Individual Project

Final Report

ID Number: B820928

Programme: Electronic and Electrical Engineering MEng

Module Code: 23WSD030/23WS50META

Project Title: Thread by Thread: Benchmarking and Comparing the Executions of Multi-threaded Applications on Multi-core Architectures.

Abstract:

Modern processors, both in desktop and embedded systems, increasingly feature multi-core architectures. This project aimed to optimise three applications using parallel computing techniques to decrease runtime and enhance scalability, testing them on desktop and embedded processors, including the Raspberry Pi 5. The applications included `mpbenchmark`, a benchmarking tool for multi-core performance; `MobileNet`, a machine learning model for image classification originally single-threaded and adapted to use the `OpenMP` library for multi-threading; and `DeBaTE-FI`, a test control framework for testing microcontrollers, optimised with Python's multi-processing and a high-performance `C++` library for `telnet` communication. Results showed significant performance improvements: `mpbenchmark` was redesigned with modern `C++` and SIMD intrinsics, improving scalability and design; `MobileNet` achieved an 87% performance increase on desktop processors, with a 60-65% gain on Raspberry Pi devices; and `Debate-FI` saw a 43.5-62.1% reduction in runtime across various setups. The project's contributions include a novel `C++` benchmark tool, the first parallel version of the `MobileNet` CNN model, and an optimised `Debate-FI` platform, all demonstrating enhanced performance scalability.

Contents

Introduction

The objective of this project is to enhance the performance of specific applications through parallel computing and to examine their efficiency on both desktop and embedded central processing units (CPUs). Parallel computing is a methodology employed in software development that enables certain segments of code to execute simultaneously, thereby diminishing the overall execution time of an application and augmenting its scalability. Contemporary processors boast a multi-core architecture, integrating multiple CPUs within a single chip, which is pivotal for facilitating parallelism [1].

Parallelism can be achieved via multi-threading and multi-processing techniques. A process refers to an active instance of a computer program, encompassing the program's code and its ongoing activities. Each process operates within its distinct memory address space and system resources, maintaining independence from other processes. However, processes can engage in communication with one another through various inter-process communication (IPC) mechanisms. Conversely, a thread represents the minimal unit of processing within a process. It executes within a process's context, sharing the process's resources like memory and file descriptors, yet it retains its distinct execution state, including the program counter, register set, and stack. Threads can facilitate the parallel execution of code segments within a single process [2].

Multi-processing involves an application initiating multiple processes to perform parallel operations, whereas multi-threading pertains to an application that generates multiple threads usually within a single process. This project concentrates on three main applications: the initial two are optimised utilising multi-threading and their outcomes are assessed on both desktop and embedded processors. The embedded processors examined in this study include the Raspberry Pi 5, 4, and 3, with the Raspberry Pi 5 representing the forefront of technology as of 2024. The final application was a graphical user interface (GUI) application, where multi-processing was employed to attain parallelism. This application was developed by Alex Henneman, a Ph.D. student, under the supervision of Dr. Luciano Ost [3] at Loughborough University. The aim here was to refine this application to minimize its execution time and enhance its scalability.

The first objective of this project is to improve the `mpbenchmark` multi-core benchmark tool, which simulates jet engine thermodynamics to evaluate CPU performance [4]. `mpbenchmark` generates two key outputs: runtime and the number of deadlines missed, the latter indicating performance quality based on thread completion times. This tool builds upon its predecessor, `JetBench`, which was originally developed in `C` language using the `OpenMP` library [5]. The transition to `mpbenchmark` addresses `JetBench`'s limitations and extends support to other languages like `Ada`, `C#`, and `Java`, with `C` and `Ada` showing the best performance. However, these versions face challenges such as `C`'s lack of object-orientated programming features and manual memory management issues [6], and `Ada`'s limited use outside defense and aerospace industries [7]. To overcome these, the project aims to develop `mpbenchmark` using `C++`, leveraging modern enhancements which were introduced in the language since 2011 [8].

The secondary objective of this project centres on enhancing the performance of `MobileNet`, a specialised machine learning algorithm within the realm of convolutional neural networks (`CNNs`) [9]. `MobileNet` excels in the field of image classification, a process in which the algorithm is provided with an image and tasked with determining the image's corresponding class. While the intricate mathematical foundations and specific operational details of `MobileNet` are beyond the scope of this discussion, the primary focus is set on augmenting its efficiency. Originally designed as a sequential algorithm, `MobileNet` primarily

operates on a single CPU core, which poses limitations on its execution speed and scalability. In pursuit of addressing these constraints, this project endeavours to transform the existing C++ implementation of MobileNet [10] into a multi-threaded architecture. By doing so, the aim is to significantly decrease its runtime and enhance its scalability, thereby optimising the algorithm's performance for more robust and efficient image classification tasks. This effort to transition MobileNet from a sequential to a multi-threaded model represents a strategic move to leverage the capabilities of modern multi-core processors, ensuring a more responsive and efficient computational process.

The project's third objective is dedicated to the optimisation of a graphical user interface (GUI) application designed for microcontroller (MCU) testing. The project will focus on minimising application runtime and bolstering its scalability. Named the DeBaTE-FI platform [3], this application is crafted in Python and leverages both multi-threading and multi-processing techniques to achieve parallelism and concurrency. It's crucial to distinguish between concurrency and parallelism; in a concurrent framework, tasks are initiated, executed, and completed in overlapping intervals, without necessitating simultaneous operation. The operational flow of the DeBaTE-FI platform involves the user connecting MCUs to a computer, after which the application dispatches specific commands to these MCUs. Following the transmission of commands, the platform awaits responses from the MCUs and proceeds to log the requisite data into output files. Among the three applications addressed in this project, the DeBaTE-FI platform emerges as the most intricate, a complexity attributable to both the scale and the sophisticated nature of its functions. This optimisation effort aims to streamline these extensive operations, enhancing the application's performance and user experience.

The report is structured as follows: it begins with a literature review that contextualises the research within the field by examining relevant studies. This is followed by the methodology section, which details strategies for identifying and addressing performance bottlenecks in the software. The software design is illustrated through Unified Modelling Language (UML) diagrams and some necessary code snippets. The "results and discussion" sections then present and analyse the performance enhancements observed in experiments conducted on desktop and Raspberry Pi models 3 through 5, with benchmark runtime and speedup plots quantifying the improvements. The discussion evaluates the optimisations' effectiveness and suggests future research directions, linking practical outcomes with theoretical implications. Appendices follow the conclusion, providing supplementary materials such as experimental setup details, system specifications, and extended code snippets to enhance understanding and ensure the report's transparency and reproducibility.

1.1 Original contributions of this project

1. A novel multi-core benchmark application has been developed using the latest `C++` features, this application built upon the previous `mpbenchmark` [4] application. It outperformed the previous `mpbenchmark` implementations in `C` and `Ada` by 11%, 4% and 1% on desktop, Raspberry Pi 5 and 4 respectively in terms of run time and provided better speedup across threads. Moreover the novel benchmark application also utilised SIMD intrinsics which provided up to an astonishing 70% reduction in run time, this unprecedented improvement provided means of analysing the target CPU's SIMD operations ability. A publication describing the contribution of this project has been submitted to SBCCI(Symposium on Integrated Circuits and System Design) conference part of IEEE(Institute of Electrical and Electronics Engineers) organisation.
2. The development of the first multi-threaded `MobileNet` CNN(convolutional neural network) model. This enhanced CNN model has been used as a case study to assess the behaviour of multi-threaded applications running on multi-core processors under neutron radiation. A publication describing this CNN model has been submitted to IEEE RADECS("RADiation Effects on Components and Systems") conference.
3. The optimised redesign of the `DeBaTE-FI` platform [3] resulted in up to 60% reduction in run time. Moreover a high performance `C++` library was developed for `telnet` communication with microcontrollers. The enhanced `DeBaTE-FI` platform produced a 43.5% reduction in run time when testing 36 `STM32F767ZI` microcontrollers when tested in PhD research office in Loughborough University(Figure 3.8).

Literature review

2.1 Introduction

The research topic focuses on parallel computing to optimise algorithms for both desktop and embedded CPUs. Parallel computing is widely regarded as a complex area within engineering and computer science. To develop or enhance multi-threaded solutions, a thorough understanding of the underlying algorithms and the specific goals of the application is crucial. This foundational knowledge is essential before attempting any modifications to ensure that changes do not deviate from the application's original objectives. In our project, we concentrate on three applications: `mpbenchmark`, `MobileNet`, and `DeBaTE-FI` platform. It is imperative to understand the background and functionality of these applications by utilising the existing publications centred around them.

Another goal of the literature review is to identify existing solutions to avoid duplicating efforts. Adherence to programming best practices is also a critical aspect of the project, given the complexity of parallel computing. This project strives to ensure that all developed software solutions conform to the best practices and guidelines recommended by industry experts, necessitating a comprehensive literature review.

The literature review is structured to support the report's overall organization, with individual subsections dedicated to the three main objectives, the review covers providing background information, reviewing current research, and offering critical analysis. The review then explores build systems, focusing on how this project aims to develop cross-platform applications that can operate on different operating systems such as `Linux`, `Windows`, or `macOS`. Further research is directed towards software design and multi-threading in line with modern `C++` guidelines, including discussions on relevant libraries, tools, and research on parallel algorithms. The conclusion section summarizes the findings from the preceding sections and underscores the significance of the reviewed literature to the research question.

2.2 Objective 1: `mpbenchmark`

Our project initially focuses on the `JetBench` software, featured in a paper from the ARCS conference series, a prominent event in Germany with over 30 years of history in computer architecture and operating systems research. The paper, "JetBench: An Open Source Real-Time Multiprocessor Benchmark" by Muhammad Yasir Qadri, Dorian Matichard, and Klaus D. McDonald-Maier, was published in a 2010 conference book [5]. It is pivotal for our research as it introduces a benchmark tool that evaluates real-time multiprocessor performance, aligning with our project objectives.

`JetBench` responds to the lack of real-time, multi-threaded benchmarks by simulating thermodynamic jet engine calculations. It primarily tests multi-core CPU capabilities through ALU-centric operations, including arithmetic and mathematical function computations, with an 88.6% parallel operation rate. Developed in `C` and utilising `OpenMP` for multi-threading, `JetBench` emphasizes portability and scalability across different computing systems without relying on system-specific libraries.

The benchmark's limitations include its narrow computation scope to ensure compatibility with lower-end CPUs and the exclusion of I/O operations to boost portability. Despite these limitations, `JetBench`'s

design effectively evaluates multi-core performance across diverse systems, making it an excellent tool for our project aimed at enhancing multi-threading on both desktop and embedded platforms.

To address the identified limitations of the `JetBench` software, a novel solution was devised using diverse programming languages. This innovation was detailed in a publication titled "Using JetBench to Evaluate the Efficiency of Multiprocessor Support for Parallel Processing," authored by HaiTao Mei and Andy Wellings. The paper, released in 2014, was part of the proceedings of a notable conference [4].

In their research, the authors of `mpbenchmark` [4] adopted an object-oriented programming (OOP) approach for `C#` and `Java` implementations. They standardised result collection by executing the application 30 times with a varying number of threads, employing a specific Linux command to control CPU core usage. Results were collected using a desktop CPU and `Simics`(a virtual platform to simulate a high end 128-core CPU). This methodology is in line with standard practices for benchmark data collection.

The authors of `mpbenchmark` [4] highlighted several design flaws in the original `JetBench` software [5], including the excessive use of shared variables leading to inaccurate performance results, race conditions from variables shared across threads, and erroneous benchmark output data printing. To remedy these issues, they restructured the `JetBench` code and introduced implementations in `Ada`, `C#`, and `Java`. Notably, the compiled languages (`C` and `Ada`) demonstrated superior performance. Additionally, the impact of virtual cores enabled by simultaneous multi-threading (SMT) was observed to vary inconsistently across different programming languages.

In the second paper [4] several limitations were found. Firstly, the data collection, based on just 30 runs, may be insufficient, especially for applications with minimal execution times on high-end CPUs. Additionally, the research focused solely on desktop CPUs and a system simulator mimicking a high-end 128-core CPU, excluding experiments on embedded or lower-end CPUs. Furthermore, given the superior performance of compiled languages like `C` and `Ada`, further research could beneficially explore comparisons with `C++`, another compiled language often regarded as superior to both `C` and `Ada` in certain contexts. This paper lays a solid foundation for our project, where its limitations present opportunities for further investigation, and its methodological approaches offer a model for emulation.

2.3 Objective 2: `MobileNet`

`MobileNet` is a specialized machine learning algorithm that falls under the category of convolutional neural networks (CNNs). Developed specifically for mobile and embedded vision applications, `MobileNet` is discussed in detail in a 2017 paper by Cornell University titled "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications" [9].

This paper [9] delves into the mathematical foundations required to construct the `MobileNet`, which are considered beyond the scope of our project. To summarise, the `MobileNet` architecture employs depth-wise separable convolutions that enhance memory efficiency and leverage optimised numerical linear algebra algorithms to reduce latency.

`MobileNet` introduces two crucial hyper-parameters: the width and resolution multiplier. These allow for adjustments to the model's size and computational requirements, with smaller `MobileNet` models typically showing slightly reduced accuracy compared to their larger counterparts. In comparative analyses with other renowned models like `GoogleNet`, `VGG 16`, and `FaceNet`; `MobileNet` offers comparable accuracy levels.

Despite its advantages of reduced size and latency, making it ideal for mobile and embedded devices,

the paper [9] does not address the use of parallel computing techniques, presenting an opportunity for further exploration of its source code to enhance performance through parallelism.

For the practical component of our project, a C++ implementation of MobileNet was chosen, sourced from a public repository on the GitHub [10]. This particular implementation was part of a computer science competition [11], challenging participants to classify images into 20 predefined classes using provided training, validation, and testing data, with submissions made in a .zip file format.

The C++ implementation comprises several header and source files but lacks build software or instructions, which poses a significant initial hurdle. Moreover, the implementation features a method for testing custom images against 12 predefined classes. However, these class names are in Chinese, necessitating translation into English for broader accessibility. Setting up and running the project before making any enhancements would be challenging. Nevertheless, this C++ implementation presents a valuable opportunity to apply parallelisation techniques to improve the efficiency of the widely recognized MobileNet algorithm.

2.4 Objective 3: DeBaTE-FI platform

The DeBaTE-FI platform, developed by PhD student Alex Henneman under the supervision of Dr. Luciano Ost at Loughborough University, is a GUI application detailed in the publication "DeBaTE-FI: A Debugger-Based Fault Injector Infrastructure for IoT Soft Error Reliability Assessment" [3].

DeBaTE-FI was engineered to test embedded devices for soft errors—random errors in processors caused by background radiation, which can be particularly harmful in safety-critical systems. Testing methods for soft errors range from high-level analytical models, which are easier to implement but less accurate, to costly full radiation tests, which offer higher precision. DeBaTE-FI seeks to emulate the latter's accuracy by simulating soft errors in embedded devices. The platform utilized Raspberry Pi 4 and Rock 4C to conduct tests on STM32F7 microcontrollers (MCUs). The research involved running a neural network binary on these MCUs, measuring the execution time across various testing scenarios, and using different numbers of MCUs to evaluate speed and efficiency. Additionally, the results from DeBaTE-FI were compared with those from simulation software to assess accuracy, and OpenOCD was used for MCU communication.

The authors observed that increasing the number of MCUs reduced overall runtime. They also discovered that while multithreading in Python did not improve performance, multiprocessing—which enables parallel processing—did enhance performance significantly. The study confirmed that DeBaTE-FI provides greater accuracy than the high-level simulation software they aimed to surpass. Challenges arose when using Raspberry Pi 4, particularly with USB controller limitations that restricted the connection to no more than 8 MCUs, prompting a switch to Rock 4C, which supported up to 30 MCUs but also reached full CPU utilization under this load.

Despite the innovative approach of DeBaTE-FI, the platform is hindered by several significant limitations. The primary issue is its lengthy runtime, which can extend to several minutes, suggesting the presence of performance bottlenecks. Moreover, the performance on embedded Linux devices, such as Rock 4C, is modest, highlighting the need for optimization across both desktop and embedded Linux devices (or single-board computers). Additionally, the utilization of OpenOCD for MCU communication presents an opportunity to enhance efficiency and expedite processes. The central goal of our project is to tackle these issues by improving the runtime efficiency and scalability of DeBaTE-FI.

2.5 Cross platform build systems

A 2008 paper published by Technical University of Munich titled "A Cross Platform Development Workflow for C/C++ Application" [12] underscores the efficacy of CMake in addressing significant challenges in cross-platform project management, particularly those stemming from the constraints of build systems and source code management practices. This is highlighted despite the standardization of C and C++ by ANSI and ISO, and the availability of the C library and STL. Given our project's focus on constructing C++ applications and the need to automate build processes across multiple operating systems such as Linux, macOS, and Windows, CMake emerges as an suitable choice.

The authors highlight CMake's ability to maintain developers' preferred environments across different platforms, reducing the need to synchronize platform-specific project files. By implementing CMake in projects like KDE4, an OpenGL application, and a platform-independent camera interface, the paper advocates for a simplified cross-platform development workflow.

CMake supports the configuration step vital for locating external package headers and libraries, offering numerous pre-written scripts and allowing developers to create tailored scripts for specific needs. It also facilitates the development of qiew, a platform-independent VRML viewer using the Qt toolkit and Coin3D, exemplifying consistent cross-platform user experiences.

Compared to tools like GNU Autotools, CMake has a gentler learning curve, enhancing accessibility and reducing the need for extensive training. Its effectiveness is illustrated at Technische Universität München, where it bridges diverse development environments in vision and robotics research, also adopted by two spin-offs. The tool aids in project generation, synchronization, configuration, dependency resolution, and deployment packaging, as shown in its use in KDE4 and OpenSceneGraph.

While outlining CMake's benefits, the paper lacks a discussion on its potential limitations, such as configuration overheads and handling complex scenarios. Including tool comparisons and considering technology's rapid evolution could provide a more balanced view of CMake's current relevance in modern software development landscapes.

Addressing the previous paper's limitations and potentially out of date content more research was conducted to find more recent publications around the use of build automation software. A recognised publication titled "Large Scale Software Building with CMake in ATLAS" obtained from the "Journal of Physics: Conference Series" published in 2017 [13] discusses the use of build automation software for C++ application in an industrial context.

The text discusses CMake's application in managing complex, multi-platform software projects like ATLAS. It uses a master CMakeLists.txt file and subdirectories reflecting the CMT package structure for streamlined project management. This approach includes Atlas-specific functions, add_subdirectory for package inclusion, and helper targets for building and testing. It ensures modularity with prefixed libraries and enhances flexibility with scripts for runtime environments and relocatable project files.

Key findings from migrating ATLAS's offline software to CMake highlight improved build efficiency and streamlined processes. The integration of CMake has optimised resource use on build machines and, alongside consolidating projects into a single repository named Athena, reduced build times to about three hours. The transition to CMake and Git aligns ATLAS with standard software development practices and is expected to support LHC's Run 3 and beyond.

The shift to CMake is critically evaluated for its strengths in build efficiency and community support, and weaknesses in dependency management and initial build time improvements. While it modernizes ATLAS's

build processes, ongoing refinement is essential to maximize CMake's benefits and ensure long-term scalability.

Conducting research on build automation software for C++ projects [12] [13] outline the CMake software as a useful tool that is industry standard and can be used for this project.

2.6 C++ guidelines and best practices

Multi-threading in C++ is a complex topic therefore in order to adhere to C++ core guidelines, textbook titled "C++ concurrency in Action" written by Anthony William was researched [14]. This book is regarded by many as the gold standard for multi-threading concepts in modern C++, as discussed by the community on stack overflow [15]. The second edition of this book has been updated to include features the C++17 standard.

The book [14] thoroughly examines the methodologies for developing concurrent and multithreaded applications with C++. This edition not only reviews the foundational principles from C++11 and C++14, but also incorporates advancements from C++17, providing an extensive overview of modern C++ concurrency features. Williams methodically explores various strategies, including lock-based and lock-free programming, as well as the development of concurrent data structures and algorithms. Each technique is clearly explained, focusing on practical implementation and the effective use of the language's concurrency capabilities. The book also addresses the challenges of concurrency with detailed examples and solutions, making it an essential resource for developers aiming to utilize the full potential of multicore processing with C++.

Strengths of the book [14] include comprehensive explanations and practical examples that are accessible even to programmers with a basic understanding of C++, making complex concurrency concepts easier to grasp. Williams meticulously explores both fundamental and advanced concurrency techniques, accompanied by a methodical presentation of best practices and common pitfalls, which helps developers build robust, high-performance applications. However, the book's weaknesses include its dense technical content, potentially overwhelming for beginners without a solid C++ background or those new to concurrency. While it thoroughly addresses C++17, those seeking insights into the latest concurrency features of C++20 might find the coverage lacking, highlighting a gap in an otherwise excellent resource. This gap points to an area for potential future updates or editions as C++ continues to evolve. Overall, the book remains an invaluable resource for both experienced and intermediate C++ developers aiming to deepen their understanding of concurrent programming.

Another valuable study, titled "Multi-Threaded Parallel I/O for OpenMP Applications," was published in the International Journal of Parallel Programming in 2014 [16]. This research is pertinent to our project as various libraries exhibit unique strengths and weaknesses, and selecting the appropriate library is crucial for achieving optimal performance.

The study focuses on optimizing multi-threaded parallel I/O operations within the OpenMP framework using a parallel I/O library integrated into the OpenUH compiler. Techniques include `omp_file_read_all()` function for collective data reading, merging small I/O requests into larger ones, and splitting large requests among threads to boost efficiency. A contiguity analyser and a work manager optimize data handling and thread utilization, with adaptive techniques accommodating different file system behaviours, including the use of the Parallel Log-structured File System (PLFS) on systems like Lustre.

Findings highlight significant performance improvements in I/O operations, especially in write operations,

achieved by enabling multiple user-level threads to operate concurrently. Write speeds approach those of separate file writes per thread, surpassing the one-file-per-thread method without needing a merging step. However, scalability for read operations is limited, showing benefits only up to four threads. The integration of OpenMP I/O with MPI/OpenMP hybrid applications is limited by MPI domain partitioning, though OpenMP I/O could enhance MPI I/O systems like ROMIO and OMPIO.

The paper [16] evaluates parallel I/O interfaces for OpenMP, demonstrating enhanced performance over traditional sequential I/O and discussing interface specifications and optimizations across various file systems and benchmarks. Future research will refine these interfaces for recent OpenMP developments and enhance the parallel I/O library for NUMA architectures, integrating multiple storage resources for broader application in diverse computing environments.

The project aims to emulate the techniques and software design highlighted in the book [14] to ensure the produced software adheres to the best practices. Another library, OpenMP, which is widely used in high-performance computing and parallel I/O interfaces [16], offers a straightforward method for implementing multi-threading in both C and C++. This library is particularly useful for our project as it provides an alternative to the std::thread library in C++.

2.7 Literature review conclusion

Research on the JetBench software has been identified as essential, as documented in the mpbenchmark publication [4], which tracks its evolution and benchmarks performance across various programming languages. Analysis of both the mpbenchmark and JetBench papers [5] [4] reveals substantial advancements, providing a robust foundation for this project to develop a more optimised C++ solution. This initiative, guided by best practices from the renowned C++ Concurrency in Action [14]—a resource recommended on Stack Overflow [15], builds upon existing work without redundancy. This project stands out from the reviewed literature in two significant ways: firstly, it rewrites the mpbenchmark using modern C++; secondly, it introduces an alternative version that employs SIMD intrinsics to extract as much performance possible from the target CPU.

For the second objective, understanding of the MobileNet algorithm was enhanced through its foundational publication [9]. This project works with a C++ implementation of MobileNet, sourced from an open-source GitHub repository [10], to analyse and enhance the code through parallelisation. Key resources for this enhancement include the C++ concurrency textbook [14] and various multi-threading libraries [16], which are crucial for optimizing performance. The project distinguishes from the reviewed literature in two major aspects as it creates the first multi-threaded version of the MobileNet CNN model and analyses its performance across different processors.

The first two objectives necessitate the development of cross-platform C++ applications, for which CMake is suitable for this project. Investigations into CMake's application across multiple platforms have been informed by pertinent literature [12] [13]. This research is essential to streamline the build and compilation processes, ensuring they meet industry standards.

The third objective addresses a unique challenge: enhancing the DeBaTE-FI platform, developed by Loughborough University's PhD students and faculty [3]. A thorough review of its publication was critical to comprehend its design and functionality. The project will leverage the existing features of DeBaTE-FI, aiming to refine its application and benchmark against the original performance metrics. This project distinguishes itself from the reviewed literature regarding DeBaTE-FI platform in two major aspects: it aims to optimise the original software stack and introduce a new library for telnet communication.

Methodology

3.1 Objective 1: `mpbenchmark`

As discussed in the introduction, `mpbenchmark` performs calculations of a jet engine using multiple threads and produces the time taken to complete these calculations as output. The source code of `mpbenchmark` provided a solution in `C#` language [17], this served as a useful reference of its implementation in code using object-oriented design. Subsequently, the `C++` object oriented design comprised of three main classes:

1. `FileDataLoader`: the primary function of this class was to load data from the input file and also to allow the user to save output data to the output file.
2. `SharedPerformanceData`: this class stores data loaded from the input file into an array and also allows storage of output data into a separate array. But importantly it allows threads to access specific parts of the input data in a thread-safe manner.
3. `Worker`: this class contains functions to perform the important calculations, computations of deadlines missed and output data. This class defines the `operator ()` which encapsulates the main calculations, this class design is know as a `Functor`.

The `C++` object-oriented class design can be visualised using a UML class diagram shown in Figure 3.1.

Figure 3.1: UML class diagram of the proposed `C++` solution.

Figure 3.2 shows the UML sequence diagram on how the three classes in the previous Figure 3.1 are used. The "`main`" in the sequence diagram represents the `.cpp` file where the `int main(...)` function exists.

The key features of the proposed solution leverage the modern enhancements introduced to the `C++` language in its 2011 update. Since then, `C++` has undergone significant transformations that include advanced multi-threading capabilities and improved memory management techniques. Significantly,

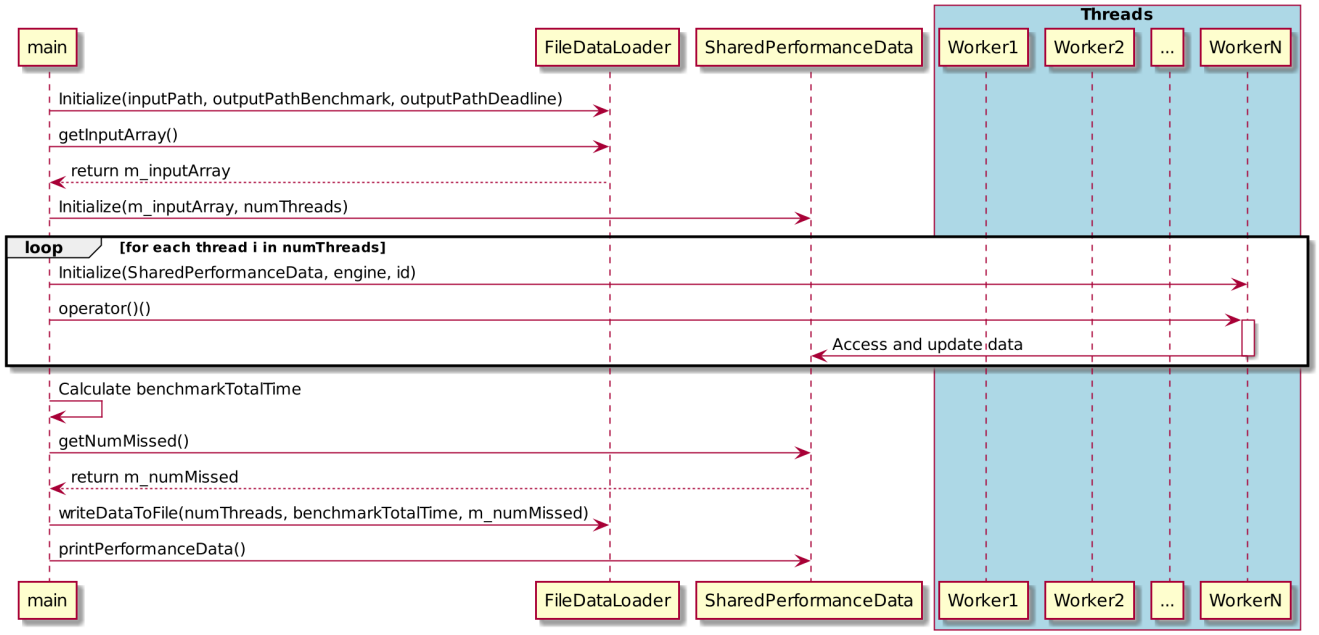


Figure 3.2: UML sequence diagram of the proposed C++ solution.

the introduction of `std::thread` and `std::mutex` in modern C++ offers a more convenient and cross-platform approach to multi-threading compared to traditional POSIX threads, which are not only more cumbersome to use but also restricted to Unix-based systems. The utilization of these features in the proposed solution enhances its portability and ease of use. Additionally, the proposed solution incorporates the `std::vector` class, a dynamic array sequence container that significantly simplifies memory management. Unlike in the C language, where manual handling of dynamic memory can be error-prone, `std::vector` automates this aspect, thereby reducing complexity and potential bugs. While modern C++ offers numerous advantages over C, including object-oriented programming features and performance parity, it does come with a steeper learning curve. This learning curve is often seen as a trade-off for the powerful features and robustness provided by C++ [8].

The proposed solution employed the `fmt` library in C++ [18] for output, providing an enhanced printing method compared to the `std::cout()` function from the C++ standard library. Although the compiler on the target system supported the C++20 standard, it lacked support for the `std::format()` function, which provides better printing capabilities in C++ [19]. Additionally, a new command line argument was introduced to adjust the number of threads used, complementing the existing argument that modifies the simulation engine. For compilation and linking, the industry-standard `CMake` [20] software was utilized. In the `CMakeLists.txt` file, essential configurations included specifying the C++20 standard, incorporating the `fmt` library, and compiling with the `-O2` optimization flag. This optimisation flag, also used by the original authors of `mpbenchmark` [4], was retained in the proposed solution for consistency.

An alternate solution was developed to further enhance the performance of `mpbenchmark`, the `Valgrind/Callgrind` function profiler tool was deployed to find potential bottlenecks. The application was compiled with debug information and optimisations turned off, this was done by specifying the build type as `Debug` and compiling with `-g` and `-O0` flags in the `CMake` file. `Callgrind` profiling shows "self-cost" of different parts of the code. "Self-cost" refers to the number of CPU cycles that were directly consumed by a specific function itself. This can be used to find potential areas of optimisations [21]. `Callgrind` profiling results showed that part of the application where it approximates the value of π

had the highest self-cost. This code snippet is shown in the Listing 3.1 below:

```
// initialise variables for the pi calculation
const long num_steps = 1000000;
double step = 1.0 / static_cast<double>(num_steps);
double x{}, pi{}, sum{};

// performing numerical integration using the midpoint Riemann sum
for (int i = 0; i < num_steps; i++) {
    x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
}
pi = sum * step;
```

Listing 3.1: Part of the code with the highest self-cost. It approximates π using numerical integration.

The problem with this part of the code was that it was already part of the parallel regions where it is executed by each thread. It may seen like a good candidate for applying parallel `for loop` from the OpenMP library however creating nested threads beyond the number of threads available on the system does not always lead to higher performance and in many cases can degrade performance. Another way to improve performance is by using SIMD intrinsics. An advantage of using `C++`(and/or `C`) is that SIMD intrinsics can be deployed whereas higher level languages like `Java` or `C#` make it very difficult to access these.

SIMD intrinsics vary by the target CPU, `x86` processors (which are found in most laptop and desktops) use `AVX2` instructions whereas `ARM` processors(commonly found in Apple products and embedded devices) use `NEON` instructions. In this project we use both as our proposed `C++` solution will be deployed on desktop(`x86` processor) and Raspberry Pi devices(`ARM` processor). The SIMD enhanced code can be summarised algorithmatically in the following steps in Figure 3.3.

1. Initialise `256-bit` wide vectors: each vector can hold four double-precision (`64-bit`) floating point numbers. The main initialisations would be a vector to hold four loop indices (`vec_i`), a vector to calculate four values of x (`vec_x`), a vector to hold the result of the integrand (`vec_temp`) and a vector to accumulate the sum (`vec_sum`) after each loop iteration.
2. `for loop` iterate until `num_steps/4`:
   - step 1: calculate the four midpoints x-values simultaneously using the vector `vec_i` and hold result in `vec_x`. Original formula used: `(i + 0.5) * step`.
   - step 2: compute the value of the integrand in parallel using the four calculated x-values stored in `vec_x`, store result in `vec_temp`. Original formula used: `4 / (1 + x * x)`.
   - step 3: accumulate the values from `vec_temp` into the `vec_sum` vector.
   - step 4: increment loop indices vector `vec_i` by `4`.
3. Final summation: upon the completion of the loop, perform a horizontal sum on the vector (`vec_sum`) that held the accumulated values.
4. Calculation of π: sum is multiplied by the step size to approximate the value of π. Original formula used : `pi = sum * step`.

Figure 3.3: Algorithm for calculating π using SIMD(`AVX2`) instructions.

As discussed, to utilise SIMD intrinsics on Raspberry Pi devices, `NEON` instructions must be employed. `NEON` instructions come with limitations, notably in their support for double precision floating points, which is restricted, and their vector width, which is only `128-bit`, compared to the `256-bit` vectors seen in `AVX2` [22]. To accommodate `NEON`, two solutions were developed: one using single precision

floating points (`float`) and the other using double precision floating points (`double`). The `NEON` code with `float` can perform four computations simultaneously, while the code with `double` can manage only two computations simultaneously, due to the `128-bit` vector's capacity to store four `float` values or two `double` values. Typically, `float` variables offer less decimal precision than `double` variables. These two SIMD-enhanced solutions, along with their decimal accuracy levels, will be compared in the results and discussion section. The `NEON` implementation follows a similar algorithm as the one shown in Figure 3.3. The `CMake` file also required some changes to allow the `AVX2` instructions to be used, this along with the code snippets for the SIMD enhanced code can be found in the [Appendix 2].

To collect benchmark data, the runtime was recorded and saved to a specified `.txt` file, the number of deadlines missed were also saved but in a separate `.txt` file. The application underwent 103 runs, with the first three designated as warm-up runs; the subsequent 100 runs were averaged and utilised for plotting bar charts and speedup plots. For the languages `Java` and `C#`, data were collected over 103 runs. In contrast, for the compiled languages `C`, `Ada`, and `C++`, the application was executed 203 times. The variance in the number of threads was controlled using the `taskset` command in Linux. Specifically for the `C++` application, the second command line argument was employed to specify the number of threads. This approach aligns with the methodology described by the authors of `mpbenchmark` [4]. An example bash script demonstrating how the application was executed multiple times is shown in the following Listing 3.2:

```
# Loop to run the executable 103 times, using "3" as the default engine type and 2 cores/threads
for i in {1..103}
do
    taskset -c 0,2 ./mpbenchmark 3 2
done

# Loop to run the executable 103 times, using "3" as the default engine type and 4 cores/threads
for i in {1..103}
do
    taskset -c 0,2,4,6 ./mpbenchmark 3 4
done
```

Listing 3.2: Bash script to run application multiple times along with `taskset` command. Command line arguments: `mpbenchmark [engine_type] [threads]`.

The process of collecting benchmark times can be visualised in the image shown in Figure 3.4. The python application used to plot the graphs was trivial and not shared in this report. The full system specification of the desktop and Raspberry Pi devices used to collect data can be found in the [Appendix 1]. The results are displayed via benchmark and speedup plots. A benchmark plot shows the mean run time of the application on the vertical axis with the number of cores or threads on the horizontal axis, in this case the lower is better as a lower run time indicates better performance. The error bars indicate 95% confidence interval of the mean. A speedup plot shows the speedup ratio on the vertical axis and the number of cores or threads on the horizontal axis, here higher values are better because a higher speedup indicates better scalability of the application. The "speedup" is a ratio of the time taken for the application using a certain number of threads compared to the original single-threaded run time. For example, if an application takes 100 ms to run using 1 thread and 20 ms to run using 8 threads. The speedup for 8 threads would be $\frac{100}{20}$ equalling `5.00`, as a ratio it would be unitless.

In summary, for desktop (`x86`) processors, two main solutions have been developed: a novel `C++` solution and a SIMD-enhanced `C++` solution as discussed in Listing 3.3. For the Raspberry Pi devices, three solutions

have been created: the first is the novel C++ solution, identical to that on the desktop, and the other two are the SIMD-enhanced versions utilizing NEON instructions with single and double precision floating point variables.

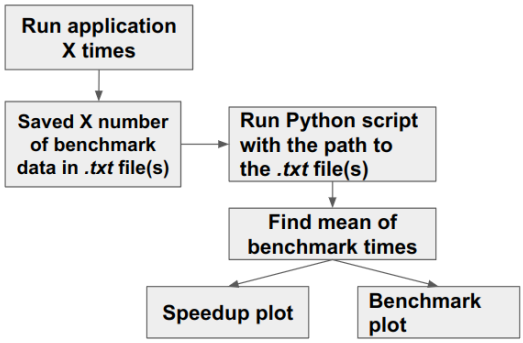


Figure 3.4: Benchmark results collection strategy.

3.2 Objective 2: MobileNet

Setting up MobileNet from the GitHub repository [10] was a time consuming task. Details about the setting up process can be found in the [Appendix Listing 6]. Once setup, the MobileNet project needed to be profiled with Callgrind function profiler to find bottlenecks and possible parallelisation points. To use Callgrind, the project was compiled with debug information and no optimisations. This required setting the build type as Debug and adding -g and -O0 optimisation flags in the CMakeLists.txt file. Callgrind results showed that functions from the convolution and batch normalisation layer had the highest self-cost. The functions with the highest self-cost were ConvLayer::forward(), BatchNormalLayer::forward() and ConvLayer::Addpad(). The detailed Callgrind profiling results can be seen in [Appendix Figure 4].

The CPU intensive function inside the convolution layer contained a number of nested for loops with large iterations. OpenMP library was used to parallelise the for loops. The collapse clause and dynamical scheduling type of the parallel for loop from the OpenMP library were implemented inside the ConvLayer::forward() function. In the OpenMP library, the parallel for directive, enhanced by the collapse(3) clause and dynamic scheduling, efficiently manages the execution of nested loops in parallel. By collapsing three nested loops into a single sequence, the collapse clause simplifies load balancing across multiple threads by increasing the pool of iterations available for distribution. Concurrently, the dynamic scheduling type assigns iterations to threads in real-time, adapting to varying execution times among iterations and ensuring an even workload distribution [23].

To parallelise the other two functions BatchNormalLayer::forward() and ConvLayer::Addpad() only the collapse clause was utilised. The ConvLayer::forward() function contained some computations which were a good candidate for SIMD optimisations, using SIMD was more straight forward than compared to objective 1 as OpenMP library allows a simple way of using SIMD without the developer having to worry about AVX2 or NEON instructions. For the Raspberry Pi processor it was decided to parallelise only the ConvLayer::forward(), BatchNormalLayer::forward() functions and not use the SIMD enhancements, as SIMD instruction's performance gains are more limited on embedded processors due to the narrower registers [24]. Having said that results from parallelising all the three functions and SIMD enhanced code were compared to find the optimal solution. This is discussed in the results and discussion section.

To address the different compilations, the `CMake` file was altered. A macro `EMBEDDED_PROC` was added. To compile the `MobileNet` project for `x86` processors the macro `EMBEDDED_PROC` was set to `OFF`, this applies parallelisation to all three functions along with the SIMD enhancements. To compile the `MobileNet` project for Raspberry Pi processor the user can specify the `EMBEDDED_PROC` to `ON`, this would parallelise only the two functions discussed and not apply SIMD enhancements. This can be seen in lines 15-18 in Listing 3.3. The parallelisation of the other two functions `BatchNormalLayer::forward()` and `ConvLayer::Addpad()` were very similar to `ConvLayer::forward` and can be found in [Appendix Listing 8 and 9].

```
1  void ConvLayer::forward(float *pfInput)
2  {
3  // collapse 3 "for" loops and use dynamic scheduling
4  #pragma omp parallel for collapse(3) schedule(dynamic)
5  for (int g = 0; g < m_nGroup; g++)
6  {
7      for (int nOutmapIndex = 0; nOutmapIndex < m_nOutputGroupNum; nOutmapIndex++)
8      {
9          for (int i = 0; i < m_nOutputWidth; i++)
10         {
11
12 // other code to be ignored ...
13
14 // Only use OpenMP SIMD optimizations if EMBEDDED_PROC is not defined
15             #ifndef EMBEDDED_PROC
16             #pragma omp simd reduction(+:fSum)
17             #endif
18             for (int n = 0; n < m_nKernelWidth; n++)
19             {
20                 nKernelIndex = nKernelStart + m * m_nKernelWidth + n;
21                 nInputIndex = nInputIndexStart + m * m_nInputPadWidth + n;
22                 fSum += m_pfInputPad[nInputIndex] * m_pfWeight[nKernelIndex];
23             }
24         }
25     }
26 }
27
28 }
```

Listing 3.3: Parallelising the `ConvLayer::forward()` function and applying SIMD depending on the macro `EMBEDDED_PROC`.

In line 16 of the code snippet above(Listing 3.3) the reduction clause on the `fSum` was deployed, this ensured that after vectorized operations, the partial results held in different elements of the SIMD vector register are correctly summed up into the single scalar `fSum`. This avoids any manual management of these partial results, simplifying the code and ensuring correctness.

Moreover the arrays must also be “aligned” to make full use of the SIMD clause, this is important in several key ways. First, it ensures efficient utilisation of the CPU’s cache lines, reducing cache misses by aligning data accesses with the processor’s cache system, thereby accelerating data retrieval. Second, it facilitates efficient SIMD operations, as modern SIMD instructions require data to be aligned with the size of their registers—here, 256 bits or 32 bytes—to prevent penalties from misalignment, such as additional cycles for data realignment. Alignment also prevents faults that occur when data is accessed at addresses not aligned to the required boundaries, particularly on systems where such misalignment leads to crashes. Lastly, by aligning memory accesses with the hardware’s memory interface, the code

optimises the throughput and maximizes the memory bandwidth usage. Together, these factors ensure that the SIMD-optimised processes in the convolutional layer computations are performed with maximal efficiency and stability [25]. A 32-byte alignment was used for arrays `m_pfInputPad` and `m_pfWeight`, this is shown in Listing 3.4.

```
ConvLayer::ConvLayer(/*Constructor arguments not shown ... */)
{
// ignore other code ...

// Creating arrays which are 32-byte aligned for SIMD optimisations
size_t alignment  = 32;
size_t inputPadSize = m_nInputNum * m_nInputPadWidth * m_nInputPadWidth * sizeof(
    float);
size_t weightSize   = m_nOutputNum * m_nInputGroupNum * m_nKernelSize * sizeof(
    float);
size_t outputSize   = m_nOutputNum * m_nOutputSize * sizeof(float);

m_pfInputPad = static_cast<float*>(aligned_alloc(alignment, inputPadSize));
m_pfWeight   = static_cast<float*>(aligned_alloc(alignment, weightSize));
}
```

Listing 3.4: Making arrays that are 32-byte aligned for SIMD clause.

The `MobileNet` application was designed with three command line arguments. The first argument determines whether to classify only one image or all images in the `data` folder. The second argument specifies whether the runtime should be saved to a `.txt` file, and the third sets the number of threads used. For data collection, the time taken to classify a single image was recorded while varying the number of threads. The number of threads utilized by the application was adjusted using the command line and `OpenMP`'s `omp_set_num_threads()` function. For benchmarking, the project was compiled with the `-O3` optimisation flag and was executed 103 times with different thread counts; the runtime and speedup plots were derived from the average of the 100 timing results stored in the `.txt` file. The method of data collection largely mirrors the process shown in Figure 3.4, with the exception that the `taskset` command was not employed to adjust the number of CPU cores; instead, thread adjustments were made through the command line. The detailed code snippet of the command line arguments can be found in the [Appendix Listing 7].

3.3 Objective 3: `DeBaTE-FI` platform

The application was profiled to find bottlenecks and possible points of optimisations. The tool `py-spy` was used to profile the application and save the output as a `.json` file [26]. The results were then viewed using the `speedscope` web application [27]. Since the application used multi-processing, the required processes were identified and their process `pid` was used to run `py-spy`, as shown below in Listing 3.5:

```
    py-spy record --pid <PID> --format speedscope -o profile.json
```

Listing 3.5: Running `py-spy` on the specified process.

Profiling results showed that the functions from `Python`'s `Telnetlib` library had the highest self-cost. To optimise the application, a `C++` open-source `Telnet` library was utilized. The `C++` library, called `telnetpp`, [28] along with `serverpp`, [29] was integrated into the application using the tool `pybind11`. `pybind11` is a lightweight library for integrating `C++` into `Python` applications [30]. The `telnetpp` library required some functions for it to be integrated into the application; thus, a wrapper class in `C++` was

created that allowed `telnetpp` to seamlessly replace the previous `telnetlib` `Python` library. The wrapper `C++` class was named `telnetlibcpp` for consistency. This `C++` class is shown in the following UML class diagram (Figure 3.5). The main functions used, `Readout()`, `Exec()`, and `write()`, have the exact same names as the functions from the `Python` library. Profiling results and detailed code snippet about the `C++` functions can be found in [Appendix 4].

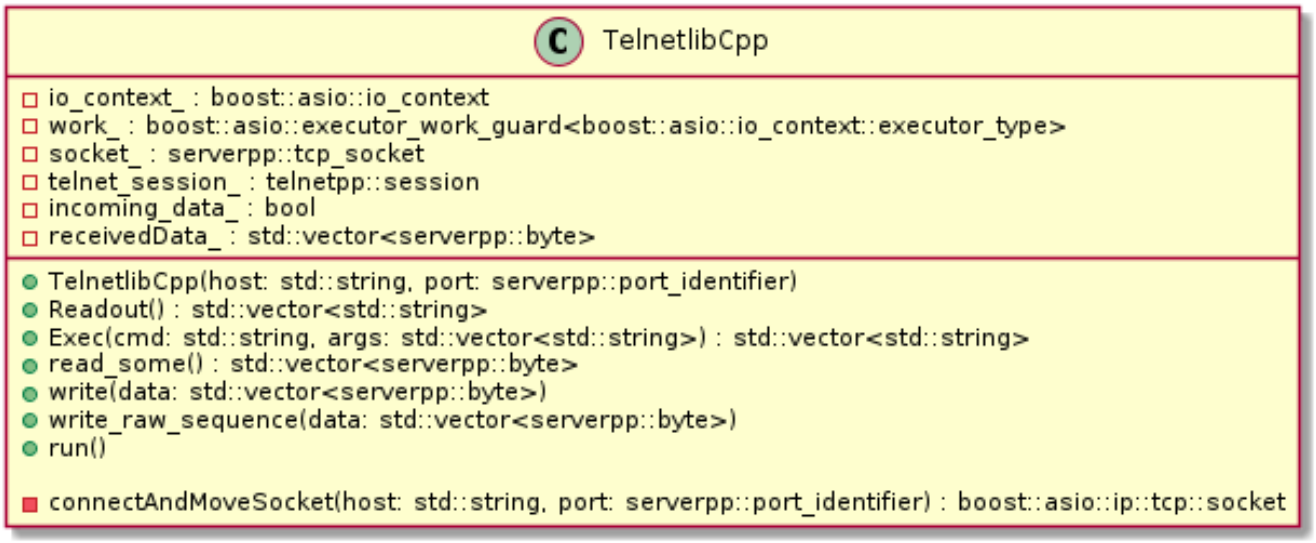


Figure 3.5: `C++` class to emulate `Python's` `telnetlib` library functions.

Using `CMake` and `pybind11` the `C++` class was compiled into a shared library(`.so`) file which was then moved into the same directory as the application. This allowed the `python` application to call the `C++` functions and use the `telnetpp` `C++` library.

Another solution was developed which aimed to optimise the application's multi-processing and multi-threading. A simplified version of the application's architecture is shown below in the UML sequence diagram(Figure 3.6):

Figure 3.6 summarises how the original application had a convoluted design leading to poor performance:

1. Two classes `device` and `deviceControl` spawned processes.
2. The `deviceControl` process had its workload split between sending commands to MCU and saving serial port data to `.txt` file.
3. The `device` process seem to be performing a redundant task of saving serial port data into a queue.

This design was improved as follows:

1. Two classes `device` and `deviceControl` spawn processes with each having their own designated tasks.
2. The `deviceControl` process now only sends data to the MCU.
3. The `device` process only focuses on receiving data from the serial port and saving it to the `.txt` file.

The key in the above improved design is removing the redundant work as previously both of the processes were handling the serial port data. Handling serial port data does not require two processes it can be

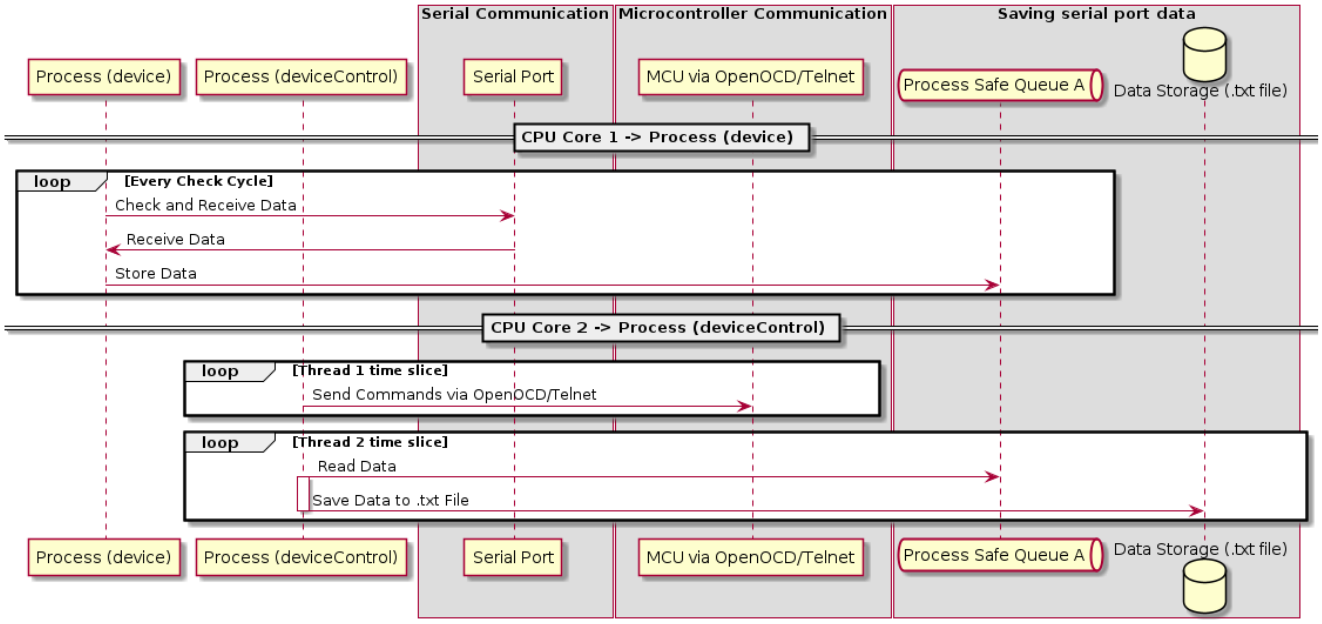


Figure 3.6: The sequence of the original application architecture. CPU Core 2 has to split CPU time between handling serial port data and sending commands.

done using only one. When threads are used inside a process, CPU time is split between the two threads. In the improved design both processes only perform one task and more importantly the `deviceControl` process has a reduced workload, this is important as the CPU time will be fully allocated to sending commands to the MCU . The challenge here was to make sure both of the processes were synchronised. This was accomplished using the `Event()` object from the `multiprocessing` `Python` library. When the `deviceControl` process started it sent a signal to the `device` process, to start checking and storing serial port data, and when the `deviceControl` process finished it sent a signal to stop the `device` process, stopping the `device` process would then save the accumulated serial port data into the `.txt` file. This can be visualised in the following UML sequence diagram (Figure 3.7):

Results were collected using four `STM32F767ZI` MCUs for both of these solutions. In collaboration with Alex Henneman, results using the main setup(see Figure 3.8) that utilised 36 MCUs running the second solution were also collected to verify the performance observed on the local setup. To collect benchmark data the application was run only once with varying number of boards and the run time was saved into a `.csv` file. The run time usually spanned in minutes thereby making multiple runs extremely time consuming, moreover the variation in run times was negligible unlike the applications in the first two objectives where the run times were in milliseconds and were more prone to variation. The benchmark collection strategy was similar to the one shown in Figure 3.4 except the application was run only once and the times were saved in a `.csv` file instead of a `.txt` file.

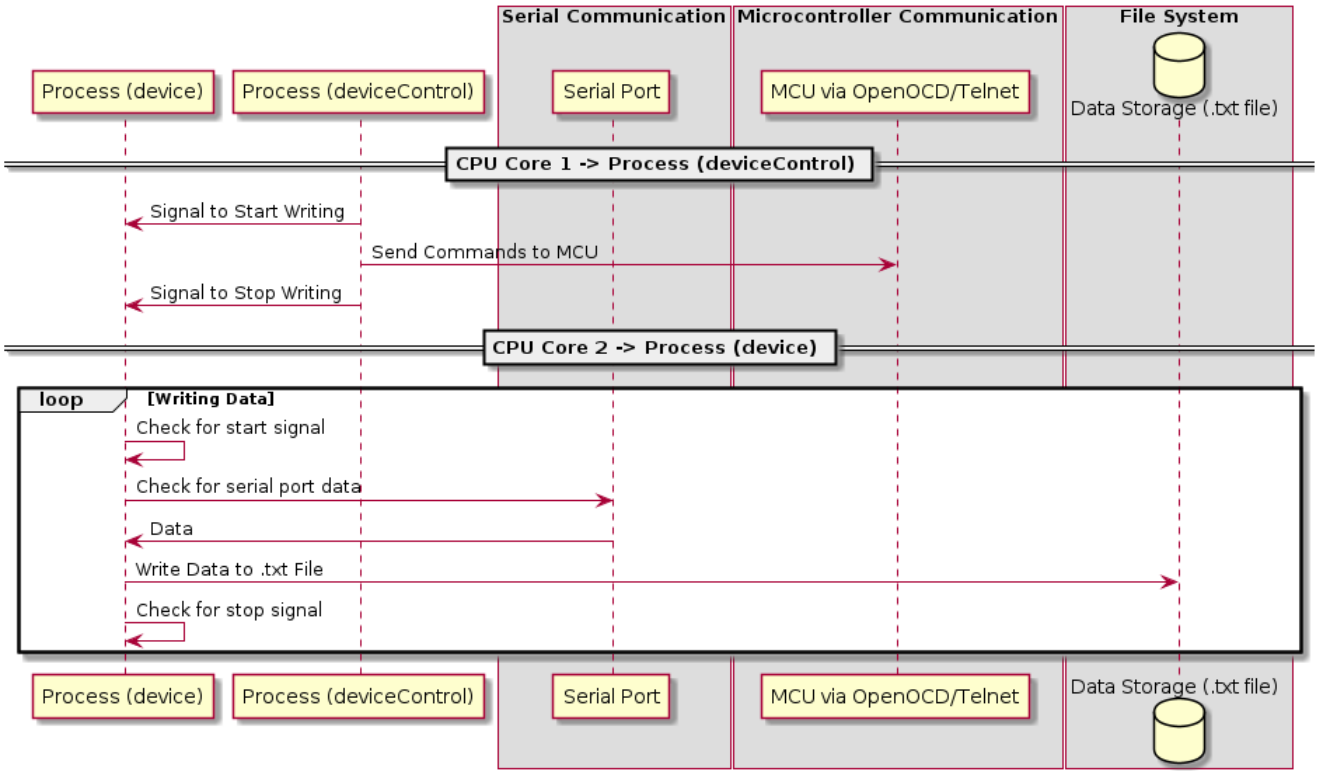
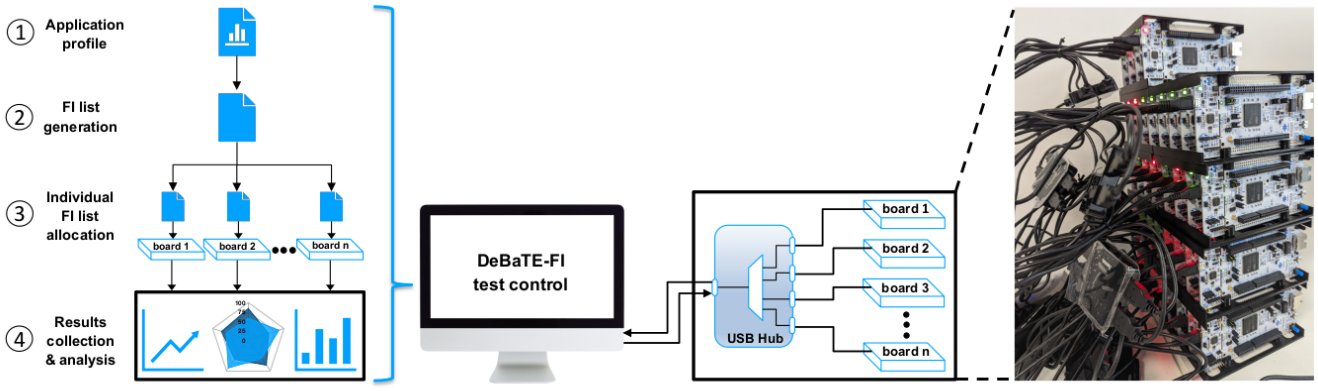


Figure 3.7: The improved multi-processing design of the application. CPU Core 2 now allocates all the CPU time to sending commands to the MCU.



(a) DeBaTE-FI Soft Error Assessment Flow

(b) DeBaTE-FI Hardware Architecture

Figure 3.8: DeBaTE-FI platform application on the main setup using 36 STM32 boards [3].

Results and Discussion

4.1 Objective 1: `mpbenchmark`

Figure 4.1 displays the results collected from desktop (x86) processor. The legend entries "C++" and "C++(SIMD Optimised)" represent the proposed solution with SIMD referring to AVX2 instructions being used.

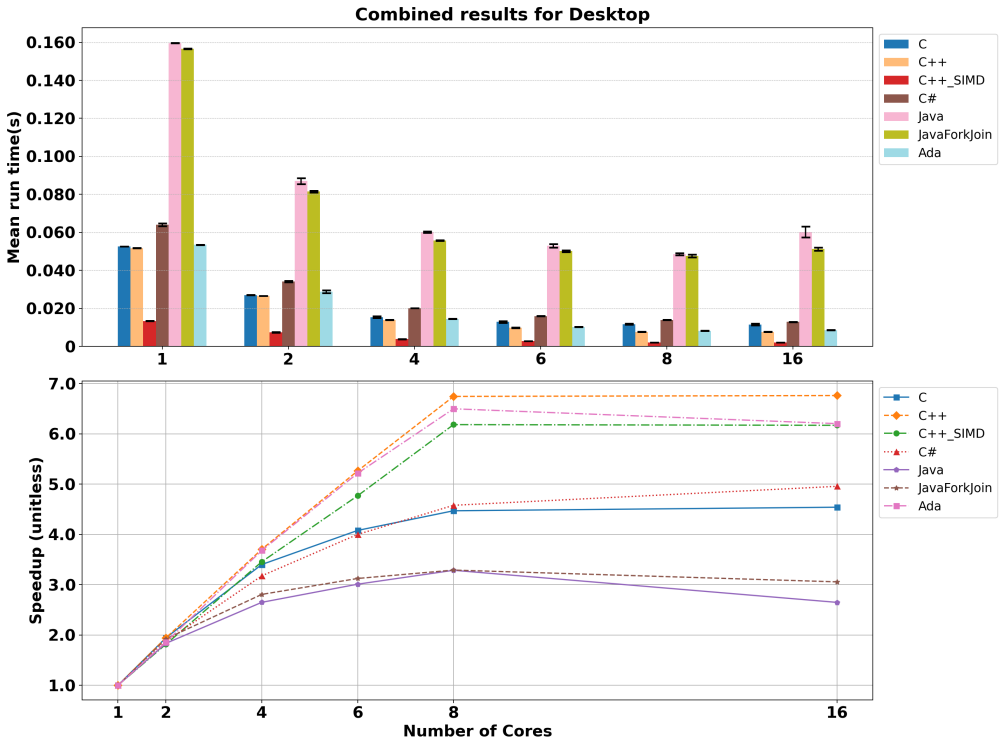


Figure 4.1: Mean benchmark plot in seconds shown top. Speedup plot shown bottom.

On desktop(x86) processor, AVX2 instructions were used to implement SIMD intrinsics therefore, a comparison of decimal precision against the original unoptimised SIMD code is shown in Table 4.1 with run time collected using 16 cores.

Programming language/configuration	Decimal value of π	Mean run time(s)
C++	3.1415926535897643	0.007659
C++/AVX2	3.1415926535899033	0.002173
C	3.1415926535897643	0.011577
Ada	3.1415926535897643	0.008623

Table 4.1: Comparing the decimal precision of the AVX2 enhanced solution with the original.

The proposed C++ solution outperformed the original implementations in C and Ada by over 11%, marking a significant reduction in runtime. Additionally, it achieved a higher speed-up compared to the Ada solution, as illustrated in Figure 4.1. The SIMD-enhanced solution achieved a remarkable runtime reduction

of over 70%, significantly outperforming all other solutions. However, this version did not show as large a speedup when run with an increased number of threads, which is not surprising given the already low runtime with a single thread. Moreover, the AVX2-enhanced code demonstrated decimal precision up to 12 decimal places, with discrepancies appearing from the 13th decimal place onwards, as shown in Table 4.1. This suggests a slight consideration that using AVX2 instructions might lead to reduced precision for calculations requiring high decimal accuracy. However, the reduction in decimal precision has been minor, and its effects on the application have been largely inconsequential. Given the dramatic improvement in performance with AVX2 instructions, the minor loss of decimal precision seems negligible compared to the benefits. Both solutions have met their objectives, with the first achieving superior speed-up and the second providing insights into CPU performance when SIMD intrinsics are utilized.

Another noteworthy observation is the lack of performance gain when scaling from 8 to 16 cores. The processor used supports simultaneous multi-threading (SMT), commonly branded as “Hyper-threading” for Intel CPUs, which is typical in modern x86 processors. SMT theoretically allows each physical core to execute two threads, and an 8-core processor with SMT would appear to have 16 cores. However, the proposed solutions, along with other compiled languages like C and Ada, showed negligible performance gains by utilizing the virtual cores. Java experienced a slight performance degradation, while C# was the only language to demonstrate a performance increase. Using the results from the x86 processor, we can conclude that the additional cores provided by SMT did not enhance performance and, in some cases, even degraded it.

Figures 4.2 and 4.3 show the results collected from the latest Raspberry Pi 5 (Cortex A-76 processor) and Raspberry Pi 4(Cortex A-72 processor) respectively.

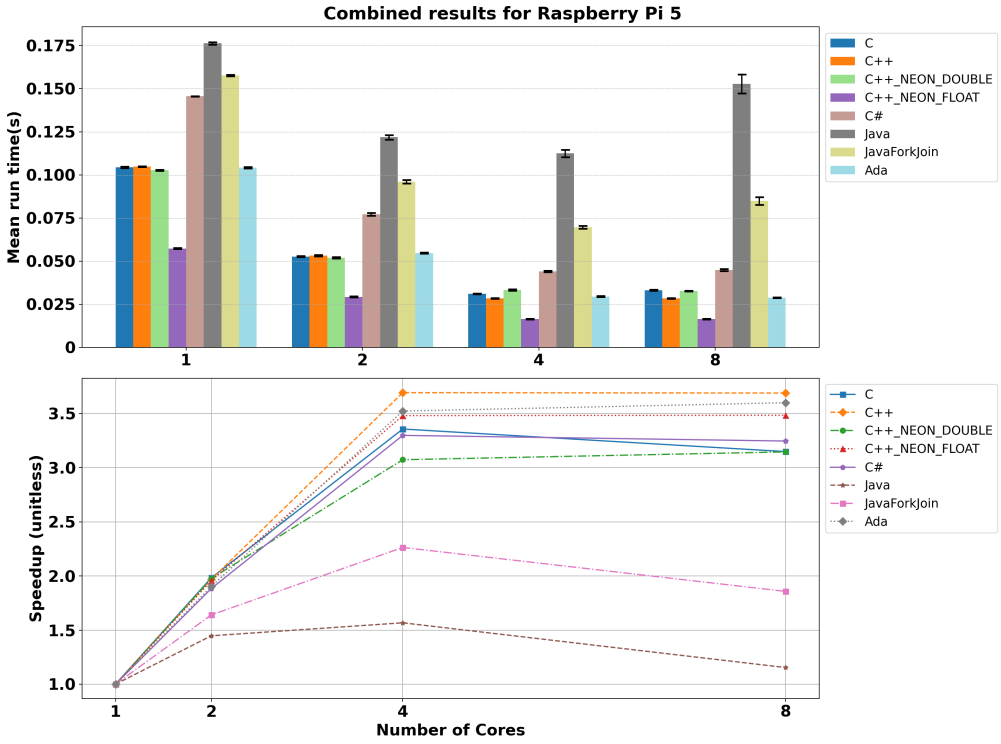


Figure 4.2: Mean benchmark plot in seconds shown top. Speedup plot shown bottom.

The decimal precision of using single and double precision floating point NEON instructions on the Raspberry Pi processors is compared in Table 4.1, with run time collected using 4 cores.

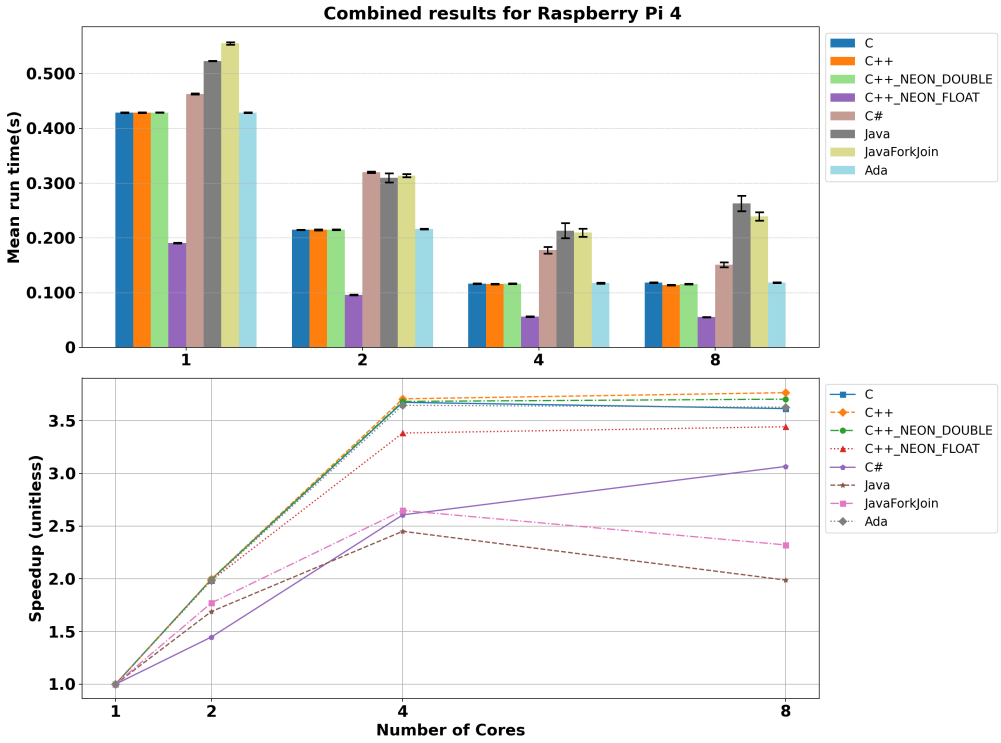


Figure 4.3: Mean benchmark plot in seconds shown top. Speedup plot shown bottom.

Programming language/configuration	Decimal value of π	Run time RPI5(s)	Run time RPI4(s)
C++	3.1415926535897643	0.028356	0.115569
C++/single-precision NEON	3.141531467437744	0.016477	0.056352
C++/double-precision NEON	3.14159265358986	0.033399	0.116423
C	3.1415926535897643	0.031107	0.116538
Ada	3.1415926535897643	0.029549	0.117536

Table 4.2: Comparing the decimal precision of the NEON enhanced solutions.

The proposed solution outperformed both the C and Ada solutions in terms of runtime and speedup. On the Raspberry Pi 5, a slight reduction in runtime (about 4%) and a notable improvement in speedup were observed. The results on the Raspberry Pi 4 were less impressive, showing only a 1% improvement in performance and a marginal increase in speedup. Nevertheless, the proposed C++ solution outperformed the original solutions in terms of runtime and speedup on both Raspberry Pi devices, although the improvement was marginal on the Raspberry Pi 4.

The NEON enhanced solutions using single precision floating-point produced over a 40% reduction in run time at the cost of lower speedup across threads and a reduced decimal precision to four decimal places. On the Raspberry Pi 5, the NEON solution using double precision failed to reduce runtime and produced the worst overall speedup across varying numbers of threads. On the Raspberry Pi 4, the NEON double precision solution did not reduce the runtime and produced a similar speedup to other solutions. However, the double precision solutions did offer far superior decimal precision, up to 12 decimal places. Given NEON instructions' reduced support for double precision floating points, developers must choose between single precision for significantly improved performance but lower decimal precision, and double precision, which did not improve performance on the tested embedded processors. These results are summarised in Table 4.1. When comparing "deadlines missed", on desktop and Raspberry Pi 5 there were no "deadlines missed" from any of the compiled languages. On Raspberry Pi 4, the proposed C++ solution missed a few deadlines reporting a noticeable improvement over C and Ada versions however the NEON single solution did not miss any deadlines showcasing a great improvement not just in run time but also in the "deadline missed" category. In depth results about "deadlines missed" and results from Raspberry Pi 3 were slightly tangential to the project's main objective, therefore they can be found in the [Appendix 2].

Benchmarks allow us to compare the latest Raspberry Pi 5 (Cortex A-76) against the older Raspberry Pi 4 (Cortex A-72). The Cortex A-76 performed 75% faster in terms of runtime when comparing the proposed C++ solution, roughly 4x faster. The Cortex A-76 processor offered a 42% reduction in runtime when NEON(float) enhanced code was used compared to a 51% reduction in runtime for the Cortex A-72. Both processors exhibited a lower speedup when NEON instructions were used, similar to what was observed with the x86 processor. The superior performance of the Cortex A-76 may be attributed to a faster clock speed of 2.4 GHz compared to 1.5GHz of the older Cortex A-72 [31].

The proposed C++ solution surpassed the previously developed mpbenchmark [4] on both x86 and ARM processors. It offered better speedup across threads, and an alternate novel SIMD enhanced version (where the application decides whether to use AVX2 or NEON depending on the processor type) can be utilised to analyse the CPUs' SIMD performance and scalability. This unprecedented improvement is part of an upcoming publication.

4.2 Objective 2: MobileNet

Figure 4.4 shows the results collected after parallelising MobileNet, from desktop(x86 processor). The results show a comparison between SIMD optimisations turned on or off.

The mean benchmark time of the original MobileNet application [10] was 1474.2 ms(using the -O3 optimisation flag). The benchmark time using the maximum threads(16) with SIMD was 188.24ms and without SIMD it was 294.68ms. The SIMD optimisations were turned on and off as shown in Listing 3.3. This presents a drastic reduction in run time of 87.2% and 80.0% for the SIMD and non-SIMD solutions respectively. Using multi-threading on the desktop processor yielded exceptional results and a dramatic

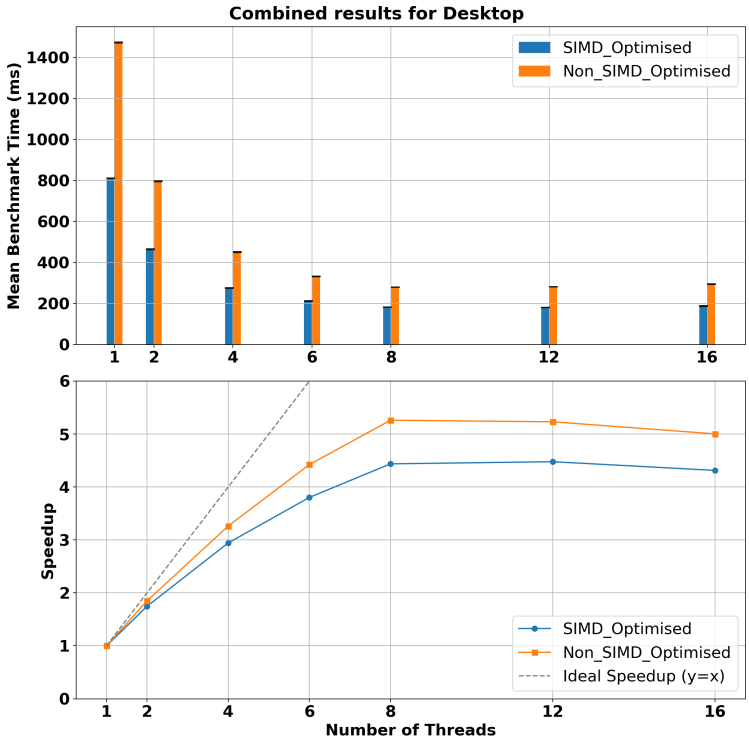


Figure 4.4: Mean benchmark plot in milliseconds shown top. Speedup plot shown bottom.

improvement of the application. The results also indicate a performance degradation when using threads greater than 8 which is the maximum physical cores on the system. Both of the solutions produced a slight degradation of performance when going from 8 to 16 threads, suggesting once again the virtual cores from SMT did not yield in performance gains.

The performance on the Raspberry Pi devices was a little more nuanced. The results of having different parallel regions and SIMD optimisations turned on or off were compared. The plots have the following legends:

1. `3_Parallel_Regions_SIMD`: all three functions `ConvLayer::forward()`, `BatchNormalLayer::forward()` and `ConvLayer::Addpad()` have been parallelised with SIMD optimisations.
2. `3_Parallel_Regions`: all three functions `ConvLayer::forward()`, `BatchNormalLayer::forward()` and `ConvLayer::Addpad()` have been parallelised without SIMD optimisations.
3. `2_Parallel_Regions_SIMD`: only two functions `ConvLayer::forward()` and `BatchNormalLayer::forward()` have been parallelised with SIMD optimisations.
4. `2_Parallel_Regions`: only two functions `ConvLayer::forward()` and `BatchNormalLayer::forward()` have been parallelised without SIMD optimisations.

Figures 4.5 and 4.6 show results collected from Raspberry Pi 5 and Raspberry Pi 4 respectively.

The results indicate that the optimal configuration for the Raspberry Pi 5 is "`3_Parallel_Regions`". This resulted in the lowest run time and best overall speedup when compared to other configurations. For the Raspberry Pi 4 the results were harder to interpret, the lowest run time was achieved with "`2_Parallel_Regions`" but the best speedup was achieved with "`2_Parallel_Regions_SIMD`". Surprisingly SIMD optimisations on both of the Raspberry Pi devices did not result in reduction of run time

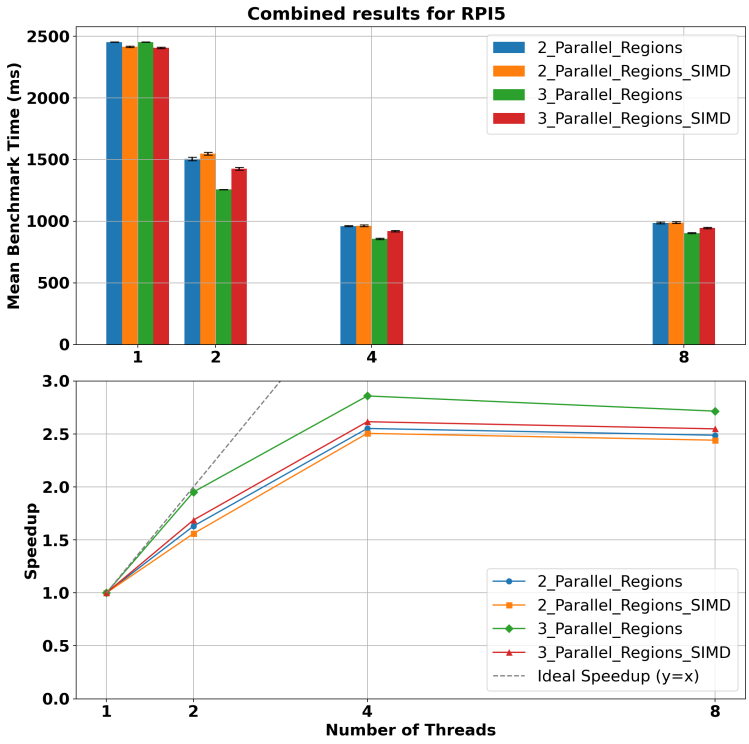


Figure 4.5: Mean benchmark plot in milliseconds shown top. Speedup plot shown bottom.

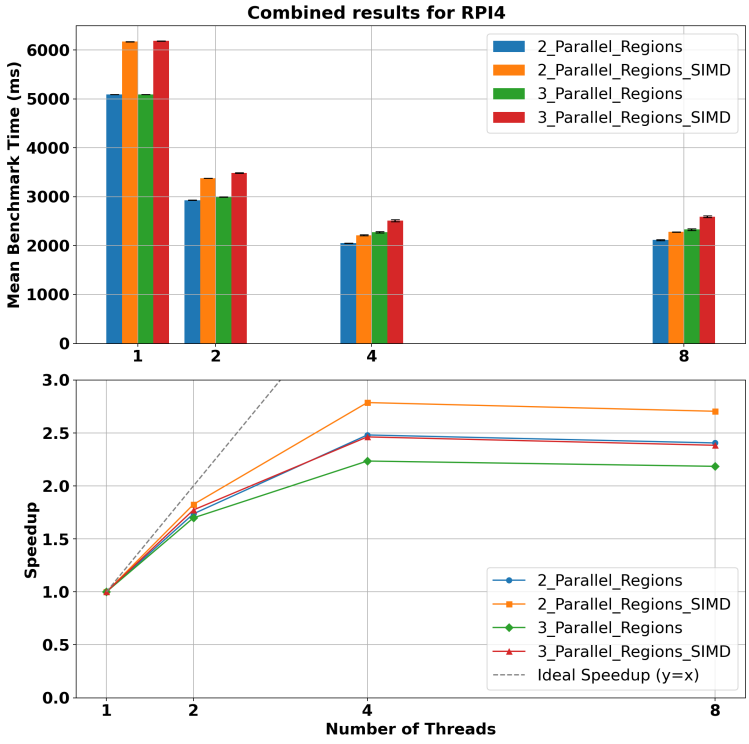


Figure 4.6: Mean benchmark plot in milliseconds shown top. Speedup plot shown bottom.

Metric	Desktop	RPI5	RPI4
Original run time(ms)	1474.2	2462.33	5108.77
Lowest run time(ms)	188.24	858.10	2050.78
Best speedup	5.26	2.86	2.79
Optimal threads	8	4	4
Performance gain versus original(%)	87.2	65.2	59.90
Performance gain versus RPI4(%)	90.8	58.2	0

Table 4.3: Comparing performance metrics across CPUs.

contrary to what was observed in objective 1. When comparing the two processors the `Cortex A-76` performed 58% faster and offered the best performance with three parallel regions and also achieved a slightly better speedup(`2.86` versus `2.76`). Summarised results can be found in Table 4.3. These results indicate that the `Cortex A-76` is better able to take advantage of parallelised regions compared to the `Cortex A-72`.

Contrasting to the Raspberry Pi devices the best performance on desktop was achieved with 3 parallel regions and SIMD optimisations. This presents an interesting challenge to the developer when porting this application on embedded devices, results collected from the Raspberry Pi devices showcase the need for tailoring the multi-threading architecture according to the target CPU to achieve optimal performance. Strangely the SIMD optimisations from the `OpenMP` library did not improve performance on the Raspberry Pi devices, this can be further investigated by manually implementing `NEON` instructions and compare its effects, similar to the method in objective 1. Both of the Raspberry Pi devices required slightly different configurations to achieve optimal performance. Optimal performance would be crucial in embedded processors part of drones and/or unnamed aerial vehicles(UAVs) [32].

4.3 Objective 3: `DeBaTE-FI` platform

Figure 4.7 shows the results collected from the local setup using four `STM32F767ZI` MCUs. The legend of the plot in in Figure 4.7 represent:

1. `Original` : the unchanged `DeBaTE-FI` platform application.
2. `Python_Optimised_C++_library` : the application with the optimised multi-processing design and using the `C++ telnet` library.
3. `Python_Optimised` : the application with the optimised multi-processing design with the original `Python telnet` library.
4. `Python_Unoptimised_C++_library` : the application with the original architecture but with the `C++ telnet` library.

The `Python_Unoptimised_C++_library` solution achieved a 55% reduction in runtime, reducing it to 1.45 minutes from 3.17 minutes. However, it resulted in a lower speedup compared to the original, likely due to the limited runtime achieved using only one board. Conversely, the `Python_Optimised` solution delivered the most substantial reduction in runtime of 62% (1.20 minutes compared to 3.17 minutes). Despite this, it also exhibited a lower speedup, which began to degrade after the incorporation of 2 boards. These results were derived from only 5 test scenarios to mitigate the prolonged runtime of

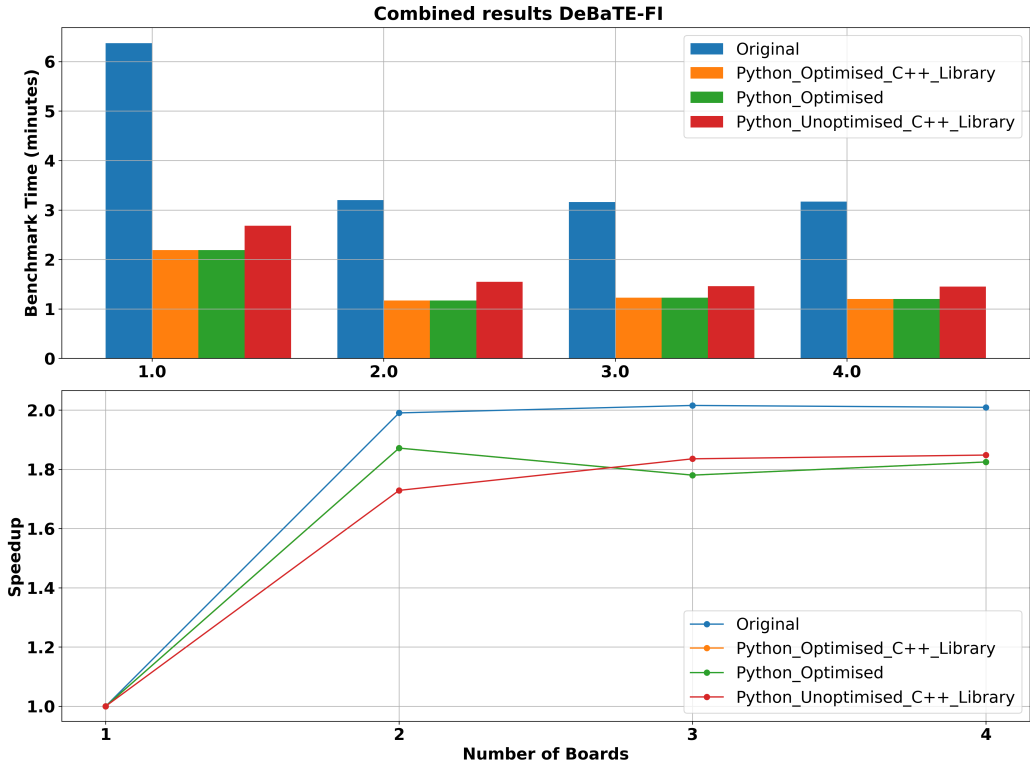


Figure 4.7: Benchmark plot in minutes shown top. Speedup plot shown bottom.

the application. The findings indicate that the `Python_Optimised` solution reached a plateau, showing no further improvements beyond the use of 2 boards. This plateau may be attributed to the limited runtime where adding additional boards did not yield performance gains, suggesting that the optimal performance with this improved solution can be achieved using only 2 boards instead of 4.

Surprisingly, no significant performance improvement was observed with the `Python_Optimised_C++_library` solution. Given previous results, one might expect that integrating the `C++` library into the application would enhance performance, and that optimising the multiprocessing design would further boost efficiency. Combining the two was anticipated to yield an even larger performance increase, yet this did not occur. In the `Python_Optimised` solution, adding the `C++` library did not enhance performance; in fact, the performance was identical to when the `C++` library was not included. This outcome raises an intriguing question: why did only the original `DeBaTE-FI` platform benefit from integrating the `C++` library, while no performance gain was observed when the `C++` library was incorporated into the `Python_Optimised` solution? Investigating this discrepancy will be crucial for future efforts to further optimise the software.

All three developed solutions significantly improved the performance of the `DeBaTE-FI` platform. The `Python_Optimised` solution is particularly convenient as it eliminates the need to compile the `C++` code into a shared library (`.so`) file. The `C++` class can be used in the application as the standard `telnet` library, replacing the older `Telnetlib` `Python` library which was deprecated in `Python 3.11` and is slated for removal in version `3.13` [33]. Additionally, the `C++` library facilitates the creation of custom functions and allows for prioritization of performance. Therefore, this project proposes the use of the newly developed `C++` library (`Telnetlibcpp`) when the `DeBaTE-FI` platform's code is updated to accommodate later versions of `Python`.

Results from the main setup (Figure 3.8) using the `Python_Optimised` solution are presented in Ta-

ble 4.4. A 43.5% performance gain was observed, compared to the 62.1% improvement noted in the local setup. This still represents a significant enhancement in performance. To further investigate the proposed solution, it would be beneficial to analyse results using a varying number of boards and their respective speed-ups in the main setup. This analysis could help determine the ideal number of boards for optimal performance. Additionally, no results were collected using single-board computers (SBCs) like the Raspberry Pi, primarily due to the time constraints of this project. Investigating the performance of the proposed solution on SBCs could be another valuable area of exploration.

Metric	**Local setup**	**Main setup**
STM32 Test Boards	4	36
Test scenarios	5	1000
Original solution run time(min)	3.17	234.47
Optimised solution run time(min)	1.20	132.4
Performance improvement(%)	62.1	43.5

Table 4.4: Comparison of benchmark metrics on the local versus the main setup(Figure 3.8).

Conclusion

The first objective was met with strong results, surpassing the initial aim. A novel benchmark was developed using modern C++, which exceeded the capabilities of the previous mpbenchmark [4] in several aspects especially those implementations in C and Ada. The novel software design was modular and object-oriented, enhancing scalability. An alternate solution was also developed to assess CPU performance using SIMD intrinsics, with the application automatically selecting AVX2 or NEON based on the CPU type. Results from desktop (x86) processors indicated negligible performance gains when utilizing SMT (or Intel's "Hyper-threading"). The latest Raspberry Pi 5, equipped with the Cortex A-76 processor, significantly outperformed the older Raspberry Pi 4's Cortex A-72 by 75%. On ARM-based CPUs, using single-point decimal precision with NEON instructions resulted in a 42-51% performance gain but limited precision to four decimal places, posing a trade-off between performance and decimal precision.

Future research and development would focus on enhancing the benchmark to include more complex and demanding calculations to better assess the performance of high-end CPUs. The x86 processor used in this project completed the benchmark application in less than 10 milliseconds, which might lead to a slightly misleading assessment of CPU capabilities if the application is too simplistic, despite utilising multiple threads. While this benchmark is effective for testing on embedded processors, it may be insufficient for assessing the performance of modern high-end CPUs, which now often feature more than 10 physical cores.

The second objective achieved novel results. The popular image classification algorithm MobileNet [9] [10] was parallelised using the OpenMP library. This optimised application provided a substantial 87% reduction in runtime on the test x86 processor when all system threads were utilised, though SMT did not yield any performance gains. On Raspberry Pi devices, performance improvements were more complex, and optimal performance was achieved when the application's multi-threaded architecture was specifically tailored to the target CPU. The optimised MobileNet application saw performance gains of 59.9% and 65.2% on Raspberry Pi 4 and 5, respectively, when all system threads were employed. However, the use of SIMD optimisations on the Raspberry Pi devices did not result in a significant performance improvement. In terms of application runtime, the latest Raspberry Pi 5 outperformed the Raspberry Pi 4 by 58%. Furthermore, the optimal configuration on the Raspberry Pi 5 involved utilising three parallel regions of the application instead of only two on the Raspberry Pi 4, indicating that the Cortex A-76 processor is better suited to exploit the OpenMP library's parallel regions.

Future work would focus on improvements to the MobileNet application's software design. Enhancements could aim to increase modularity and leverage modern C++ features. Another area for investigation is to determine why the OpenMP library's SIMD clause did not yield performance gains. This could be explored by manually employing NEON instructions and benchmarking the application to analyse the effects.

The third objective, the most challenging of all, was also met with strong results, achieving and even surpassing the initial aims. The DeBaTE-FI platform's application software, specifically its multiprocessing and multithreading capabilities, was optimised to reduce runtime and enhance scalability. The optimised solution successfully reduced runtime by 62.1% in the local setup. In the main setup, the optimised solution achieved a 43.5% reduction in runtime, confirming that the performance improvements are consistent beyond the local environment. Both results demonstrate significant and unprecedented improvements in performance. Additionally, an alternative solution was developed, which involved

integrating an open-source C++ library into the Python application for telnet communication. This solution also showed strong performance gains in the local setup, though not as substantial as the optimised Python solution, and it was not tested in the main setup due to project time constraints. However, the developed C++ library can replace the existing, obsolete Python telnetlib library currently used by the DeBaTE-FI platform. Thus, this project not only enhanced the performance of the DeBaTE-FI platform but also offers a high-performance library that can be integrated into the DeBaTE-FI platform to replace its existing telnet library.

Future work would involve collecting more detailed results using the main setup to further analyse the optimized solution's performance. This task is time-consuming due to the long runtime of this application. Another area worth researching is the optimisation of the application's design, which currently remains convoluted and challenging to enhance, fix bugs, and potentially discover further optimisations for improved performance. The project also proposes using the developed C++ library for telnet communication to replace the application's soon-to-be deprecated telnetlib Python library.

References

[1] S. J. Bigelow. (2022, Mar) Multi-core processor. Accessed: May 6, 2024. [Online]. Available: https://www.techtarget.com/searchdatacenter/definition/multi-core-processor#:~:text=A%20multicore%20processor%20is%20an,with%20parallel%20processing%20and%20multithreading.

[2] K. J. Wong. (2023, Aug) Multithreading vs. multiprocessing explained. Built In. Accessed: May 6, 2024. [Online]. Available: https://builtin.com/data-science/multithreading-multiprocessing

[3] A. Hanneman, J. Gava, V. Bandeira, R. Reis, and L. Ost, "Debate-fi: A debugger-based fault injector infrastructure for iot soft error reliability assessment," in *Proceedings of the IEEE World Forum on Internet of Things (WF-IoT)*. IEEE, 2023.

[4] H. T. Mei and A. Wellings, "Using jetbench to evaluate the efficiency of multiprocessor support for parallel processing," in *Java Technologies for Real-time and Embedded Systems (JTRES)*. University of York, UK: ACM, 2014, accessed: May 6, 2024. [Online]. Available: https://mpbenchmark.sourceforge.net/MeiAndWellings.pdf

[5] M. Y. Qadri, D. Matichard, and K. D. McDonald Maier, "Jetbench: An open source real-time multiprocessor benchmark," in *Architecture of Computing Systems - ARCS 2010*, C. Müller-Schloer, W. Karl, and S. Yehia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 211–221.

[6] Lit Mentor. (2023) Advantages and disadvantages of c programming language. Accessed: May 6, 2024. [Online]. Available: https://litmentor.com/learn-c-tutorial/advantages-and-disadvantages-of-c-language.php

[7] M. Posch. (2019, Sep) Why ada is the language you want to be programming your systems with. Accessed: May 6, 2024. [Online]. Available: https://hackaday.com/2019/09/10/why-ada-is-the-language-you-want-to-be-programming-your-systems-with/

[8] A. Kirsh. (2021, oct) Modern c++ – the evolution of c++. Incredibuild LTD. Accessed: May 6, 2024. [Online]. Available: https://www.incredibuild.com/blog/modern-c-the-evolution-of-c

[9] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. (2017) Mobilenets: Efficient convolutional neural networks for mobile vision applications. Accessed: May 6, 2024.

[10] universecool. (2020) mobilenet. GitHub. Accessed: May 6, 2024. [Online]. Available: https://github.com/universecool/mobilenet

[11] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge 2007. Accessed: May 6, 2024. [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/voc2007/

[12] M. Wojtczyk and A. Knoll, "A cross platform development workflow for c/c++ applications," in *2008 The Third International Conference on Software Engineering Advances*, 2008, pp. 224–229.

[13] J. Elmsheuser, A. Krasznahorkay, E. Obreshkov, and A. Undrus, "Large scale software building with cmake in atlas," *Journal of Physics: Conference Series*, vol. 898, p. 072010, 10 2017.

[14] A. Williams, *C++ Concurrency in Action*, 2nd ed. Manning Publications, 2019, accessed: May 6, 2024. [Online]. Available: https://beefnoodles.cc/assets/book/C++%20Concurrency%20in%20Action.pdf

[15] community wiki. (2008) The definitive c++ book guide and list. Stack Overflow. Accessed: May 6, 2024. [Online]. Available: https://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list

[16] K. Mehta and E. Gabriel, "Multi-threaded parallel i/o for openmp applications," *International Journal of Parallel Programming*, vol. 43, no. 2, pp. 286–309, April 2015. [Online]. Available: https://doi.org/10.1007/s10766-014-0306-9

[17] Haitaomei. mpbenchmark. SourceForge. Accessed: May 6, 2024. [Online]. Available: https://sourceforge.net/projects/mpbenchmark/

[18] A. Müller. (2022) Overview of the c++ fmt library. Accessed: May 6, 2024. [Online]. Available: https://hackingcpp.com/cpp/libs/fmt.html

[19] B. Filipek. (2023) Formatting custom types with std::format from c++20. Accessed: May 6, 2024. [Online]. Available: https://www.cppstories.com/2022/custom-stdformat-cpp20/

[20] (2023) Cmake. Kitware. Accessed: May 6, 2024. [Online]. Available: https://cmake.org/features/

[21] V. Developers. (2024) Callgrind: a call-graph generating cache and branch prediction profiler. Sourceware. Accessed: May 6, 2024. [Online]. Available: https://valgrind.org/docs/manual/cl-manual.html

[22] (2023) Arm neon intrinsics reference. Arm. Accessed: May 6, 2024. [Online]. Available: https://arm-software.github.io/acle/neon_intrinsics/advsimd.html

[23] Microsoft. (2022) Openmp directives. Microsoft. Accessed: May 6, 2024. [Online]. Available: https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170#for-openmp

[24] Y. Fu. (2016, Dec) Performance optimization on modern processor architecture through vectorization. Object Computing. Accessed: May 6, 2024. [Online]. Available: https://objectcomputing.com/resources/publications/sett/december-2016-performance-optimization-on-modern-processor-architecture-through-vectorization

[25] P. Karpiński. (2017, May) Ume::simd tutorials 6: Static alignment. Accessed: May 6, 2024. [Online]. Available: https://gainperformance.wordpress.com/2017/05/07/2629/

[26] B. Frederickson. (2024, Mar.) py-spy: A sampling profiler for python programs. GitHub. Vancouver, BC, Canada. Accessed: May 6, 2024. [Online]. Available: https://github.com/benfred/py-spy

[27] J. Wong. (2024) speedscope. GitHub. San Francisco, California. Accessed: May 6, 2024. [Online]. Available: https://www.speedscope.app/

[28] M. Chaplain. (2024) telnetpp. GitHub. Utrecht, Netherlands. Accessed: May 6, 2024. [Online]. Available: https://github.com/KazDragon/telnetpp

[29] ——. (2024) serverpp. GitHub. Utrecht, Netherlands. Accessed: May 6, 2024. [Online]. Available: https://github.com/KazDragon/serverpp

[30] W. Jakob. (2024, Mar.) pybind11: Seamless operability between c++11 and python. GitHub. Accessed: May 6, 2024. [Online]. Available: https://github.com/pybind/pybind11

[31] A. Allan. (2023, 10) Benchmarking raspberry pi 5. Raspberry Pi Foundation. Accessed: May 6, 2024. [Online]. Available: https://www.raspberrypi.com/news/benchmarking-raspberry-pi-5/

[32] M. A. Haq, G. Rahaman, P. Baral, and A. Ghosh, “Deep learning based supervised image classification using uav images for forest areas classification,” *Journal of the Indian Society of Remote Sensing*, vol. 49, no. 3, pp. 601–606, 2021. [Online]. Available: https://doi.org/10.1007/s12524-020-01231-3

[33] telnetlib - telnet client. Python. Accessed: May 6, 2024. [Online]. Available: https://docs.python.org/3/library/telnetlib.html

[34] AMD. (2024) Amd ryzen™ 7 5800h mobile processor. Official Product Page. Accessed: May 6, 2024. [Online]. Available: https://www.amd.com/en/products/apu/amd-ryzen-7-5800h

[35] M. Donnison. (2023, Sep.) The differences between raspberry pi 4 model b & raspberry pi 5. Kitronik Blog. Accessed: May 6, 2024. [Online]. Available: https://kitronik.co.uk/blogs/resources/the-differences-between-raspberry-pi-4-model-b-raspberry-pi-5

Appendices

1 Appendix (System and Raspberry Pi specifications)

The following setup was used to collect results and benchmarking data:

Specification	Desktop	RPI5	RPI4 Model B
Processor	AMD Ryzen 7 5800H	Cortex A-76	Cortex A-72
Processor Type	x86	ARM	ARM
Processor Speed	3.2 - 4.4 GHz	2.4 GHz	1.5 - 1.8 GHz
Cores/Threads	8/16	4/4	4/4
RAM Size	16GB	8GB	4GB
RAM Type	DDR4	LPDDR4	LPDDR4
RAM Speed	3200 MT/s	4267 MT/s	3200 MT/s
Operating System	Ubuntu Linux 22.04.4 LTS	Debian Linux 12 (bookworm)	Debian Linux 12 (bookworm)
GCC Compiler	11.4.0	12.2.0	12.2.0
GNAT Compiler	10.5.0	12.2.0	12.2.0
Java Compiler	11.0.22	17.0.10	17.0.10
C# (.NET SDK)	7.0.117	6.8.0.15	6.8.0.15

Table 1: Specifications and software Versions of Desktop and Raspberry Pi Systems [31] [34] [35].

2 Appendix (Objective 1: `mpbenchmark`)

Figure 1 shows the profiling results using `Valgrind/Calgrind`.

```
#      Ir          CEst        Source
    11
    12      0.00        0.00 void Worker::operator()(){
    13                           // variables for calculating time related, e.g.deadline etc.
    14                           /* for each thread */
    15                           double TimePoint = 0;
    ...                     ...
    19                           // variables for pi calculation
    20                           const long num_steps = 1000000;
    21                           double step = 1.0 / static_cast<double>(num_steps);
    22      0.00        0.00     int i = 0;
    23                           double x, pi, sum;
    24                           // variables for input data
    25                           double a, b, c, d;
    26
    27                           while (true) {
    28
    29      0.00        0.00         if (m_sharedData.isLineCountGreaterThanNumPoints()) {
            0.00        0.00 64 call(s) to 'SharedPerformanceData::isLineCountGreaterThanNumPoints() const' (mpbenchmark: sharedPerformanceData.cpp, ...)
    30      0.00        0.00             int CurrentPoint = m_sharedData.nextNumPoints();
            0.00        0.00 48 call(s) to 'SharedPerformanceData::nextNumPoints()' (mpbenchmark: sharedPerformanceData.cpp, ...)
    31      0.00        0.00             if (CurrentPoint > m_sharedData.getLineCount()) {
            0.00        0.00 48 call(s) to 'SharedPerformanceData::getLineCount() const' (mpbenchmark: sharedPerformanceData.cpp)
    32                                       m_sharedData.decrementNextNumPoints();
    33                                       break;
    34                               }
    35                               /** Pi calculation */
    36      0.00        0.00             sum = 0;
    37      0.00        0.00             auto piwatch = std::chrono::high_resolution_clock::now();
            0.00        0.00 48 call(s) to '0x000000000010a360'
    38    24.85       24.85              for (i = 0; i < num_steps; i++) {
    39    33.13       33.13                  x = (i + 0.5) * step;
    40    41.42       41.42                  sum += 4.0 / (1.0 + x * x);
    41                               }
    42                               {
    43      0.00        0.00                 pi = sum * step;
    44                               }
    45      0.00        0.00             auto end = std::chrono::high_resolution_clock::now();
            0.00        0.00 48 call(s) to '0x000000000010a360'
    46                               PiTime = std::chrono::duration<double>(end - piwatch).count();
    47
```

Figure 1: `Calgrind` profiling results on the `mpbenchmark` application. Ir - Instruction Fetch, CEst - Cycle estimation.

Listing 1 shows the SIMD enhanced code using `AVX2` instructions of the approximation of π, detailed in the algorithm (Figure 3.3).

```
1  double Worker::approximatePi(){
2      double pi = 0.0; // Initialize pi to 0.0
3      static constexpr long num_steps = 1000000;
4      static constexpr double step = 1.0 / static_cast<double>(num_steps);
5
6      #if defined(__AVX2__)
7      // Use AVX2 SIMD instructions if available
8      double sum = 0; // Initialize scalar sum to accumulate final result
9
10     // Initialise all necessary 256-bit vectors
11     __m256d vec_sum = _mm256_set1_pd(0.0);
12     __m256d vec_step = _mm256_set1_pd(step);
13     __m256d vec_half_step = _mm256_set1_pd(0.5 * step);
14     __m256d vec_one = _mm256_set1_pd(1.0);
15     __m256d vec_four = _mm256_set1_pd(4.0);
16     __m256d vec_x, vec_temp;
```

```
    __m256d vec_i = _mm256_set_pd(3, 2, 1, 0);
    __m256d vec_increment = _mm256_set1_pd(4);

    // Perform 4 computations at once, note "i" is incremented by 4 instead of 1
    for (int i = 0; i < num_steps; i += 4) {
        vec_x    = _mm256_add_pd(_mm256_mul_pd(vec_i, vec_step), vec_half_step);
        vec_temp = _mm256_div_pd(vec_four, _mm256_add_pd(vec_one, _mm256_mul_pd(
            vec_x, vec_x)));
        vec_sum  = _mm256_add_pd(vec_sum, vec_temp);
        vec_i    = _mm256_add_pd(vec_i, vec_increment);
    }
    // Perform horizontal sum on vec_sum to get a scalar sum
    sum = hsum256_pd(vec_sum);
    // Multiply the sum by the step size to approximate pi
    pi = sum * step;

    #else
    // If AVX2/SIMD instructions are unavailable then resort to using regular code
        ...
    #endif

    return pi;
}
```

Listing 1: SIMD enhanced code for approximation of π using `AVX2` instructions.

Listing 2 shows the code for the horizontal sum function for performing the sum of `256-bit` vector using `AVX2` instructions.

```
// Check for AVX2 support
#if defined(__AVX2__)
#include <immintrin.h> // AVX2 and earlier

/**
* @brief Helper function used when approximating pi using AVX2 instructions.
* It is used for performing horizontal sum of a vector.
* @param v vector.
* @return double summed vector.
*/
inline double hsum256_pd(__m256d v) {
    __m256d temp1 = _mm256_hadd_pd(v, v);
    __m256d temp2 = _mm256_permute2f128_pd(temp1, temp1, 0x01);
    __m256d temp3 = _mm256_add_pd(temp1, temp2);
    double result[4];
    _mm256_storeu_pd(result, temp3);
    return result[0]; // The sum of all elements in the vector
}

#elif defined(__ARM_NEON)
// if ARM NEON instructions are available then include the necessary header file
#include <arm_neon.h>

#else
// neither AVX2 or NEON instructions found
#endif
```

Listing 2: Horizontal sum function for `256-bit` `AVX2` vector.

Code for implementing the approximation of π using single and double point floating precision with NEON instructions are shown in Listings 3 and 4.

```
double Worker::approximatePi(){
    double pi = 0.0;
    static constexpr long num_steps = 1000000;
    static constexpr double step = 1.0 / static_cast<double>(num_steps);

    #if defined(__AVX2__)
    // Insert AVX2 specific code here ...

    #elif defined(__ARM_NEON)
    // Insert NEON specific code here
    float32_t f_step = 1.0f / static_cast<float32_t>(num_steps); // Using float
       for NEON
    float32x4_t vec_step = vdupq_n_f32(f_step);
    float32x4_t vec_half_step = vdupq_n_f32(0.5f * f_step);
    float32x4_t vec_one = vdupq_n_f32(1.0f);
    float32x4_t vec_four = vdupq_n_f32(4.0f);
    float32_t sum = 0.0f; // Using float for NEON

    for (int i = 0; i < num_steps; i += 4) {
        float32x4_t vec_i = vsetq_lane_f32(i + 3, vsetq_lane_f32(i + 2,
           vsetq_lane_f32(i + 1, vsetq_lane_f32(i, vdupq_n_f32(0.0f), 0), 1), 2),
           3);
        float32x4_t vec_x = vaddq_f32(vmulq_f32(vec_i, vec_step), vec_half_step);
        float32x4_t vec_temp = vdivq_f32(vec_four, vaddq_f32(vec_one, vmulq_f32(
           vec_x, vec_x)));
        sum += vaddvq_f32(vec_temp); // Horizontal sum of vector elements
    }

    pi = sum * f_step;
    #else
    // Insert regular code here ...
    #endif
    return pi;
}
```

Listing 3: Approximation of π using single point precision NEON instructions.

```
double Worker::approximatePi(){
    double pi = 0.0;
    static constexpr long num_steps = 1000000;
    static constexpr double step = 1.0 / static_cast<double>(num_steps);

    #if defined(__AVX2__)
    // Insert AVX2 specific code here ...

    #elif defined(__ARM_NEON)
    // Insert NEON specific code here
    float64x2_t vec_step = vdupq_n_f64(step);
    float64x2_t vec_half_step = vdupq_n_f64(0.5 * step);
    float64x2_t vec_one = vdupq_n_f64(1.0);
    float64x2_t vec_four = vdupq_n_f64(4.0);
    float64_t sum = 0.0;

    for (int i = 0; i < num_steps; i += 2) {
```

```
        // Since direct lane setting for float64x2_t via intrinsics like vsetq_lane_f64 isn't straightforward,
        // We compute x and its index scalarly and then load them into vectors.
        double x0 = (i + 0.5) * step;
        double x1 = (i + 1.5) * step;
        float64x2_t vec_x = {x0, x1}; // Directly initialize the vector with double values.

        float64x2_t vec_temp = vdivq_f64(vec_four, vaddq_f64(vec_one, vmulq_f64(vec_x, vec_x)));
        sum += vaddvq_f64(vec_temp); // Horizontal sum of vector elements.
    }

    pi = sum * step;
    #else
    // Insert regular code here ...
    #endif
    return pi;
}
```

Listing 4: Approximation of π using double point precision `NEON` instructions.

To use `AVX2` instructions for the application, the following changes were made to the `CMakeLists.txt` file, however no changes were required to use `NEON` instructions, see Listing 5.

```
# Compiler optimizations
include(CheckCXXCompilerFlag)

# Check and enable AVX2 and FMA for x86_64 architecture
CHECK_CXX_COMPILER_FLAG("-mavx2" COMPILER_SUPPORTS_AVX2)
CHECK_CXX_COMPILER_FLAG("-mfma" COMPILER_SUPPORTS_FMA)
if (COMPILER_SUPPORTS_AVX2 AND COMPILER_SUPPORTS_FMA)
target_compile_options(${PROJECT_NAME} PRIVATE -mavx2 -mfma)
endif()

if (CMAKE_SYSTEM_PROCESSOR MATCHES "arm" OR CMAKE_SYSTEM_PROCESSOR MATCHES "aarch64")
# For ARMv8-A (aarch64), NEON is always available. No need for -mfpu=neon
# We can set architecture-specific flags if necessary, but for NEON, it's not required.
endif
```

Listing 5: Adding `-mavx2 -mfma` flags to the `CMake` file to allow the project to use `AVX2` instructions.

Figure 2 shows the results collected from Raspberry Pi 3 (`Cortex-A53` processor).

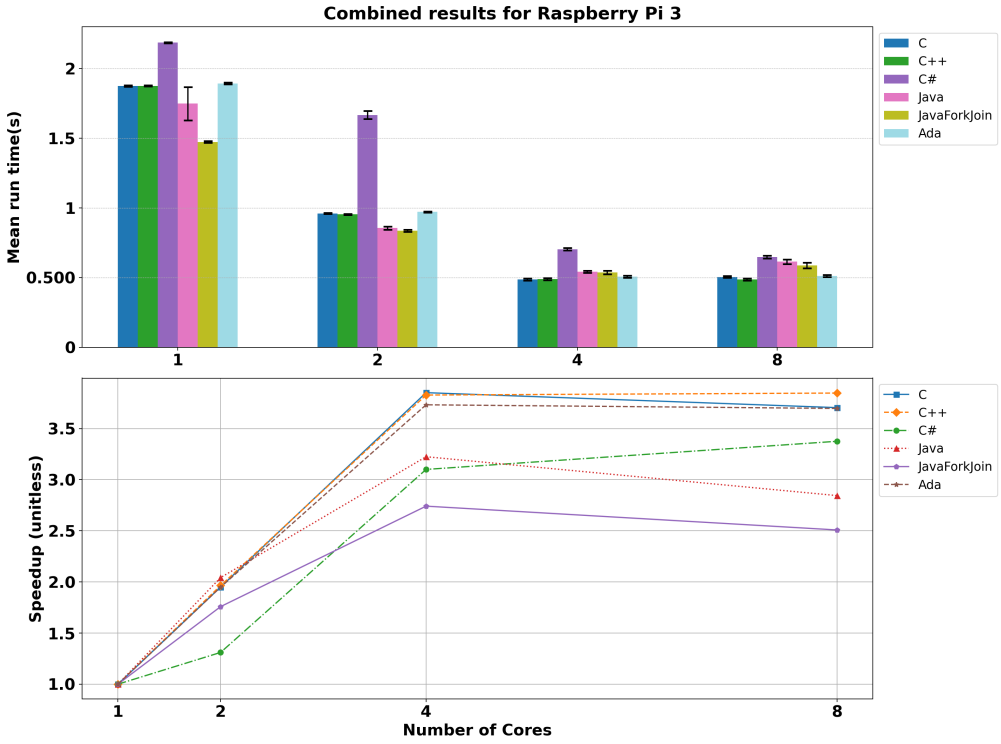


Figure 2: Mean benchmark plot in seconds shown top. Speedup plot shown bottom.

Cores	C	C++	C++(NEON single)	C++(NEON double)	Java	JavaForkJoin	C#	Ada
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0.18	0	0	0
4	0	0	0	0	1.33	0	0	0
8	0	0	0	0	8.21	1.03	0	0

Table 2: Mean deadlines missed, results collected from the Raspberry Pi 5.

Cores	C	C++	C++(NEON single)	C++(NEON double)	Java	JavaForkJoin	C#	Ada
1	0	0	0	0	0.90	0	0	0
2	0	0	0	0	1.00	0.27	0	0
4	0.005	0	0	0.005	1.34	0.52	0	0.01
8	0.03	0.0295	0	0.04	6.35	6.93	2.33	0.04

Table 3: Mean deadlines missed, results collected from the Raspberry Pi 4.

Cores	C	C++	Java	JavaForkJoin	C#	Ada
1	1.17	1.30	1.60	1.10	3.40	1.33
2	2.80	2.47	2.73	2.40	31.63	2.97
4	3.61	3.67	5.13	4.65	19.33	3.83
8	44.60	46.55	30.2	25.23	46.63	43.57

Table 4: Mean deadlines missed, results collected from the Raspberry Pi 3.

3 Appendix (Objective 2: `MobileNet`)

The `MobileNet` repository [10] contained some outdated portions of the code which were updated, certain parts of the code contained Chinese language which were translated into English and `CMake` build software was used to setup the `MobileNet` application.

The main updates required to the original `MobileNet` repository [10] were inside the `readdata.cpp` file (Listing 6).

```
float* ReadData::ReadInput(const char* pcName) {
    std::cout << "Reading Picture: " << pcName << "..." << std::endl;

    // Use cv::Mat for image representation
    cv::Mat srcImage = cv::imread(pcName, cv::IMREAD_UNCHANGED);
    if (srcImage.empty()) {
        std::cerr << "Error: Image not loaded." << std::endl;
        return nullptr;
    }

    // Resize image
    cv::Mat dstImage;
    cv::resize(srcImage, dstImage, cv::Size(m_nInputWidth, m_nInputHeight), 0, 0,
       cv::INTER_LINEAR);

    int nOutputIndex = 0;

    for (int i = 0; i < dstImage.rows; i++) {
        for (int j = 0; j < dstImage.cols; j++) {
            nOutputIndex = i * m_nInputWidth + j;
            cv::Vec3b pixel = dstImage.at<cv::Vec3b>(i, j);
            m_pfInputData[nOutputIndex] = static_cast<float>(pixel[0]) - m_pfMean[
               nOutputIndex];
            m_pfInputData[nOutputIndex + m_nImageSize] = static_cast<float>(pixel
               [1]) - m_pfMean[nOutputIndex + m_nImageSize];
            m_pfInputData[nOutputIndex + 2 * m_nImageSize] = static_cast<float>(
               pixel[2]) - m_pfMean[nOutputIndex + 2 * m_nImageSize];
        }
    }

    std::cout << "Reading Picture Done..." << std::endl;

    return m_pfInputData;
}
```

Listing 6: Updating the `ReadData::ReadInput()` function to use the latest `OpenCV` functions.

Figure 3 shows the translation from Chinese to English language. The classes that the `MobileNet` was trained on were in Chinese therefore it was translated into English, found inside the `Network.cpp` file of the original project [10].

```
-m_vcClass.push_back("室内");
-m_vcClass.push_back("人像");
-m_vcClass.push_back("LDR");
-m_vcClass.push_back("绿植");
-m_vcClass.push_back("商场");
-m_vcClass.push_back("沙滩");
-m_vcClass.push_back("逆光");
-m_vcClass.push_back("日落");
-m_vcClass.push_back("蓝天");
-m_vcClass.push_back("雪景");
-m_vcClass.push_back("夜景");
-m_vcClass.push_back("文本");
+    m_vcClass.push_back("Indoor");
+    m_vcClass.push_back("Portrait");
+    m_vcClass.push_back("LDR");
+    m_vcClass.push_back("Green Plant");
+    m_vcClass.push_back("Shopping Mall");
+    m_vcClass.push_back("Beach");
+    m_vcClass.push_back("Backlit");
+    m_vcClass.push_back("Sunset");
+    m_vcClass.push_back("Blue Sky");
+    m_vcClass.push_back("Snow Scene");
+    m_vcClass.push_back("Night Scene");
+    m_vcClass.push_back("Text");
```

Figure 3: GitHub page showing the commit diff when classes were translated from Chinese to English.

The application was modified to have three command line arguments, this served primarily to make testing and collecting benchmark results easier for development purposes. The three arguments are as follows and can be seen in code in listing 7:

1. `test_all`: This argument can be altered to either test one image or test all images in the data folder.
2. `write_to_file`: This argument is for writing benchmark data to a `.txt` file, this was useful for development purposes. It is recommended to leave this blank, so the project does not save any benchmark data.
3. `threads`: The number of threads that will be used by the application. It can be left blank to use the maximum number of available threads.

```
int main(int argc, char* argv[])
{
    //----------------------------Command line app logic
    bool testAllImages = false;
    bool writeDataToFile = false;  // Declare the variable to handle write to file
        logic
    bool g_DebugMode = true;       // Default debug mode setting
    int numThreads = 1;            // Default to 1 thread unless specified

    if (argc > 1) {
        std::string firstArg(argv[1]);
        testAllImages = (firstArg == "test_all");
    }
    if (argc > 2) {
        std::string secondArg(argv[2]);
        writeDataToFile = (secondArg == "write_to_file");
        g_DebugMode = !writeDataToFile;
    }
    if (argc > 3) {
        numThreads = std::atoi(argv[3]);
        if (numThreads <= 0) {
            numThreads = omp_get_max_threads();
        }
    }
    //-------------------------------------------------
    // Further logic and operations can follow here
```

```
}
```

Listing 7: Altering the `MobileNet` application to use three command line arguments. `./mobilenet [test_all] [write_to_file] [threads]`

`Calgrind` profiling results from the `MobileNet` application show the following functions with the highest self-cost, these were `ConvLayer::forward()`, `BatchNormalLayer::forward()` and `ConvLayer::Addpad()` as seen in Figure 4.

Incl.	Self	Called	Function	Location
98.06	97.46	27	ConvLayer::forward(float*)	mobilenet: convLayer.cpp
0.69	0.61	27	BatchNormalLayer::forward(float*)	mobilenet: batchnormalLayer.cpp
0.59	0.59	27	ConvLayer::Addpad(float*)	mobilenet: convLayer.cpp
0.21	0.21	27	ReluLayer::forward(float*)	mobilenet: reluLayer.cpp
0.20	0.19	165 888	0x0000000000028d40	libjpeg.so.8.2.2
0.20	0.19	36 784	do_lookup_x	ld-linux-x86-64.so.2: dl-lookup.c, dl-protectec
0.14	0.14	2 304	0x000000000003cfb0	libopencv_imgcodecs.so.4.5.4d
0.06	0.06	165 888	0x00000000000373d0	libjpeg.so.8.2.2
0.35	0.06	1 440	0x0000000000018ed0	libjpeg.so.8.2.2
0.04	0.04	2 304	0x0000000000034030	libjpeg.so.8.2.2
0.07	0.04	5 042 688	sqrt	libm.so.6: w_sqrt_compat.c
0.04	0.04	331 776	0x0000000000028a30	libjpeg.so.8.2.2
0.04	0.04	3 458	__memset_avx2_unaligned_erms	libc.so.6: memset-vec-unaligned-erms.S
0.03	0.03	1	init_scan_orders()	libde265.so.0.1.1

Figure 4: `Calgrind` profiling results, functions with the highest self-cost.

The parallelisation of functions `BatchNormalLayer::forward()` and `ConvLayer::Addpad()` can be found in listings 8 and 9.

```
void BatchNormalLayer::forward(float *pfInput)
{
    #pragma omp parallel for collapse(2) shared(m_nInputNum, m_nInputSize, pfInput
        , m_pfOutput, m_pfFiller, m_pfMean, m_pfVar, m_pfBias)
    for (int i = 0; i < m_nInputNum; i++)
    {
        for (int j = 0; j < m_nInputSize; j++)
        {
            int nOutputIndex = i * m_nInputSize + j;

            m_pfOutput[nOutputIndex] = m_pfFiller[i] * ((pfInput[nOutputIndex] -
                m_pfMean[i])
            / sqrt(m_pfVar[i] + 1e-5)) + m_pfBias[i];
        }
    }
}
```

Listing 8: Parallelising the `BatchNormalLayer::forward()` function.

```
void ConvLayer::Addpad(float *pfInput)
{
    // Only use OpenMP pragmas if EMBEDDED_PROC is not defined, such as on a
        laptop/desktop CPU
    #ifndef EMBEDDED_PROC
    #pragma omp parallel for collapse(2)
    #endif
```

```
    for (int m = 0; m < m_nInputNum; m++)
    {
        for (int i = 0; i < m_nInputPadWidth; i++)
        {
            for (int j = 0; j < m_nInputPadWidth; j++)
            {
                if ((i < m_nPad) || (i >= m_nInputPadWidth - m_nPad))
                {
                    m_pfInputPad[m * m_nInputPadSize + i * m_nInputPadWidth + j] =
                        0;
                }
                else if ((j < m_nPad) || (j >= m_nInputPadWidth - m_nPad))
                {
                    m_pfInputPad[m * m_nInputPadSize + i * m_nInputPadWidth + j] =
                        0;
                }
                else
                {
                    m_pfInputPad[m * m_nInputPadSize + i * m_nInputPadWidth + j] =
                        pfInput[m * m_nInputSize + (i - m_nPad) * m_nInputWidth +
                       (j - m_nPad)];
                }
            }
        }
    }
}
```

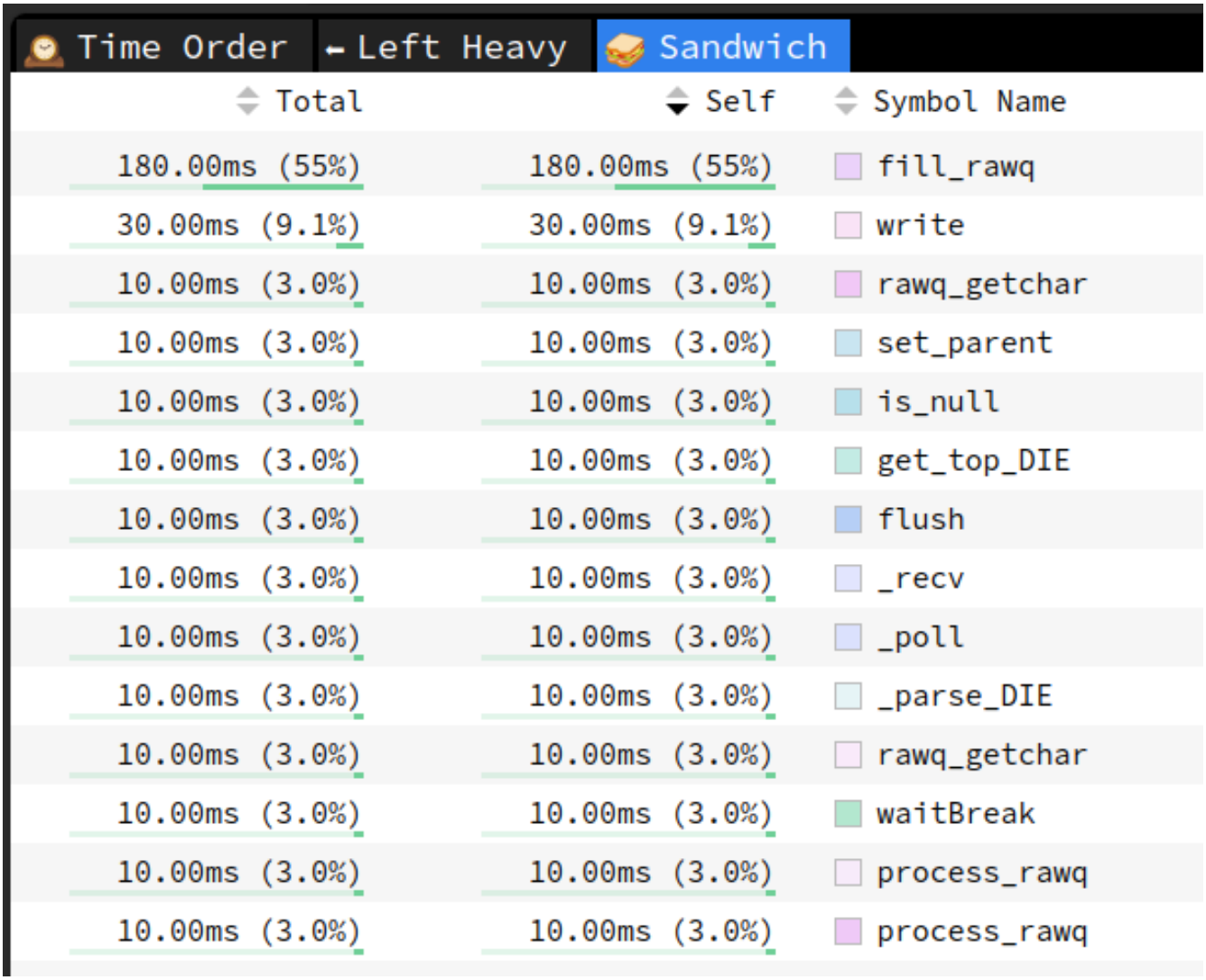
Listing 9: Parallelising the `ConvLayer::Addpad()` function If the `EMBEDDED_PROC` is not set.

4 Appendix (Objective 3: DeBaTE-FI platform)

The application was profiled using py-spy tool. The application spawned multiple processes, therefore they were profiled separately. Profiling results from the process responsible for the GUI of the application are shown in Figure 5 and the profiling results from the process responsible for communicating with the MCUs are shown in Figure 6.

Total	Self	Symbol Name
1:21 (36%)	1:21 (36%)	__init__
1:16 (34%)	1:16 (34%)	in_waiting
3:02 (80%)	9.55s (4.2%)	receiveDataThread
5.74s (2.5%)	5.74s (2.5%)	is_set
43.78s (19%)	5.72s (2.5%)	receiveDataThread
5.54s (2.4%)	5.54s (2.4%)	__enter__
11.06s (4.9%)	5.37s (2.4%)	__enter__
9.63s (4.2%)	5.30s (2.3%)	__exit__
14.45s (6.4%)	4.69s (2.1%)	is_set
1:33 (41%)	4.48s (2.0%)	read
4.21s (1.9%)	4.21s (1.9%)	__exit__
15.38s (6.8%)	4.09s (1.8%)	is_set
3.08s (1.4%)	3.08s (1.4%)	__init__
2.41s (1.1%)	2.41s (1.1%)	is_set
1.94s (0.85%)	1.94s (0.85%)	__init__
1.53s (0.67%)	1.53s (0.67%)	read
1.51s (0.66%)	1.51s (0.66%)	read
1.25s (0.55%)	1.25s (0.55%)	__init__
490.00ms (0.22%)	490.00ms (0.22%)	__init__
470.00ms (0.21%)	470.00ms (0.21%)	read
470.00ms (0.21%)	470.00ms (0.21%)	receiveDataThread

Figure 5: Profiling results for the application GUI process visualised in speedscope web application [27].



Time Order | Left Heavy | Sandwich

Total	Self	Symbol Name
180.00ms (55%)	180.00ms (55%)	fill_rawq
30.00ms (9.1%)	30.00ms (9.1%)	write
10.00ms (3.0%)	10.00ms (3.0%)	rawq_getchar
10.00ms (3.0%)	10.00ms (3.0%)	set_parent
10.00ms (3.0%)	10.00ms (3.0%)	is_null
10.00ms (3.0%)	10.00ms (3.0%)	get_top_DIE
10.00ms (3.0%)	10.00ms (3.0%)	flush
10.00ms (3.0%)	10.00ms (3.0%)	_recv
10.00ms (3.0%)	10.00ms (3.0%)	_poll
10.00ms (3.0%)	10.00ms (3.0%)	_parse_DIE
10.00ms (3.0%)	10.00ms (3.0%)	rawq_getchar
10.00ms (3.0%)	10.00ms (3.0%)	waitBreak
10.00ms (3.0%)	10.00ms (3.0%)	process_rawq
10.00ms (3.0%)	10.00ms (3.0%)	process_rawq

Figure 6: Profiling results for the `OpenOCD/telnet` process visualised in `speedscope` web application [27].

Figure 7 shows the thought process of designing the `C++` wrapper class to use the necessary `telnet` library for `DeBaTE-FI` platform.

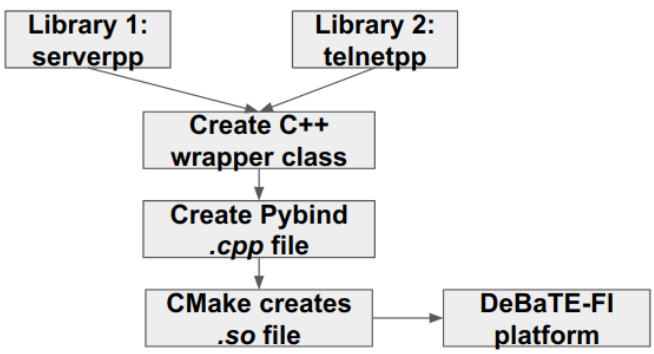


Figure 7: Design process for creating the `telnet` library using `C++`.

The C++ wrapper was created by emulating the Python's telnetlib library. Using the code in listing 10 the following functions required emulation:

1. read_some()
2. write()
3. Exec()
4. Readout()

```
import telnetlib

class OpenOCD:
    def __init__(self, Host="localhost", Port=4444):
        try:
            self.tn = telnetlib.Telnet(Host, Port)
        except ConnectionRefusedError as e:
            print("ERROR: Could not open port",Port)
            raise e
        self.Readout()

#
# Communication functions
#
    def Readout(self):
        s = ''
        Lines = []
        while True:
            s += self.tn.read_some().decode('UTF-8') # 'telnetlib' function called
            l = s.splitlines()
        if len(l) > 1:
            for s in l[:-1]:
                if len(s) > 0:
                    Lines.append(s)
            s = l[-1]
    if s == '> ':
        return Lines

    def Exec(self, Cmd, *args):
        Text = Cmd
        for arg in args:
            if arg:
                Text += ' ' + arg
        Text += '\n'
        self.tn.write(bytearray(Text, 'UTF-8')) # 'telnetlib' function called
        return self.Readout()
```

Listing 10: Functions used from Python's telnetlib library.

Using the open-source telnet libraries in C++: telnetpp [28] and serverpp [29] the necessary functions were created in the wrapper class seen in Figure 3.5. Listing 11 shows the code:

```
std::vector<serverpp::byte> TelnetlibCpp::read_some() {
    incoming_data_ = false; // Ensure the flag is reset before starting the async
       operation
    receivedData_.clear(); // Clear previous data
    // Return a copy of the data or move receivedData as appropriate
```

```
    telnet_session_.async_read([this](gsl::span<const serverpp::byte> data) {
        if (!data.empty()) {
            // Directly assign the data since receivedData is a member variable
            receivedData_.assign(data.begin(), data.end());
        }
        incoming_data_ = true; // Indicate that data has been received or the operation is complete
    });
    // Efficiently wait for the async operation to complete using io_context.run_one()
    while (!incoming_data_) {
        io_context_.run_one();
    }
    // Return a copy of the data or move receivedData as appropriate
    return receivedData_;
}

void TelnetlibCpp::write(const std::vector<serverpp::byte>& data) {
    // Assuming 'serverpp::byte' is compatible with 'telnetpp::byte'
    // Wrap the data in a 'telnetpp::bytes' variant to create a 'telnetpp::element'
    telnetpp::bytes byte_data(data.data(), data.size());
    telnetpp::element elem = byte_data;  // Implicitly converts to variant

    // Now call the write method with the constructed element
    telnet_session_.write(elem);
}

std::vector<std::string> TelnetlibCpp::Exec(const std::string& cmd, const std::vector<std::string>& args) {
    std::string text = cmd;
    for (const auto& arg : args) {
        if (!arg.empty()) {
            text += ' ' + arg;
        }
    }
    text += '\n';

    // write(Cmd), write directly to avoid unnecessary conversions
    telnetpp::bytes byte_data(reinterpret_cast<const std::uint8_t*>(text.data()), text.size());
    telnetpp::element elem = byte_data;
    telnet_session_.write(elem);
    //
        ------------------------------------------------------------------------------------------


    // Read and return the output
    return Readout();
}

std::vector<std::string> TelnetlibCpp::Readout() {
    std::string s;
    std::vector<std::string> lines;

    while (true) {
        std::vector<std::uint8_t> byte_data = read_some();
```

```
        std::string chunk(byte_data.begin(), byte_data.end());

        s += chunk;

        size_t start_pos = 0;
        size_t pos;
        while ((pos = s.find('\n', start_pos)) != std::string::npos) {
            if (pos > start_pos) { // Ignore empty lines
                std::string line(s.begin() + start_pos, s.begin() + pos);
                line.erase(std::remove(line.begin(), line.end(), '\r'), line.end()
                    );
                line.erase(std::remove(line.begin(), line.end(), '\x00'), line.end
                    ());

                if (line.size())
                lines.push_back(std::move(line)); // Use move semantics to avoid
                    copying
            }
            start_pos = pos + 1;
        }
        s.erase(s.begin(), s.begin() + start_pos); // Only erase the processed
            part

        if (s.find('>') != std::string::npos) {
            return lines; // Assuming lines are already filtered correctly
        }
    }
}
```

Listing 11: Emulating functions `read_some()`, `write()`, `Exec()` and `Readout()` in C++.

This was integrated into the Python application using a tool called `pybind11`. This required creating a `.cpp` file to allow the C++ funtions to be called from the Python file, this is seen in Listing 12.

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h> // For automatic conversion of std::vector
#include "telnetlibcpp.hpp" // Include your class definition

namespace py = pybind11;

PYBIND11_MODULE(telnetlibcpp, m) {
    py::class_<TelnetlibCpp>(m, "TelnetlibCpp")
    .def(py::init<const std::string&, serverpp::port_identifier>())
    .def("Readout", &TelnetlibCpp::Readout) // Bind the Readout function
    // Bind the Exec function, using a lambda to handle optional vector<string>
        args
    .def("Exec", [](TelnetlibCpp& self, const std::string& cmd, const py::list&
        args) {
        std::vector<std::string> vecArgs;
        for (const auto& arg : args) {
            if (!arg.is_none()) {
                vecArgs.push_back(arg.cast<std::string>());
            } else {
                // Optionally, handle None values differently, e.g., by inserting
                    an empty string
                // vecArgs.push_back("");
            }
```

```
        }
        return self.Exec(cmd, vecArgs);
    })
    .def("read_some", &TelnetlibCpp::read_some)
    .def("write", &TelnetlibCpp::write)
    .def("write_raw_sequence", &TelnetlibCpp::write_raw_sequence)
    .def("run", &TelnetlibCpp::run);
}
```

Listing 12: Preparing our C++ class `telnetlibcpp` using `pybind11` to be used with `Python`.

The final step was to compile the C++ codes into a shared library(`.so`) file which was then placed in the directory of the `Python` application where it needed to be used. Listing 13 shows hows this shared library file was created.

```
cmake_minimum_required(VERSION 3.14 FATAL_ERROR)
project(telnetlibcpp)

# Set the C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# Optimization flag -O3 for Release builds
set(CMAKE_CXX_FLAGS_RELEASE "-O3")

# Include Pybind11
include(FetchContent)
FetchContent_Declare(
pybind11
GIT_REPOSITORY https://github.com/pybind/pybind11.git
GIT_TAG v2.6.1
)
FetchContent_MakeAvailable(pybind11)

# Find required Boost components in one call
find_package(Boost REQUIRED COMPONENTS container locale)
find_package(serverpp REQUIRED)
find_package(telnetpp 3.0.0 REQUIRED)
find_package(gsl-lite REQUIRED)

# Source files including the Pybind11 binding file
set(SOURCE_FILES
src/telnetlibcpp.cpp
src/pybind_module.cpp  # Your Pybind11 binding source file
)

# Compile as a shared library
add_library(telnetlibcpp SHARED ${SOURCE_FILES})

target_include_directories(telnetlibcpp PRIVATE include)

target_link_libraries(telnetlibcpp
PRIVATE
KazDragon::serverpp
KazDragon::telnetpp
Boost::container
```

```
Boost::locale
pybind11::module  # Link against Pybind11 module support
)

# Set output library name and ensure it doesn't have the 'lib' prefix
set_target_properties(telnetlibcpp PROPERTIES PREFIX "" OUTPUT_NAME "telnetlibcpp"
    )
```

Listing 13: `CMake` file to produce a shared library(`.so`) file of the `C++` class.