentially, compiler optimization includes two techniques: loop unrolling and Superword-Level parallelism (SLP) optimization. The compiler targets loops and merges similar independent scalar instructions into vector instructions. Compilers that support auto-vectorization include Arm Compiler 6, Arm C/C++ Compiler, LLVM-clang, and GCC.

The following are a few advantages of employing auto-vectorization:

- The source codes written in high-level languages, such as C, and C++, are portable unless there's the inclusion of architecture-specific code elements such as inline assembly or intrinsics.
- Auto-vectorization requires significantly less design time compared to writing hand-optimized assembly code.
- Targeting a specific micro-architecture can be as simple as changing a single compiler option, but optimizing an assembly program demands a thorough understanding of the target hardware.

However, auto-vectorization isn't always the best option as there are chances when the compiler fails to identify opportunities to use Neon thereby failing to produce completely optimized code.

2.4.3 Neon Intrinsics

Intrinsics are functions whose exact implementation is known to a compiler. Accordingly, when generating the assembly code, the compiler replaces this function calls with an appropriate instruction or sequence of instructions. Intrinsics provide almost the same control as writing assembly language, but the compiler handles register allocation and pipeline optimization.

Neon Intrinsics are a collection of C and C++ functions that are included in the `arm_neon.h` header file. They are supported by Arm compiler and GCC as well. The following piece of code is one example that describes the usage of intrinsics.

```
//Intrinsic Function usage in C/C++
float64x2t C = vaddq_f64(float64x2t A, float64x2t B);

//Assembly code generated for the above instruction
fadd v0.2d, v1.2d, v0.2d
```

The 'vaddq_f64' function is an intrinsic function that essentially adds two vectors, each containing two elements of size 64-bit and returns a similar vector. In the above-provided instruction code, compared with the assembly instruction code generated by the compiler, it can be seen that the vectors A and B are placed in the two source SIMD registers 'v0' and 'v1' respectively and '.2d' indicates that each register has two lanes.

2.4.4 Neon-enabled Libraries

There are also a few libraries that provide support to employing Neon technology. One such library, termed Ne10 [22], is focus in this thesis. It is an open source software library maintained by Arm, targeting Arm architectures, which provides optimized implementations of essential operations in general math, signal processing, image processing, and physics functions. The Ne10 project was designed to provide a collection of widely used functions that have been significantly optimized for the Arm architecture and offer dependable, well-tested behavior that is simple to include into programs. Both the assembler and Neon implementations come with C interfaces to the routines [23].

2.5 Performance metrics

Execution Time Measurement

Chronos Library

In this thesis, to measure the execution time `std::chrono` library is used. It is a part of in C++14 Standard Library and includes three standard clocks that the user can interface with [18]. The **System clock** represents a system-wide real-time clock which is the machine's best guess of the current time on the wall clock. **Steady**

clock is the monotonic clock where each tick of the clock takes the same amount of time. **High resolution clock** is the clock with the shortest tick period. As a high-resolution clock provides high precision and resolution of up to nanoseconds, it was selected for the timing measurements.

Furthermore, Chronos provides real-time measurement. Real-time is also termed wall clock time which is the time counted from the start of the process to the end of it. This means, that since we are running tests over an OS, the time measurements include time slices of other processes which interrupted the process we were interested in. Consequently, the time measurements obtained would not be the actual time spent on the process we were interested in but would also contain the time spent on other high-priority tasks scheduled by the OS.

Perf tool

To obtain the CPU time, which is the actual time spent on the process by the CPU, one other tool known as Perf is investigated and employed in this thesis. The Perf tool is a performance analyzing tool for Linux [17]. It contains various sub-commands which could be used to get in-depth detail of the event/process. The `stat` sub-command can be used to get the CPU time of the particular task and the measurement has a resolution up to 9 decimal places (nanoseconds).

3

Implementation Models

As the title suggests, the investigation is done on both traditional and AI-based models in the thesis. This section provides a comprehensive background study of these two different models used in this work.

3.1 NMPC model

As a traditional-based approach, the non-linear MPC (NMPC), which is implemented in the Chalmers MSc thesis [16], is used. In [16], Hultgren and Husmark proposed the implementation of an offline-online coupled powertrain control solution for a three-mode hybrid electric vehicle to minimize the total energy consumption over a route. The offline component uses a simplified version of the powertrain to generate optimized State Of Charge (SOC) and velocity references through dynamic programming given speed limits and the topographic profile of a road segment. The online component acts on these references by generating torque setpoints to the power sources and selecting the most optimal gear using the optimal control technique of nonlinear model predictive control on a detailed dynamical model of the powertrain. The basic block diagram of the controller is shown in figure 3.1.

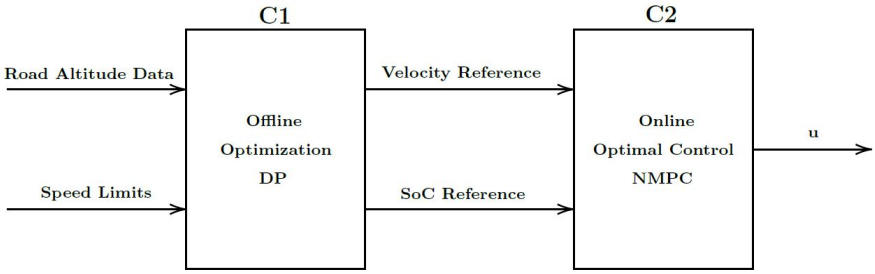

Figure 3.1: Block diagram of the control concept design [16]

In this case, offline optimization (C1 in figure 3.1) is only evaluated once per trip whereas online optimization (C2 in figure 3.1) is used several times with intervals. Also, C2 contains multiple vector operations which is an advantage of SIMD architecture. Thus, this thesis only considers online optimization (C2) for further SIMD implementation.

3.1.1 C2: Online NMPC Controller

The online NMPC controller is split up into two parts, namely, Elector and the Controller.

3.1.1.1 Elector

The Elector's job is to orchestrate the Controller by determining which hybrid mode in three modes i.e. Electric, Serial, and, Parallel mode and gear is best for the current driving situation. The desired mode and gear are elected by optimizing the objective function given by a specific NMPC until the next election is done.

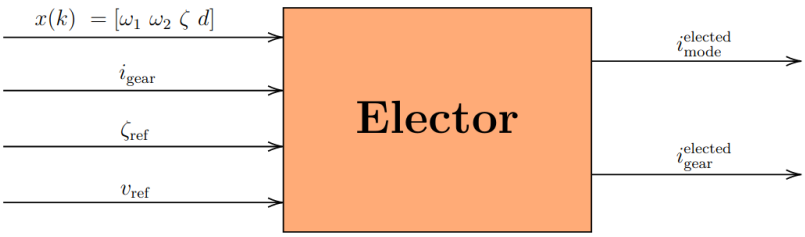

Figure 3.2: Representation of Elector [16]

Figure 3.2 shows the representation of Elector where ω_1 is ICE output shaft speed, ω_2 is gearbox input shaft speed, ζ is SOC, d is distance, v_{ref} is Velocity reference, ζ_{ref} is SOC reference from C1 optimization, and i_{gear} is the previous gear from the previous run.

3.1.1.2 Controller

Once the election is complete, the chosen hybrid mode-gear combination is sent to the Controller. The purpose of the Controller is to compute the input signals to the plant. Figure 3.3 represents the input and output of the controller. Here, $i^{elected}_{mode}$, $i^{elected}_{gear}$ are hybrid mode-gear combination elected by the elector, and x(k) are the state which will be optimized to get the desired output torques by using the IPOPT solver (see Section 3.1.2). ζ_{ref} and v_{ref} are the SOC and velocity reference computed by offline optimization. By taking these inputs, the controller runs at each time step to compute the various input torques by optimizing the previous states using the NMPC solver (IPOPT). The torque calculated is further required for the vehicle to follow the SOC and velocity references.

In both elector and controller, the solver used for nonlinear programs is "Interior Point OPTimizer", IPOPT [24] library for large-scale nonlinear optimization. This model is basically implemented in MATLAB and SIMULINK. Further, this SIMULINK model is used to generate C scripts for use in SIMD applications.

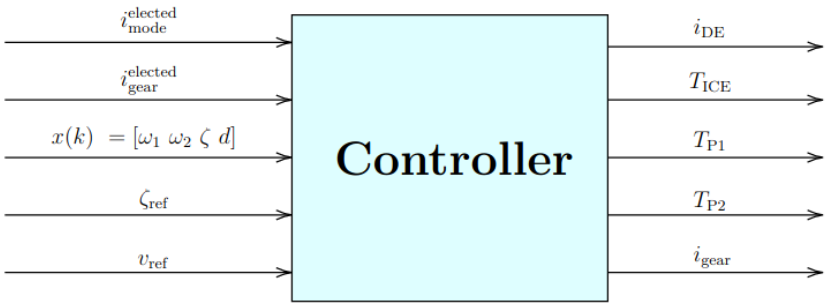

Figure 3.3: Representation of Controller [16]

3.1.2 IPOPT Library

IPOPT is an open-source software package that is written in c++ for large-scale nonlinear optimization. It is used to solve general nonlinear programming problems of the form,

$$\begin{aligned} \min_{x \in R^n} \quad & f(x) \\ s.t \quad & g^L \leq g(x) \leq g^U \\ & x^L \leq x \leq x^U \end{aligned}$$

where $x \in R^n$ are optimization variables, f : $R^n \rightarrow R$ is the objective function, and g : $R^n \rightarrow R^m$ are general nonlinear constraints. The functions $f(x)$ and $g(x)$ can be convex or non-convex, and they can be linear or nonlinear (but should be twice continuously differentiable). The nonlinear, non-convex constraints and objective functions in this work are solved by this IPOPT library. It is used to solve optimization problems written in a programming language such as C, C++, Fortran, or Matlab. To accomplish this, the following functions/methods provide the necessary information to IPOPT as input:

- Problem size [get_nlp_info] and bounds [get_bounds_info];
- Starting point [get_starting_point];
- Function values $f(x_k)$ [eval_f] and $g(x_k)$ [eval_g];
- First derivatives $\nabla\ f(x_k)$ [eval_grad_f] and $\nabla\ c(x_k)$ [eval_jac_g];
- Second derivatives $\sigma_f\ \nabla^2\ f(x_k) + \Sigma_j\ \lambda_k^{(j)}\ \nabla^2\ c^{(j)}\ (x_k)$ [eval_h];

3.2 AI model

Since the introduction of AlexNet [25] in 2012, progress in the field of AI, particularly Deep Neural networks (DNN), has accelerated exponentially. With their matrix multiplication strategies, these DNNs provide the most significant opportunity to exploit parallelism, in return demand a significant computation power for training and inference since they use large neural networks. One such neural network model is chosen in this thesis viz. xEV Battery State-of-Charge using a deep feedforward neural network (FNN) [26]. This model was chosen mainly considering its ability for potential exploitation in SIMD implementation, as being a DNN and the core of the DNN is Multiply–accumulate operation. The other considerations include the model being related to vehicle motion and energy field, and having open-source copyright.

The objective of this model is to utilize a deep FNN approach to estimate the battery State-of-Charge. The project was mainly done in MATLAB and outlines data collection, preparation, development, tuning, and robust validation of the FNN to sensor noise. The [26] model was subjected to datasets with errors purposefully added to the data during training to produce a robust estimator. For example, introducing cell voltage variation of 4mV, cell current variation of 110mA, and temperature variation of 50 °C. The error values were chosen to be comparable to the noise and error found in actual sensors used in commercially available electrical vehicles. The robust FNN trained from two Li-ion cells datasets, one for a nickel manganese cobalt oxide (NMC) cell and the second for a nickel cobalt aluminum oxide (NCA) chemistry cell, was shown to overcome the added errors and obtain a SOC estimation accuracy of 1% RMSE (Root Mean Squared Error) [26].

3.2.1 Neural network

Figure 3.5 shows the representation of a neural network of the model in [26]. The neural network contains a normalized input layer with 5 inputs i.e, voltage V, current I, temperature T, average voltage V_avg, and average current I_avg. As illustrated in the figure 3.5, inputs are further fed to three fully connected layers (hidden layers in figure 3.5) containing 55, 55, and 1 neurons respectively. The three hidden layers contain a non-linear activation function each, i.e, hyperbolic tangent, leaky Rectified Linear Unit (ReLU), and clipped ReLU, respectively. The hyperbolic tangent activation function, simply known as *tanh* function, maps any real number in the range 1 to -1 by applying *tanh* function to it. The leaky ReLU activation function receives a value as input and, if the value is positive, returns the same value. However, if the value is negative, the function returns the input value scaled with a constant number. The equation of the leaky ReLU is shown in the equation 3.1. The clipped ReLU function, as shown in equation 3.2 [27], performs a threshold operation where the input value is compared to a constant termed "ceiling." If the input value is less than 0, the function returns 0 and if the input value is greater than the ceiling constant, the output is set to the ceiling value. The input value is retained if it is between 0 and the ceiling value.

$$f(x) = max(scale * x, x) \tag{3.1}$$

$$f(x) \begin{cases} 0 & x < 0 \\ x & 0 \leq x < ceiling \\ ceiling & x \geq ceiling \end{cases} \tag{3.2}$$

In the neural network model at discussion, for the leaky ReLU layer, a scale of 0.3 is used whereas 1 is used as ceiling for the clipped ReLU layer. The model is trained in MATLAB for 50 epochs (total number of iterations of all the training data in one cycle for training the model) and is further used for optimization. Figure 3.4 shows an overview of how training and testing processes are carried out in a feed-forward neural network. The trained model is further saved, and the trained parameters i.e., weights and biases from the last epoch are extracted. These parameters will be used for the development of inference by exploiting SIMD in the thesis as shown in figure 3.4.

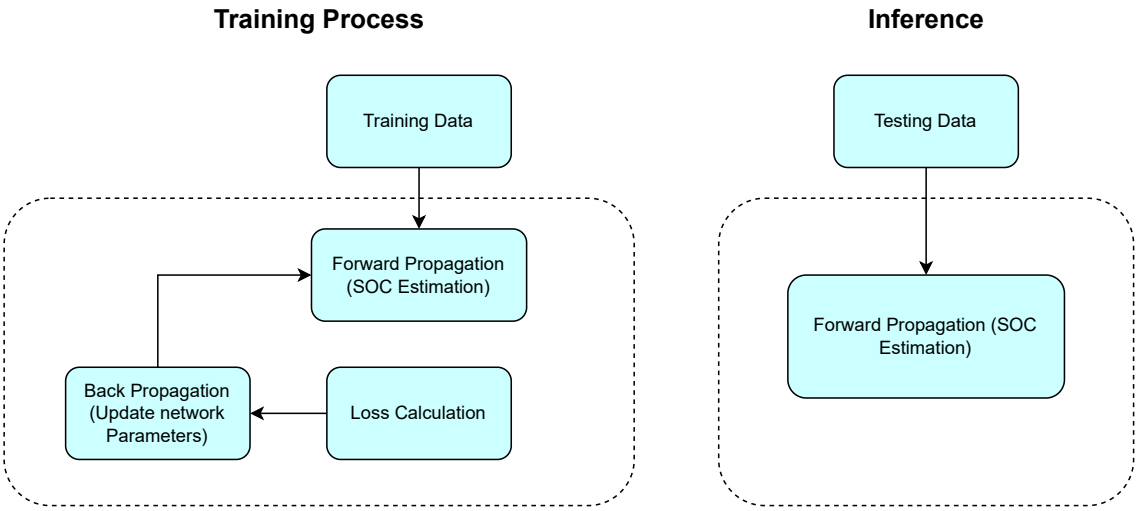

Figure 3.4: Representation of feed forward training and testing overview

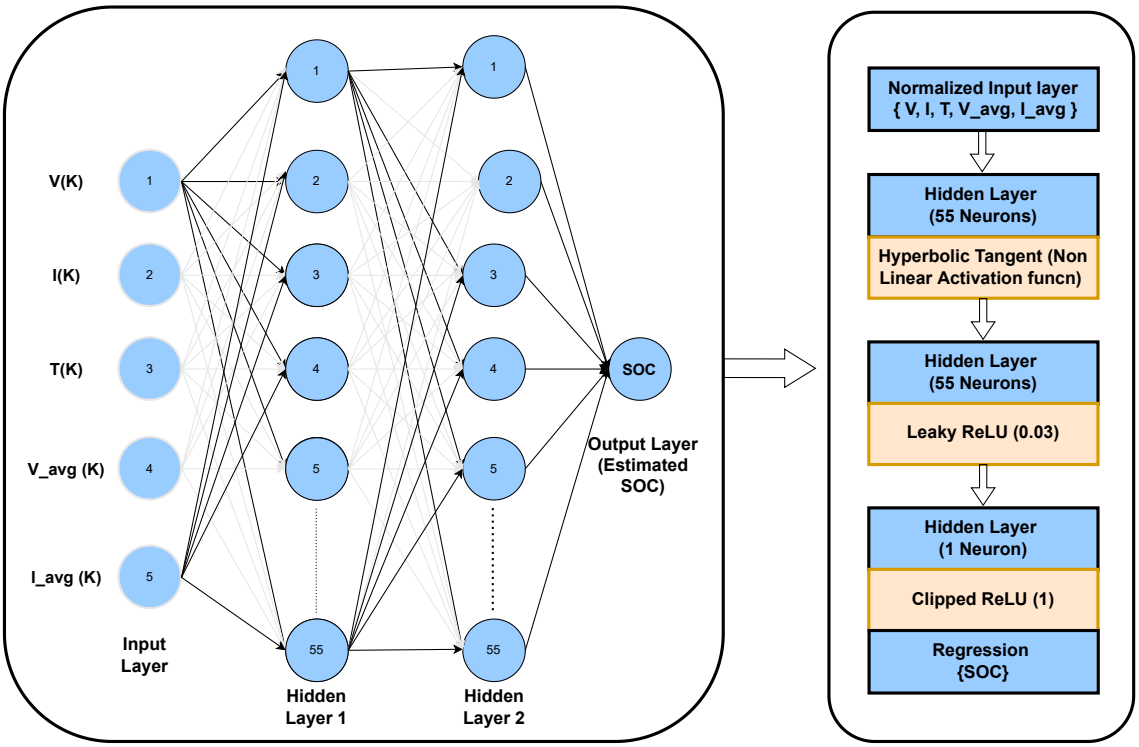

Figure 3.5: Representation of Neural Network

4

Methods

The purpose of this section is to describe how the various SIMD implementations were carried out by detailing the approach taken, and coding style used, etc.

4.1 Working Environment

The working environment, as illustrated in figure 4.1, is employed in this thesis. Programming and debugging are carried out in the development environment, which is an x86-64 machine running on Windows OS. The program is then compiled into a static binary file using a suitable cross-compiler. For the cross-compilation, GCC's **aarch64-none-linux-gnu** [28] is utilized. The GNU Toolchain is chosen since it is open-source and most extensively used. Once the executable is created, it is then transferred to the target environment, which is Raspberry Pi Zero 2W running on Raspberry Pi OS (64-bit), for performing evaluations.

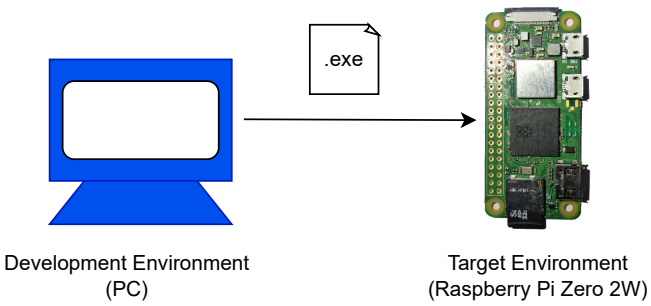

Figure 4.1: Setup Environment

4.2 NMPC Model Implementation, Optimization and Evaluation

In this thesis work, only the 'C2 Online NMPC Controller' has been used for evaluation. The controller is decoupled from the plant model which consequently limits the model to run only on initial inputs but not on updated inputs that are essentially the previous output of the controller.

4.2.1 Hardware Implementation

As mentioned previously in section 3.1.1, the controller was designed and developed using MATLAB/SIMULINK. For the hardware implementation of the controller, C codes were generated using the SIMULINK coder tool [29]. To interact with the controller, for providing inputs, forming the optimization problem, and receiving outputs, C/C++ codes were then manually developed. Since the Chronos library has been used for the timing measurements, the top file is written in C++. For building this entire project, the CMake tool [30] was employed which is further explained in the Appendix.

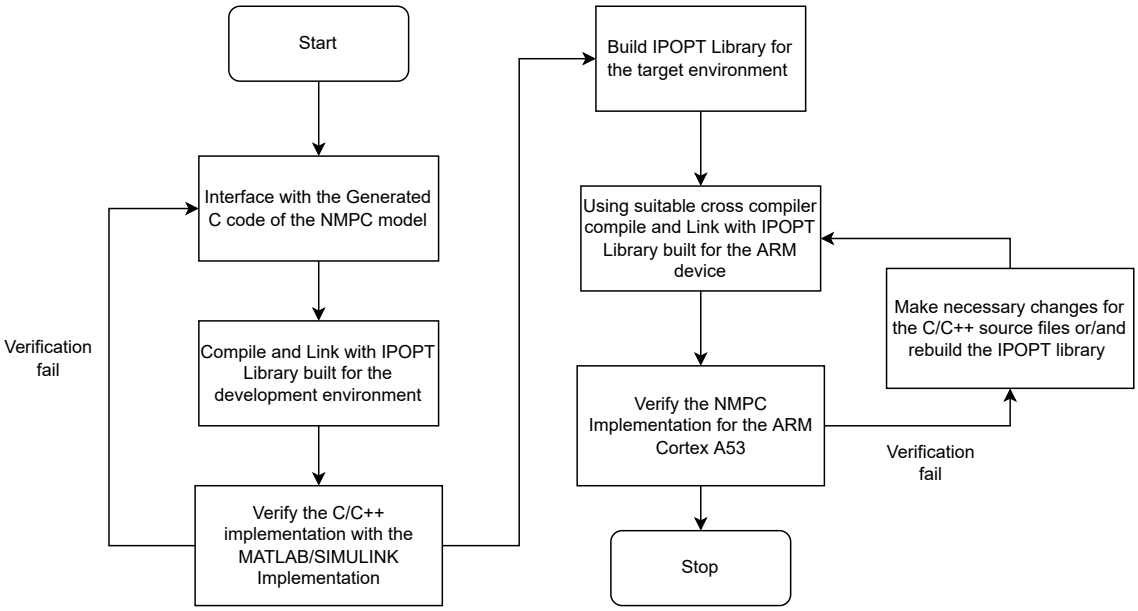

Figure 4.2: Flow chart for Hardware Implementation of the NMPC Model

4.2.2 SIMD Optimization

The NMPC model that is built in C is further used to optimize using SIMD architecture. Since SIMD is useful for vectorizing the sequentially computing scalar operations, it is necessary to find suitable complex operations which can be adopted or replaced with vector operations in the model. By thorough investigation, it is found that evaluation functions such as objective function, constraint function, etc, which are input to the IPOPT solver explained in section 3.1.2 have several multiple complex arithmetic operations that could be vectorized. As per profiling results shown in table 4.1, it can be seen that these evaluation functions are the least contributing execution time in the model; however, they were the major part that could be utilized for vectorization in the model. Thus, only these functions were selected for SIMD implementation in this thesis.

Table 4.1: Profiling results of NMPC model run for 1000 times.

Serial No.	Name	Percentage time (%)	Execution Time (sec)
1	dmumps_solve_node_fwd	4.32	0.15
2	init_malloc	3.17	0.11
3	dmumps_fac_asm_master	3.17	0.11
4	Ipopt::DenseVector::AddTwoVectorImpl	2.88	0.1
.	.	.	.
.	.	.	.
.	.	.	.
123	eval_h_EM	0.29	0.01
124	eval_jac_g_EM	0.29	0.01
125	eval_jac_g_Serial	0.29	0.01

Optimization through Auto-vectorization

As previously stated in chapter 2, auto-vectorization is used to optimize the NMPC model. By using the auto-vectorization technique, a compiler attempts to vectorize any blocks of code that are automatically recognized as being vectorizable. The open-source GCC compiler is utilized in this thesis since it provides auto-vectorization. With the GCC compiler, auto-vectorization can be enabled by setting optimization options like -ofast, -o3, etc. However, the compiler failed to identify much SIMD optimization in the model. This is because the compiler cannot optimize multiple arithmetic operations, but helps in producing optimized code while in loops.

Optimization through Arm Intrinsics

Further, Arm intrinsics are used to optimize the evaluation functions in the NMPC model. Following is one example of the usage of Arm intrinsics in one of the evaluation functions.

```
//Arithmetic scalar operations before SIMD implementation,
    t2 = in1[63] * in1[63];
    t3 = in1[64] * in1[64];
    t4 = in1[65] * in1[65];
    t5 = in1[66] * in1[66];

// Assigning address of in1[63] to in1_63 variable,
const double* in1_63 = &in1[63];

// Arithmetic vector operations using Arm Intrinsics,
float64x2_t t25_1=vmulq_f64(*((float64x2_t *)in1_63),\
    *((float64x2_t *)in1_63));
float64x2_t t25_2=vmulq_f64(*((float64x2_t *)in1_63+1),\
    *((float64x2_t *)in1_63+1));
```

Here, the scalar operations done to calculate `t2`, `t3`, `t4`, and `t5` are vectorized using *vmulq_f64* by taking two inputs at a time. For example, `(*((float64x2_t *)in1_63` which is used as an input to the intrinsic function contains two values i.e. `in1[63]` and `in1[64]` which will be multiplied as a vector and is assigned to the variable `t25_1`. The same procedure is applied to `t25_2` to vectorize the operations of `t4` and `t5` respectively. Hence, this procedure is carried out to replace most of the scalar operations present in the evaluation functions to vectorize them. Further, this is checked in the assembly code to ensure whether the compiler is vectorizing these operations.

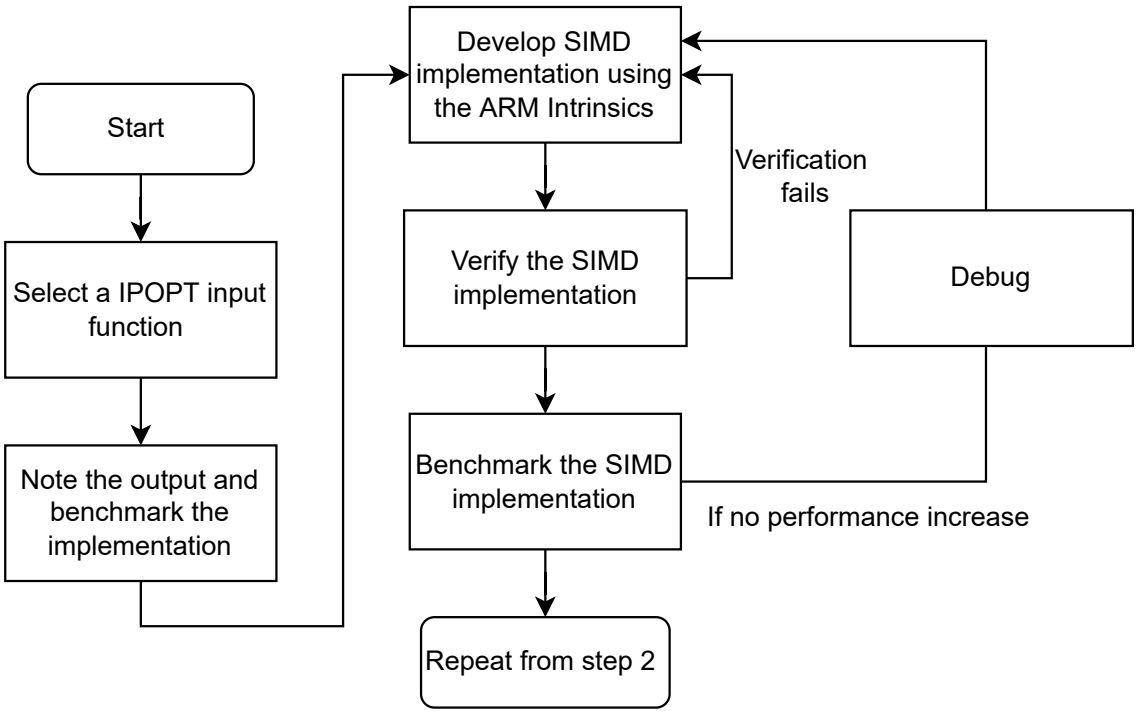

Figure 4.3: Flow chart for Neon Intrinsics Implementation of the NMPC Model

4.3 AI model implementation, Optimization, and Evaluation

Figure 4.4 shows how the AI model is implemented, developed, and used in this thesis for the benchmark.

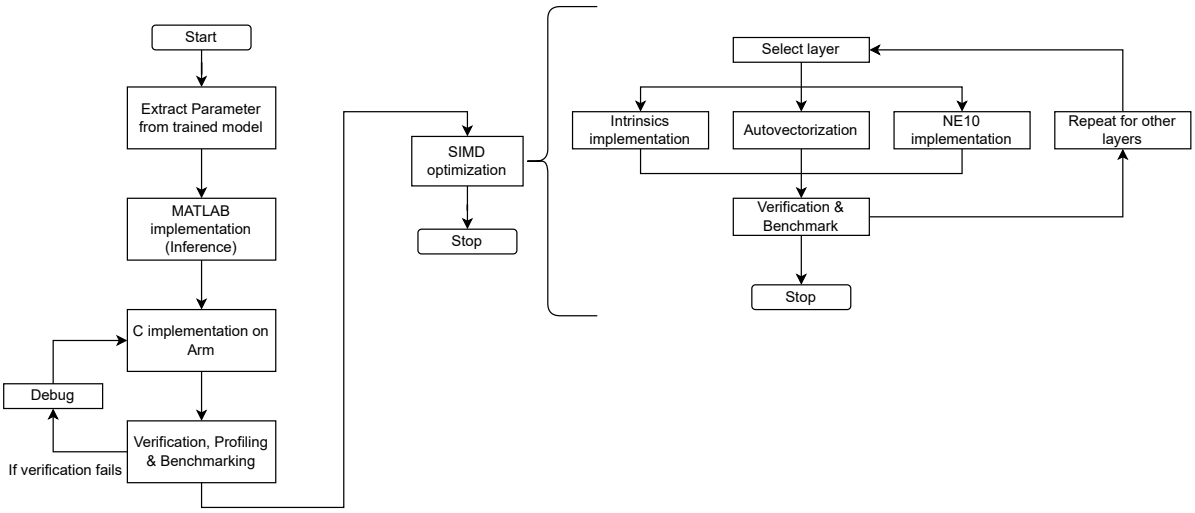

Figure 4.4: Flow chart for Neon Intrinsics Implementation of the AI model

4.3.1 Hardware implementation

The development of a model is done in C and is usually developed in two ways. One way is to use a generated C code from MATLAB, and another is to code it manually. However, for the maximum utilization of SIMD in the model, the second way is chosen. This means the model is built from scratch for every layer by extracting all parameters from the trained neural network. Further, verification is done by comparing the output with the MATLAB model which is developed by integrating a trained network in it. Therefore, by following this procedure, a model is built layer by layer. The entire project is built using the CMake tool, which is described in the Appendix.

4.3.2 SIMD Optimization

The AI model built in C is further used to optimize using SIMD architecture and benchmark. Since SIMD is useful for vectorizing complex scalar operations, selecting a suitable layer that yields more outcomes for SIMD is necessary. Hence to accomplish this, profiling is done for the model layers. As per profiling results shown in Table 4.2, a suitable layer will be selected for further implementations. As explained in chapter 2, optimization is done using auto vectorization, Arm Neon intrinsics, and the Ne10 library.

Table 4.2: Profiling results of AI model run for 1000 times.

Serial No.	**Name**	**Percentage time (%)**	**Execution Time (sec)**
1	Fully Connected Layer 2	76.19	0.16
2	Fully Connected Layer 1	9.52	0.02
3	Fully Connected Layer 3	9.52	0.02
4	tanh layer	4.76	0.01
5	Clipped ReLU Layer	0.00	0.00
6	Leaky ReLU Layer	0.00	0.00
7	Sequence Input Layer	0.00	0.00

Optimization through Auto-vectorization

As explained in section 2.4.2, the model developed in C is optimized using auto-vectorization. The auto-vectorization is a technique where a compiler automatically identifies the vectorizable block of codes and tries to vectorize it. In this thesis, an open-source compiler that supports auto-vectorization, GCC is used. Auto-vectorization can be activated in the GCC compiler by using optimization flags such as -ofast, -o3, etc.

Optimization through Arm Intrinsics

Individual layers of the AI model are further optimized using Arm-Neon intrinsics. The layer is selected for optimization as per the results of profiling. As explained in section 2.4.3, suitable functions in Arm Neon intrinsics are used for the optimization of all the layers in the model.

Following list 4.1 is one example of Arm intrinsics usage in one of the fully connected layers (hidden layer 2) in the figure 3.5. It is developed to optimize the matrix multiplication using Arm Neon intrinsic functions. It takes weight matrix 4×5 (A0, A1, A2, A3, and A4 vectors with 4 elements) and input matrix 5×1 (B) as an input at a time, the function performs vector multiplication on individual vectors with individual elements in B and provides 4×1 matrix as an output. The matrix multiplication is usually done in the following way as shown in eq. 4.1. As intrinsic functions are specifically developed for vectors, matrix multiplication is done by dividing its elements as vectors. Accordingly, the vectors A0, A1, A2, A3, and A4 are multiplied by each element in B respectively as shown in the equation 4.2.

$$\begin{pmatrix} A0[0] & A1[0] & A2[0] & A3[0] & A4[0] \\ A0[1] & A1[1] & A2[1] & A3[1] & A4[1] \\ A0[2] & A1[2] & A2[2] & A3[2] & A4[2] \\ A0[3] & A1[3] & A2[3] & A3[3] & A4[3] \end{pmatrix} \times \begin{pmatrix} B[0] \\ B[1] \\ B[2] \\ B[3] \\ B[4] \end{pmatrix} = \begin{pmatrix} C[0] \\ C[1] \\ C[2] \\ C[3] \end{pmatrix} \tag{4.1}$$

$$\begin{pmatrix} C[0] \\ C[1] \\ C[2] \\ C[3] \end{pmatrix} = \begin{pmatrix} A0[0] \\ A0[1] \\ A0[2] \\ A0[3] \end{pmatrix} \times B[0] + \begin{pmatrix} A1[0] \\ A1[1] \\ A1[2] \\ A1[3] \end{pmatrix} \times B[1] + \begin{pmatrix} A2[0] \\ A2[1] \\ A2[2] \\ A2[3] \end{pmatrix} \times B[2] + \begin{pmatrix} A3[0] \\ A3[1] \\ A3[2] \\ A3[3] \end{pmatrix} \times B[3] + \begin{pmatrix} A4[0] \\ A4[1] \\ A4[2] \\ A4[3] \end{pmatrix} \times B[4] \tag{4.2}$$

```
1
2 /* Function to multiply 4*5 matrix with 5*1 matrix using Arm Neon
     function */
3
4 void matrix_multiply_4x5_neon (float32x4_t A0, float32x4_t A1,
5 float32x4_t A2, float32x4_t A3, float32x4_t A4, float32x4_t B,
     float32_t B5, float32_t *C)
6
7 /* Initializing vectors C0 and C1 */
8 float32x4_t C0;
9 float32x4_t C1;
10
11
12 /* Using 'vmovq_n_f32' function to initialize the vectors C0
13 and C1 to zero */
14
15 C0 = vmovq_n_f32(0);
16 C1 = vmovq_n_f32(0);
17
18 /* Using 'vfmaq_laneq_f32' function to multiply and
19 accumulate in 4x1 blocks, i.e. each column in C */
20
21 /* Multiply An with nth element of B and add it to C to all the
     elements in the vector */
22 C0 = vfmaq_laneq_f32(C0, A0, B, 0);
23 C0 = vfmaq_laneq_f32(C0, A1, B, 1);
24 C0 = vfmaq_laneq_f32(C0, A2, B, 2);
25 C0 = vfmaq_laneq_f32(C0, A3, B, 3);
26
27 /* Using 'vmulq_n_f32' function to multiply a vector with a scalar */
28 C1 = vmulq_n_f32(A4, B5);
29
30 /* Using 'vaddq_f32' function to add two vectors */
31 C1 = vaddq_f32(C0,C1);
32
33 /* Using 'vst1q_f32' function to store a vector to C variable */
34 vst1q_f32(C, C1);
35
36 }
```

Listing 4.1: Intrinsics implementation of Fully Connected Layer 2

Optimization through Arm Neon library

Similar to the Neon intrinsics implementation, the Ne10 implementation is carried out. As discussed in section 2.3, the Ne10 library contains the optimized implementation of various common math, physics, and other functions. Unlike the Neon intrinsics, the usage of the Ne10 library for utilizing the Neon technology is straightforward.

Following is a code snippet, 4.2, of the layer 2 implementations of the SOC AI model used in the thesis. Compared to list 4.1, it can be observed that the usage of Ne10 functions are straightforward. Ne10 functions, for instance the `ne10_mul_float_neon` function in the list 4.2, takes the entire array as input, irrespective of the register width and individual data element size, and manages the packing of the data in the SIMD register for computation. Thus making the functions convenient for the user. Whereas in intrinsic implementation, the user must have knowledge about the width and the lanes of the SIMD register and then, based on that, has to choose a specific function to compute the result.

```
#include <../inc/NE10.h>
void fullyConnectedLayer1(float *in1, float in2[1][55]) {
    float tempMAT[5][55];
    float temp[5];
    for (int i=0; i<55; i++) {
        ne10_mul_float_neon (temp, in1, layer2Weight[i], 5);
        // Transpose the Matrix
        tempMAT[0][i] = temp[0];
        tempMAT[1][i] = temp[1];
        tempMAT[2][i] = temp[2];
        tempMAT[3][i] = temp[3];
        tempMAT[4][i] = temp[4];
        in2[0][i] = 0;
    }
    for (int j=0; j<5; j++) {
        // Addition of the Matrix Dot Products
        ne10_add_float_neon (in2, in2, tempMAT[j], 55);
    }
    //Addition of the Bias
    ne10_add_float_neon (in2, in2, layer2Bias, 55);
}
```

Listing 4.2: Ne10 implementation of Fully Connected Layer 2

GPU implementation

AI inference is primarily performed on GPUs and TPUs, with SIMD being less sought-after for the same use case. As a result, it is crucial to benchmark on a GPU to understand the performance differences compared to SIMD. The AI model inference on the GPU is accomplished using CUDA programming. CUDA programming is an extension of the C/C++ programming language and is largely similar to it. Essentially, the block of code that needs to be run on the GPU has to be present in a function with the specifier being "`__global__` ". A function with this specifier is referred to as a kernel, and the GPU can only operate with kernels; it cannot work

with any other functions or pieces of code in the source files. Further, before execution of the kernel, the data related to or required for the computations in the kernel has to be copied to the GPU. This movement of the data to and forth between the GPU and the host can be accomplished by using the CUDA API `cudaMemcpy`. The `cudaMemcpyHostToDevice` argument is required to be passed in the `cudaMemcpy` API to transfer data from the host to the device's memory (the GPU), and the `cudaMemcpyDeviceToHost` argument is required to copy data back from the device to the host.

```
1 /*Sequence Input Layer GPU Implementation*/
2 __global__ void sequenceInputLayerKernel(const float *A, const float *B
      , float *C, int numElements) {
3
4   int i = blockDim.x * blockIdx.x + threadIdx.x;
5
6   if (i < numElements) {
7     C[i] = A[i] - B[i];
8   }
9 }
```

Listing 4.3: Sequence Input Layer GPU Implementation

The block of code in list 4.3 demonstrates how the sequence input layer from the SOC AI Model is implemented using the CUDA programming. As described in the previous paragraph, a function named `sequenceInputLayerKernel` with the specifier `__global__` is declared. This makes it a kernel, which during compilation signifies to the compiler, `NVCC`, to generate respective machine code to run this function on the GPU rather than the CPU. The global index or thread indexing is computed in line 4 of list 4.3 which is used to select the elements of the array. The predefined variables such as `gridDim`, `blockDim`, `blockIdx`, and `threadIdx` which provide information on the dimension of the grid, dimension of the block, index of the block, and index of the thread respectively, help in the computation of the global index.

5

Results

In this section, the results of the benchmarking of various implementations in terms of execution time and memory consumption are presented. Furthermore, the comparison of the performance (execution time) of Neon with the GPU is detailed.

5.1 Execution Time

As explained in the previous section, to compare the performance changes of different implementations, the models implemented in basic C code are considered standard and termed as *base* in this thesis work.

Since the bench-marking is carried out on top of the Raspberry Pi OS, the dynamic clock setting is originally activated in the OS. With the dynamic clock enabled, the OS manages the clock rate depending on the load in order to save power. Hence, the dynamic clock is disabled by adding the following lines detailed in listing 5.1 in the `/boot/configtxt` boot file.

```
1 //Clocking at 1 GHz
2 arm_freq=1000
3 over_voltage=6
4 force_turbo=1
```

Listing 5.1: Environment variables setting to disable dynamic clocking

The `arm_freq` parameter can be used to set the maximum frequency of the CPU in the SOC. The `over_voltage` parameter sets the level of voltage consumed by the core, and the `force_turbo` parameter, if activated, forcefully runs the CPU at maximum frequency even during the idle state.

Hence, by overclocking the Raspberry Pi device, clock rate was maintained at an almost stable rate when benchmarking for different workloads.

Furthermore, inconsistencies in the measured execution times were observed when an implementation was benchmarked multiple times. Although the variances were negligible, an average of 10 repeated runs were captured for every executable timing measurement.

5.1.1 SOC AI Model

The optimization flag used when compiling plays a vital role in the executable's run time and code size. The optimization can target either on the code size (Os) or

execution time (`O0`, `O1`, `O2`, `O3` or `Ofast`). The user has the freedom to set the type and level of the optimization.

Initially, when generating the *base* executable, the compiler optimization was set to `O0`. However, benchmarking with the auto-vectorization implementation which is compiled at `Ofast` to enable the usage of Neon technology seemed to be fallacious. The reason is that the GCC compiler when compiling at `Ofast` not only optimizes using the co-processor under discussion but also employs several other techniques which are not in the interest of the thesis. The various techniques involved at different optimization levels could be found on Linux's GCC website [31]. Since compiling at `O3` and then switching off vectorization through an argument flag for realistic measurement was not available, the *base* implementation is compiled at `O2`, where there are barely any techniques involved other than vectorization.

The accompanying bar graph viz. figure 5.1, and table 5.1 detail the results of the execution time benchmarking for the SOC AI Model for each of the four implementations: *base*, *auto-vectorization*, *Ne10*, and *intrinsics*.

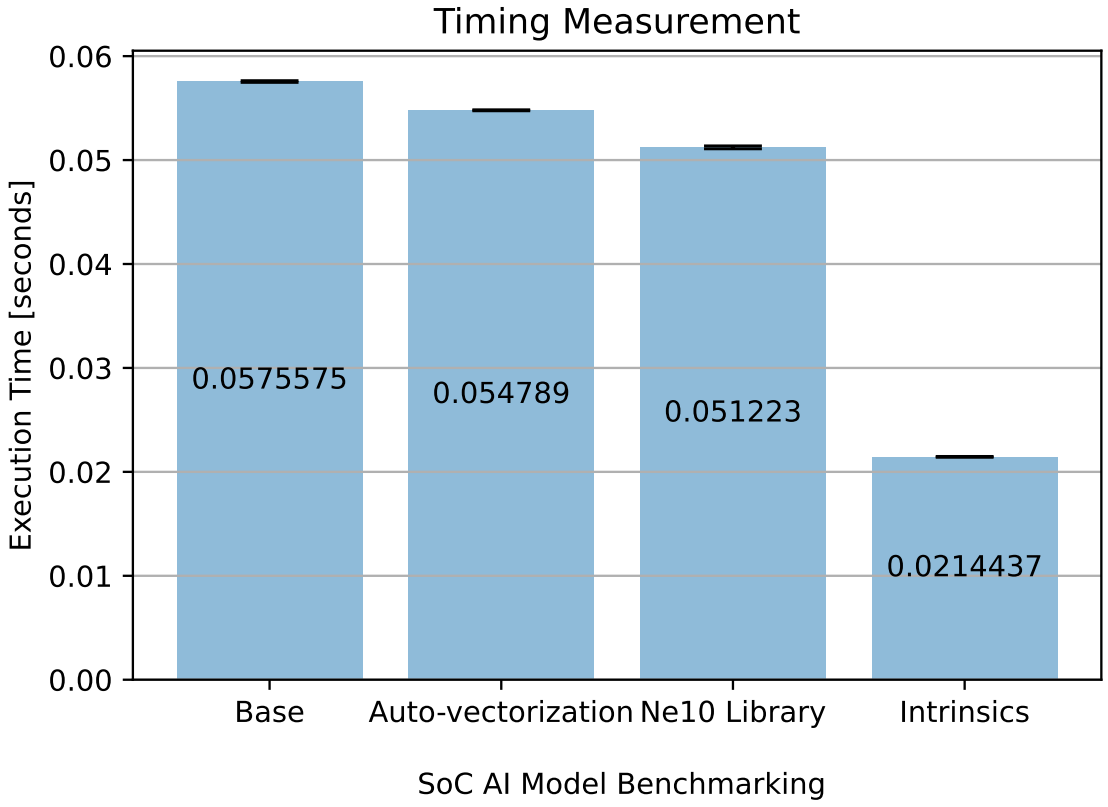

Figure 5.1: Comparison of execution time between various implementations of the SOC AI model

The *base* implementation which is considered as standard for comparison had dynamic instruction count of 31,025,571 and took 0.0576 sec to execute. Comparatively, the *auto-vectorization* implementation resulted in performance improvement of 4.8% and a drop of 6% in the dynamic instruction count. When the assembly code was examined, it was found that only 2 layers out of the 7 layers in the AI

Table 5.1: SOC AI Model Execution Time Benchmarking Results

Implementations	Mean Execution Time (sec)	Instruction Count	Execution Time Percentage Change w.r.t Base	Instructions Count Percentage Change w.r.t Base
Base	0.0575575	31025571	-	-
Auto-vectorization	0.0547890	29132224	-4.8 %	-6.10254 %
Ne10	0.0512230	34653925	-11%	+11.7 %
Intrinsics	0.0214437	13469458	-62.7%	-56.58%

Model were vectorized. These two layers were the sequence input layer and leaky ReLU layer (see figure 3.5), and from the profiling results in table 4.2, it can be seen that these were the two least contributing factors to the execution time. Hence, only a mere performance increase of about 5% was obtained.

As for the implementation using the intrinsics, a performance boost of 62.7% was achieved as there was significant amount (56.5%) reduction in the instruction count. Since the implementation was performed such that 4 data elements were processed at once, we expected around 75% performance gain as a fourfold reduction of 0.0575 sec accounts to 0.0144 sec. It is challenging to reach this theoretical performance value practically. One explanation could be that it was observed that the length of the array influences majorly. The sequence input layer contains very few elements, and its execution time was found to be the same, 4.39 ms, whether it was performed on a CPU or SIMD using intrinsics. Although it is intriguing to learn how the length of an array affects the performance (execution time), this topic area is not further investigated in this project work as it was out of the scope of the thesis.

For the Ne10 implementation, about an 11% decrease in the execution time was obtained. Surprisingly, in contrast to other implementations, there was an increase in the instruction count. But, these instructions on average consumed 1.282 clock cycles to execute whereas the *base* implementation instructions consumed 1.694 clock cycles, which explains the reason Ne10 implementation outperformed the base implementation. Furthermore, scrutinizing the assembly code generates, it was observed that, although the 4 data elements are processed simultaneously, the load and store operations were using the D registers which are 64 bits wide. Hence, before and after computations, the data-transfer operations were operating on only 2 elements instead of 4.

5.1.2 NMPC Model

As described in the previous sections, the evaluation functions were optimized by utilizing Arm Neon technology. Unlike in SOC AI Model benchmarking, the NMPC benchmarking was done on only three implementations excluding the Ne10 implementation. Since the evaluation functions in the NMPC model contained data types of 64 bits and the math functions in the Ne10 library were only developed for the data types of width 32 bits, the Ne10 implementation was dropped. Further, as data types in the evaluation functions were 64 bits wide, we could only pack two data elements at a time in the Intrinsics implementation.

The bar graph in Figure 5.2 details the obtained benchmarking results (execution

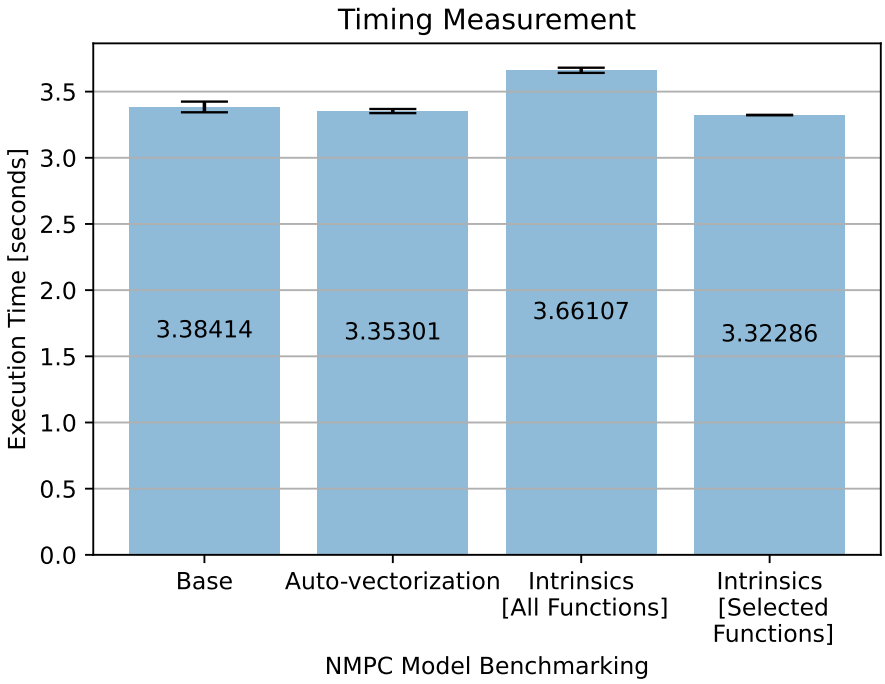

Figure 5.2: Comparison of execution time between various implementations of the NMPC model

time) for the NMPC model. The *base* implementation of the NMPC Model took around 3.38 sec to execute while the auto-vectorization implementation took around 3.35 sec to execute. While examining the assembly code it was noticed that the GCC compiler had completely failed to compute using the Arm Neon SIMD registers. The 0.828% decrease we see in execution time is not due to vectorization but due to other compiler optimization techniques such as inlining functions to remove the function call overhead.

In the case of intrinsics implementation, when all the Neon-optimized evaluation functions were used to run the NMPC model, the model took 8.28% more time to execute compared to the *base* implementation. Unlike the SOC AI Model, the NMPC model had a very large number of small blocks of code that could be vectorized. And not everything in these blocks of code could be run on the SIMD co-processor, but some need to be computed on the CPU. Furthermore, when the assembly code was scrutinized, we could observe more data-transfer instructions between the Neon and CPU registers which would explain the reason for deterring performance.

Not all evaluation functions were rendered non-optimized when using intrinsics. Employing the functions that were optimized we could get a mere 1.7% which is as expected that these were insignificant contributors in the execution time with at most 0.3%, as obtained from the profiling result in table 4.1.

5.2 Memory Usage Benchmark

In order to perform the memory consumption benchmark for the various implementations in the thesis, a customized memory footprint metric is defined based on the needs of CEVT. Memory footprint is defined as the aggregate of the size of the binary on the disk and peak run-time memory consumption. Run-time memory consumption is further defined as the sum of the stack and heap memory allocation for a particular process. Furthermore, it is important to note that the peak memory allocation value during the run-time of an implementation is considered but not the accumulation of the memory allocation values over the time of the run. For the measurement of the memory allocation `Valgrind` tool is used.

Table 5.2 provides the result obtained from the memory footprint benchmarking. The Ne10 implementation consumed much of the disk space compared to any other implementation. The reason is that the Ne10 implementation was statically linked to the Ne10 library for generating the executable, meaning that the resulting binary file would contain the machine code for both the used and unused functions in the implementation. The reason for not linking dynamically is that the dynamic linking references to the library would be made during run-time causing additional overhead which would affect the execution time, the primary performance metric considered in the thesis. It is interesting to note that, although the standard implementation, *base*, occupied the least disk space, there is no relationship between the execution time and size as the *Intrinsics* implementation has slightly larger binary size than the *Auto-vectorization* implementation. Furthermore, it can be observed that for all four implementations the peak run-time memory consumption is the same, being 8880 Bytes. In all the cases the CPU allocated fixed-size stack memory of 8880 Bytes and heap memory allocation was found to be null as there was no object creation in any of the implementations.

Table 5.2: SOC AI Model Memory Footprint Benchmarking Results

Implementations	Size of the Executable on disk (B)	Run-time Memory Consumption (B)	Total (B)
Base	27,048	8,880	35,928
Auto-vectorization	28,416	8,880	37,296
Ne10	437,884	8,880	446,764
Intrinsics	28,536	8,880	37,416

5.3 Execution Time SIMD vs GPU

Since we encountered a hard time installing the *Perf* tool on the Linux4Tegra OS which the Jetson Nano runs on, the *Chronos* library was used to take timing measurements.

The Jetson Nano GPU took 2.22 sec to execute the SOC AI model, while the Neon Intrinsics implementation on the CPU took a mere 0.0214 sec to execute. It was

assumed that copying data from and to host memory and device memory could be the issue for the bottleneck. Hence, to perform further analysis profiling was carried out and the result is detailed in table 5.3.

Table 5.3: SOC AI Model Profiling in Jetson Nano GPU

Sl No.	**Name**	**Percentage time (%)**	**Execution time (ms)**
1	fullyConnectedLayer2Kernel	28.11	34.538
2	[CUDA memcpy HtoD]	21.19	26.037
3	fullyConnectedLayer1Kernel	13.89	17.070
4	[CUDA memcpy DtoH]	12.44	15.284
5	sequenceInputLayerKernel	6.42	7.8927
6	tanhLayerKernel	6.32	7.7609
7	leakyReLULayerKernel	5.95	7.3121
8	fullyConnectedLayer3Kernel	5.67	6.9720

As we observe from the profiling results, although the memory copy API *CUDA memcpy HtoD* and *CUDA memcpy DtoH* are among the significant contributing factors for the execution time, still the *fullyConnectedLayer2Kernel* tops the list with 34.5 ms compared to Neon Intrinsics that barely took 15.4 ms.

6

Conclusion

According to the aim of this thesis, the two selected models were investigated and benchmarked on an Arm processor utilizing its SIMD component. The investigation was done for all possible methods in Arm Neon such as Arm Neon intrinsics, auto-vectorization, and a Neon-enabled library (Ne10). Further, two performance metrics are considered when benchmarking: execution time and memory footprint. Since execution time is considered the primary metric in the thesis, unless otherwise stated, the term "performance" in this section refers to execution time. In general, when used for optimization, the fixed width of the SIMD register itself might become a shortcoming. For instance, consider an application that contains most of its data as double-precision (64-bit) values. If this application needs to be optimized using 64-bit wide SIMD registers, then the developer has to compromise the accuracy of the model as only 32-bit or lesser-width data is allowed on the registers.

Irrespective of the target models for optimization, the following conclusion can be reached based on the results obtained in the thesis. Optimization using auto-vectorization doesn't provide satisfactory results as the compiler fails to recognize most of the vectorizable blocks of code while dealing with complex models. Hence, an open-source toolchain like GCC is not recommended if one relies on the auto-vectorization method. Although it only offers a slight performance gain, auto-vectorization is nevertheless advantageous to use as the compiler does the optimization and doesn't require the developer to modify the code for the optimization.Comparatively to other methods, Neon-enabled libraries, such as Ne10, do provide satisfactory optimization. Although these libraries are straightforward to use, they have their own flaws. For instance, the Ne10 library used 64-bit registers to load and store the data even when the 128-bit registers were available, hampering the performance gain. On the other hand, considering the performance boost obtained, Arm Neon intrinsic is advised when compared to other methods used for comparison in the thesis. In comparison to the Ne10 library, the functions in Arm Neon intrinsics require the developer to manually pack the data into the SIMD register for computations. This provides more flexibility for the user to efficiently utilize the SIMD register and obtain theoretical performance gains.

The results show that the Arm Neon approach works well for an AI model than MPC model, and the reason is further discussed while answering the primary thesis research question at the end of this chapter 6. As for the findings regarding the

memory footprint benchmarking, except for the Ne10 library, all the other methods had almost the same size. However, as the Ne10 library had to be statically linked, the size of the binary file generated was found to be at least 10 times larger than the other method. One significant finding was that there is no relationship between memory footprint and execution time. Regarding the performance of the selected AI model on Arm Neon and GPU, although the inference time was found to be lesser in Arm Neon, as previously stated in the chapter additional research is required to draw a firm conclusion(s). However, it was clear from the results (see section 5.3) that, for small models such as the selected AI model that are not very resource intensive, the memory copy operation between the CPU and GPU would bottleneck the execution time.

Hence to conclude with the results obtained, the following research question stated in chapter 1 is answered:

How well can both AI and MPC models be implemented on Arm processor using Neon technology?

Arm Neon is best suited for applications whose core operations can be vectorized. This is evident from the results (chapter 5) obtained in the thesis. As per the results obtained, the Arm Neon intrinsics method decreases the inference time of the AI model by 62% when compared to the unoptimized C implementation. However, a mere performance improvement was noticed when the MPC model was optimized using Arm Neon. The significant difference between the two models is the amount of code that could be vectorized in each of them. Since the chosen AI model is a DNN and its primary operations involve multiplication and accumulation, the majority of the code was optimized through vectorization. In contrast, the MPC model optimization target included small blocks of code. Additionally, these code blocks were not entirely vectorizable since they had a few operations that required computations to be performed on the CPU. The movement of data from Arm Neon to the CPU proved costly [32], and therefore the simultaneous usage of CPU and Arm Neon for computation depleted the performance gain. Hence, Arm Neon is recommended to be used when large blocks of code in the target application are purely vectorizable. Therefore, the selection of the application for SIMD optimization is crucial for exploiting the performance gain in terms of execution time.

7

Future Work

Based on the results and discussion presented, the following aspects can be addressed to carry forward the work in this project,

- Results in section 5.3 is not enough to determine whether the Arm Neon is superior to the GPU for inferences. Hence, further investigation has to be made either by performing performance tuning or experimenting with different implementation methods to conclude.
- In this thesis, the compiler used is GCC which is an open-source compiler. Instead, an Arm 6 compiler which is specifically built for Arm processors could be used for further investigation.
- In this thesis, the evaluation functions were used exclusively when optimizing the NMPC model. It would be of interest if all possible optimization could be done further in the investigation. For instance, optimizing the IPOPT Library as it contributed the most to execution time.
- It would be interesting to extend the study to a wider family of Arm cores, especially the Cortex A-77 and Cortex A-78 processors that implement the Armv8.2-A ISA. Since Armv8.2-A ISA or later supports half-precision floating point arithmetic operations on the Neon, it would be interesting to understand the accuracy and performance trade-off.
- Further, benchmarking in a microcontroller environment, at least on bare metal, should be prioritized over an Operating System (OS). Benmarking on OS introduces undesired noises when measuring execution time.

Bibliography

[1] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[2] C. K. Law, D. Dalal, and S. Shearrow, "Robust model predictive control for autonomous vehicles/self driving cars," *arXiv preprint arXiv:1805.08551*, 2018.

[3] G. Plastiras, M. Terzi, C. Kyrkou, and T. Theocharidcs, "Edge intelligence: Challenges and opportunities of near-sensor machine learning applications," in *2018 IEEE 29th International conference on application-specific systems, architectures and processors (asap)*. IEEE, 2018, pp. 1–7.

[4] M. Roopaei, P. Rad, and M. Jamshidi, "Deep learning control for complex and large scale cloud systems," *Intelligent Automation & Soft Computing*, vol. 23, no. 3, pp. 389–391, 2017.

[5] P. San Juan, A. Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí, "High performance and portable convolution operators for multicore processors," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 91–98.

[6] Y.-L. Lee, P.-K. Tsung, and M. Wu, "Techology trend of edge AI," in *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, 2018, pp. 1–2.

[7] M. Sung *et al.*, "SIMD parallel processing," *Architectures Anonymous*, vol. 6, p. 11, 2000.

[8] D. Seal, *ARM architecture reference manual*. Pearson Education, 2001.

[9] C. Lomont, "Introduction to Intel advanced vector extensions," *Intel white paper*, vol. 23, 2011.

[10] S. Oberman, G. Favor, and F. Weber, "AMD 3dnow! technology: Architecture and implementations," *IEEE Micro*, vol. 19, no. 2, pp. 37–48, 1999.

[11] S.-J. Lee, S.-S. Park, and K.-S. Chung, "Efficient SIMD implementation for accelerating convolutional neural network," in *Proceedings of the 4th International Conference on Communication and Information Processing*, 2018, pp. 174–179.

[12] S. Yagneswar, "Performance optimization of signal processing algorithms for SIMD architectures," Master's thesis, KTH Royal Institute of Technology, 2017.

[13] K. Karras, E. Pallis, G. Mastorakis, Y. Nikoloudakis, J. M. Batalla, C. X. Mavromoustakis, and E. Markakis, "A hardware acceleration platform for AI-based inference at the edge," *Circuits, Systems, and Signal Processing*, vol. 39, no. 2, pp. 1059–1070, 2020.

[14] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1107–1116.

[15] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Computer architectures for autonomous driving," *Computer*, vol. 50, no. 8, pp. 18–25, 2017.

[16] D. Hultgren and T. Husmark, "Three-mode hybrid powertrain optimal control to track offline optimized references," Master's thesis, Chalmers University of Technology, 2020.

[17] Perf: Linux profiling with performance counters. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[18] C++ chronos library. [Online]. Available: https://en.cppreference.com/w/cpp/chrono

[19] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[20] Y. Ngoko and D. Trystram, "Revisiting Flynn's classification: The portfolio approach," in *Euro-Par 2017: Parallel Processing Workshops*, D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, and J. Weidendorfer, Eds. Cham: Springer International Publishing, 2018, pp. 227–239.

[21] A. F. Hernández, "Yet another survey on SIMD instructions," 2013.

[22] Project Ne10. [Online]. Available: https://projectne10.github.io/Ne10/

[23] Ne10. [Online]. Available: http://web.archive.org/web/20160706043552/http://projectne10.github.io/Ne10/

[24] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical programming*, vol. 106, no. 1, pp. 25–57, 2006.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[26] C. Vidal, P. Kollmeyer, M. Naguib, P. Malysz, O. Gross, and A. Emadi, "Robust xEV battery state-of-charge estimator design using a feedforward deep neural network," *SAE International Journal of Advances and Current Practices in Mobility*, vol. 2, no. 2020-01-1181, pp. 2872–2880, 2020.

[27] Clipped Rectified Linear Unit (ReLU) layer. [Online]. Available: https://www.mathworks.com/help/deeplearning/ref/nnet.cnn.layer.clippedrelulayer.html

[28] Arm GNU Toolchain. [Online]. Available: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads

[29] SIMULINK Coder. [Online]. Available: https://www.mathworks.com/products/simulink-coder.html

[30] CMake. [Online]. Available: https://cmake.org/

[31] GCC - GNU project C and C++ compiler. [Online]. Available: https://linux.die.net/man/1/gcc

[32] Latency issue. [Online]. Available: https://developer.arm.com/documentation/ddi0344/k/ch16s05s02

A

Appendix 1

A.1 Build System: CMake

In the thesis, a CMake-based build system is setup for the NMPC Model as well as the SOC AI Model. CMake is open-source build system which makes easier to manage the build process of the project or software. Essentially, the CMake builds the project by reading the set of instructions in the text file named `CMakeLists.txt` inside the project. There can be more than one `CMakeLists.txt` as the developer has complete control over how many files to create and where to put them in the project. Further, by defining set of instructions in these files, the developer can configure on how to build or compile the project.

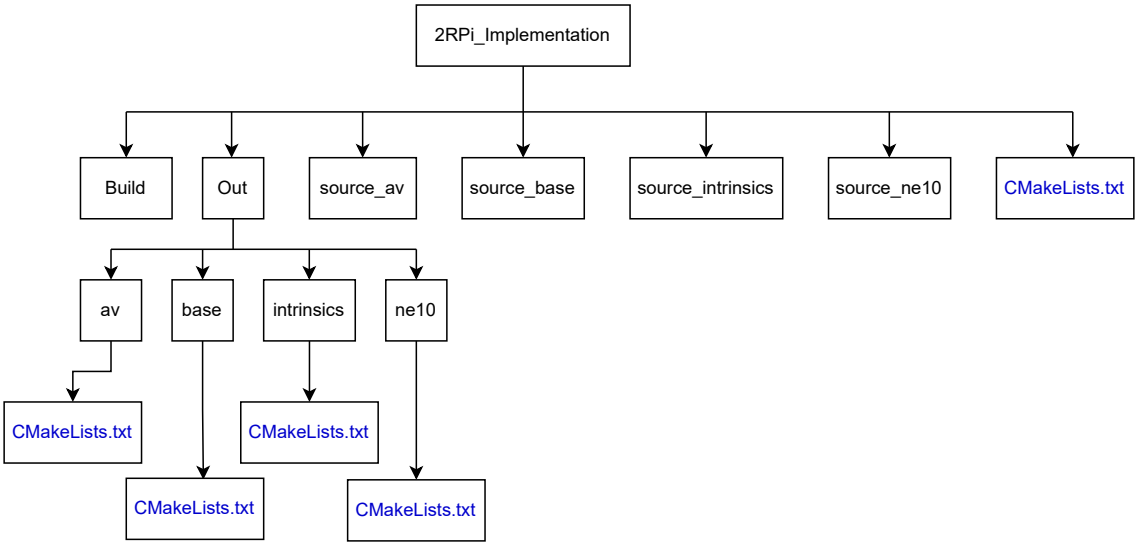

Figure A.1: Folder Structure for the SoC AI Model

The diagram in figure A.1 illustrates the folder structure used for working on the SoC AI Model and provides information on the location of the `CMakeLists.txt` files used by the CMake tool to build and generate the output. From figure A.1, it can be observed that one `CMakeLists.txt` file placed at the top most directory `2Rpi_Implementation`, and four placed inside the `Out` directory. Separate sub-directories are made in the `Out` directory for each of the implementations: *base*, *av*, *ne10*, and *intrinsics*. The build process is mostly defined in the `CMakeLists.txt` file located in the `Out` sub-directories, while the top most CMake file only specifies

the locations of the these four CMake text files. The instructions in these four files provides information to the CMake tool on which compiler to use to compile what files, location of the source codes, libraries, where to store the generated output file and so on. Since it is open-source and straightforward, CMake was chosen to set up the build environment.

A.2 Measuring Execution Time

A.2.1 Chronos Library

As already discussed in the section 2.5, Chronos library is employed and investigated in this thesis project. Out of the three clocks available in the Chronos for measurement, we have choosen the high resolution clock since we needed high accuracy. In order to measure the execution time using the Chronos, source code needs to be modified. As it can be observed from the example provided in listing A.1, before and after the relevant code section, whose execution time needs to be measured, the time must be fetched.

```
1
2     auto begin = std::chrono::high_resolution_clock::now();
3         .
4         /* Code Segment */
5         .
6     auto stop = std::chrono::high_resolution_clock::now();
7     auto elapsed = std::chrono::duration_cast\
8         <std::chrono::nanoseconds>(stop - begin);
9     auto elapsed_nano = elapsed.count() * 1e-9;
10
```

Listing A.1: Execution time measurement by using Chronos

A.2.2 Perf Tool

Perf is a profiler tool for Linux based systems that provides command line interface for the user with the information on the hardware differences in Linux performance measurements. In the thesis, Perf is predominantly used, where-ever possible, for the execution time measurement as hardware counters are made use of. Running `perf list` helps to find out all the measurable events in a particular CPU. The `-e` argument helps to specify the desired events to be measurement and for repeated measurements one could use `-r` argument followed by the desired number of repetitions. One example of employing the Perf tool for timing measurement is provided in listing A.2. The given example provides information on the events, that include number of cycles consumed and number of instructions executed when the executable is run. Further, 10 repeated measurements is taken before providing the results along with the mean and standard deviation.

```
1     perf stat -e cycles instructions -r 10 <executable_name>
2
```

Listing A.2: Execution time measurement with the Perf tool

A.3 Profiling Tools

A.3.1 Perf Profiling

Profiling with Perf is straightforward. The `perf record` command records all the profiling information in the function or API level. The `perf report` command output the recorded profiling information in the terminal.

```
1    perf record <executable_name>
2    perf report
3
```

Listing A.3: Profiling with Perf tool

A.3.2 Nvprof Profiling

Nvpof, similar to perf, is a profiling tool available in Linux for collecting profiling data such as kernel execution, memory transfer, etc. that are related to the CUDA activities happening on both GPU and CPU. It is included in the CUDA toolkit. For profiling the GPU implementation the Nvprof is employed. The command format for profiling with Nvprof is shown in listing A.4.

```
1    nvprof <executable_name>
2
```

Listing A.4: Profiling with nvprof tool

A.4 Memory footprint measurement

A.4.1 Valgrind: Massif tool

Valgrind, originally built as a memory debugging tool for Linux, has now developed into a instrumentation framework for building dynamic analysis tool. In the thesis, the *Massif* tool, which is a heap profiler, is obtained under the Valgrind is used to measure the run-time memory consumption of different implementation during their execution.

The format of command line for profiling an executable with the Massif tool is provided in listing A.5. After running the command, an file with name `massif.out.xxxx` is generated, where the file ending 'xxxx' represents random numbers. The generated file can then be parsed or converted to readable text file which would contain all the memory profiling information. The profiling result is represented in as graph and is also detailed in a table as seen in the reference snap provided in figure A.2.

```
1    valgrind --tool=massif --stack=yes <executable_name>
2
```

Listing A.5: Profiling with Massif tool

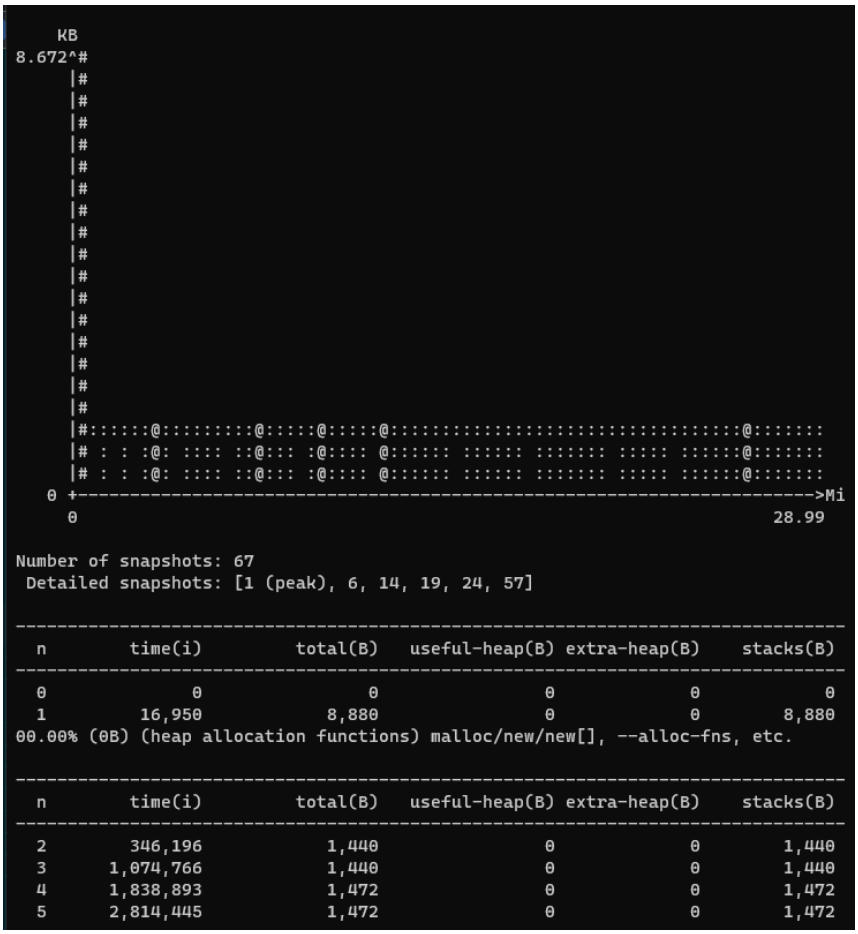

Figure A.2: Profiling result from Massif tool