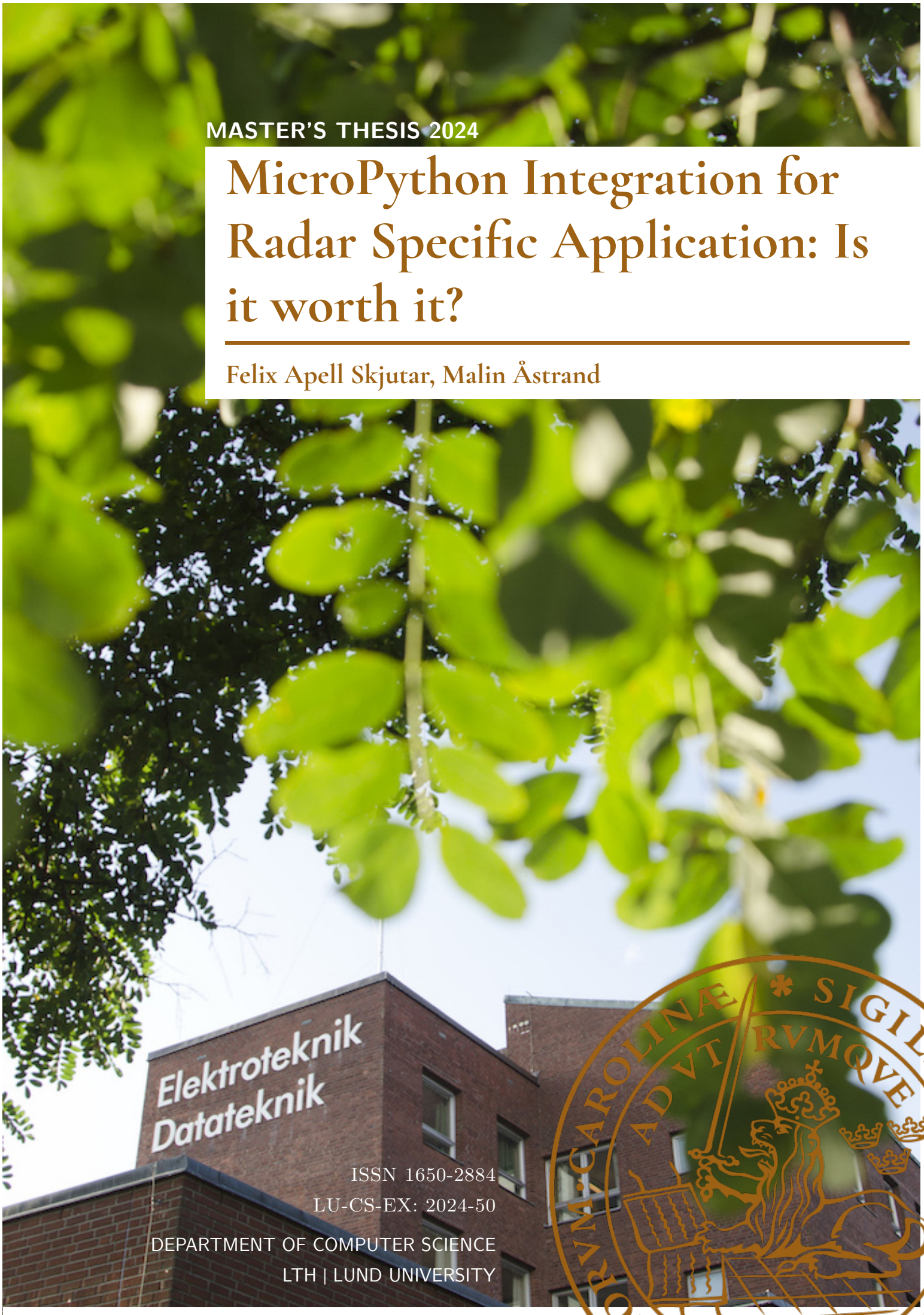

MASTER'S THESIS 2024

MicroPython Integration for Radar Specific Application: Is it worth it?

Felix Apell Skjutar, Malin Åstrand

ISSN 1650-2884
LU-CS-EX: 2024-50

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-50

MicroPython Integration for Radar Specific Application: Is it worth it?

MicroPythonintegration för radarspecifik applikation: Är det värt det?

Felix Apell Skjutar, Malin Åstrand

MicroPython Integration for Radar Specific Application: Is it worth it?

(Exploring Performance and Feasibility in Embedded Development)

Felix Apell Skjutar
fe7865ap-s@student.lu.se

Malin Åstrand
ma7566as-s@student.lu.se

August 8, 2024

Master's thesis work carried out at Acconeer AB.

Supervisors: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Anders Buhl, anders.buhl@acconeer.com

Examiner: Sven Robertz, sven.robertz@cs.lth.se

Abstract

In contemporary embedded development, speed and ease of prototyping are crucial. However, the prevalent use of C in this domain not only poses challenges for those less acquainted with the language but also increases development time. Python, renowned for its readability and ease of use, emerges as a compelling alternative. With libraries like NumPy and SciPy at its disposal, Python offers a rich ecosystem that supports efficient coding practices. MicroPython further extends this accessibility to the embedded realm, enabling swift development cycles similar to traditional Python. This thesis investigates the feasibility of implementing a radar-specific application in MicroPython, lowering the threshold of going from idea to prototype. While previous findings indicate MicroPython's sluggishness when performing computationally heavy tasks, we explore a real-world application leveraging Ulab and a radar API written in C. Our study contrasts the performance of MicroPython against its pure C counterpart, implementing a surface velocity algorithm. Our approach, with optimized algorithm design and data handling strategies, showcases a 40% improvement in performance compared to the naive implementation, albeit still trailing C by a factor of three. Our findings underscore the significance of thoughtful algorithm design and data management in mitigating MicroPython's performance disparity with C. Finally, we present a practical guide to aid decision-making regarding MicroPython adoption in embedded applications.

Keywords: MicroPython, C, optimisations, execution time, memory management, power consumption, Ulab, embedded development, Radar

Acknowledgements

We would like to thank our supervisors Anders Buhl and Jonas Skeppstedt for supporting our ideas and answering our questions. We would also like to thank Fredrik, Per, Karin, André, Johannes, Henrik, Johan and Martin from the software team for taking care of us. We wrote you all a poem.

Anders and Jonas, mentors true and strong,
Fredrik's aid, a melody in our song.
With their guidance, like a python's warm embrace,
We navigate radar's realm, with limited disk space.
In MicroPython's world, we dream in bytes,
With microcontrollers, true innovation ignites.
Thanks to Acconeer's team, a supportive array,
We're finally done, hip hip hooray!

Contents

Chapter 1
Introduction

1.1 Context

In many companies, Python is the natural choice when creating demos or proof of concepts. A reason for this is that it is easy and fast to write in. Another reason is the widespread competence present among a variety of people. Python is used in many fields within the scientific community, as well as in finance and education, and it has a reputation for being a good beginner's language. On the other hand, C might not be on top of the list when thinking of languages to create quick demos in. It lacks the abstractions and ease of use provided by Python, and one could argue it demands more effort from the developer. In addition, it could be assumed that competence in C isn't as widespread in non-computer science-related fields, as is with Python. Nevertheless, it is an important programming language, especially within the embedded community, where it is often the natural choice. Because of the limitations on memory and computational power in embedded systems, development is confined to a select few programming languages, and as a consequence of this, embedded development frequently involves complexity and boilerplate code, which is abstracted in scripting languages, such as Python.

At Acconeer, Python is used by the algorithm developers to create example applications that demonstrate different use cases for their radar sensors. Due to limitations of Python and the targeted embedded hardware, this part of the development is done using a digital environment connected to the hardware through a PC. Once the example is created and tested, the program is translated by another team with C competence to run on their embedded devices. As of now, C has been the only supported language to write applications in on their embedded devices. However, there is an ambition to make the process of creating freestanding demos, running directly on the embedded device, easier and faster, as well as to make it available for Python developers that are lacking knowledge in C. By bringing scripting languages such as Python to microcontrollers, the complex and time-consuming parts of embedded devel-

opment can be reduced and the prototyping of products become much more accessible for professionals and hobbyists. For this purpose, MicroPython, an optimised Python version made to be used on microcontrollers, was chosen as a candidate to be explored for future use in Acconeer's embedded device.

1.2 Problem Statement

Using MicroPython, we hope to show that it is possible to write real-world applications, utilising signal processing algorithms on large datasets, in an interpreted scripting language on microcontrollers. By comparing our implementation of a radar application algorithm to production code written in C, we investigate if it is feasible to prototype products in MicroPython. The focus of the comparison will be execution time, memory usage, and power consumption.

MicroPython, a project propelled by community efforts and with limited documentation, might be challenging for newcomers to learn and troubleshoot. Without comprehensive documentation or alternative resources, individuals unfamiliar with the platform might struggle to develop optimised code that will run efficiently on a microcontroller. We aim to investigate different optimisation techniques and construct a guideline with our learnings for others to use when considering building their own MicroPython program.

1.3 Contribution

Previous studies, like those conducted by Plauska et al. [14] and Ionescu et al. [9], have investigated MicroPython's performance on embedded systems across various popular algorithms, comparing it to other popular and emerging languages used in embedded systems. They found that MicroPython is most suitable as an entry-level language for students, but argued that it is lacking in performance. Furthermore, Wurl et al. [16] found that MicroPython has a use case as a control language for nano satellites. Our study aims to contribute to previous experiments by applying MicroPython in a radar sensor application and comparing it to production code written in C. We hope to combine previously found knowledge and evaluate the feasibility of using MicroPython as a prototyping tool, assessing whether its performance is sufficient for practical applications. Furthermore, we complement the existing documentation by gathering our experience from this project into guidelines on how to optimise a MicroPython program.

1.4 Distribution of work

For each part of the process, we have discussed and divided the work to achieve an equal workload. Felix had a bigger focus on execution time, whereas Malin focused more on memory aspects. We were both responsible for reviewing the other person's code. In the writing

process, Malin handled most of the data and produced the figures, while Felix took a larger responsibility for the accompanying text.

Chapter 2
Background

2.1 Acconeer

Acconeer, based in Malmö, Sweden, develops Pulsed Coherent Radar (PCR) sensors with millimetre precision and low power consumption. A PCR sensor transmits signals in short pulses with a known starting phase. The signals are reflected by objects and the elapsed time between transmission and reception of the reflected signal is used to calculate the distance to the object [4]. Transmitting short pulses instead of continuous waves allows the sensor to be turned off when using the received data. Measurements can be conducted only when needed and in the meantime, the sensor is not draining the connected energy source. Due to their small footprint and power efficiency, Acconeer's products are being used in businesses such as automotive, IoT, and agriculture.

In our study we used the A121 sensor, with picosecond time resolution and sub-millimetre accuracy [1]. The A121 sensor is often found in products where power consumption, memory (hard disk drive (HDD), random access memory (RAM), Cache) and processor power is limited, which must be reflected in the code running on the machine. Along with their sensors, Acconeer develops a software library, Radar System Software (RSS), which handles communication with the sensor for configurations and data gatherings, etc.

2.1.1 Example Applications

Acconeer provides example applications, written in both C and Python, in order to demonstrate use cases for the sensor. These examples include Hand Motion Detection, Vibration Measurement, and Surface Velocity, to name a few. The surface velocity example was chosen as a base for the MicroPython demonstration, and because of this, it will be described in

further detail in the subsequent section.

Since the Python examples can not run directly on Acconeer's hardware, they need additional software in the form of the Acconeer Exploration Tool. This software runs on a Windows or Linux machine and interacts with the sensor through a server, which gets flashed onto the Acconeer hardware. The server provides the application with radar data, which in turn runs the data through different algorithms to calculate distances, speeds, etc. Since the application is running on a PC and not an embedded system, performance does not need to be of the highest priority and Python is chosen for its ease of use, fast development time, and support for libraries like SciPy and NumPy.

The C example applications are made to be flashed and executed directly on the Acconeer hardware. These examples mirror the algorithms featured in the Python examples to the best of their ability, however, considering the nature of the languages, this is not always possible. Some typical Python features, like list slicing, are not supported in C, which means some logic in how the data is processed is changed. The NumPy and SciPy functions used in the Python example have been translated to C but made minimally to just fit the Acconeer algorithms. The C examples also contain some optimisations done to accommodate the memory constraints imposed by the hardware.

Surface Velocity Algorithm

We used the surface velocity algorithm [3] to test how well MicroPython fits to be used in embedded radar applications. This algorithm is chosen due to its use of complex data manipulation, signal processing algorithms (such as Fast Fourier Transform) and high requirements on processor and memory usage.

For this application, the sensor is placed above the running fluid, directed towards the surface at an angle, as shown in Figure 2.1. Fluid running towards the sensor is seen as negative flow, while fluid running away from the sensor is seen as positive flow.

The sensor data is received as a matrix of complex values, a frame, containing several sweeps of data. The frame is added to a time series. The Power Spectral Density (PSD) is calculated using Welch's method with a hann window. This means the data is split into segments and smoothed by multiplying it with the hann window. The segments are then fourier transformed and the square magnitude is calculated on the results. Finally, all segments are averaged. After Welch's method, the algorithm finds all valid peaks in the PSDs. The peak with the greatest energy is used to estimate the surface velocity of the target.

The code implementations of the algorithm feature a main function that configures the setup. It also features a loop that calls the *measure* function that collects radar data, this is followed by a call to the *process* function that does all of the data processing which returns a calculated velocity. To get a deeper insight into how the process function works, see appendix A.1.

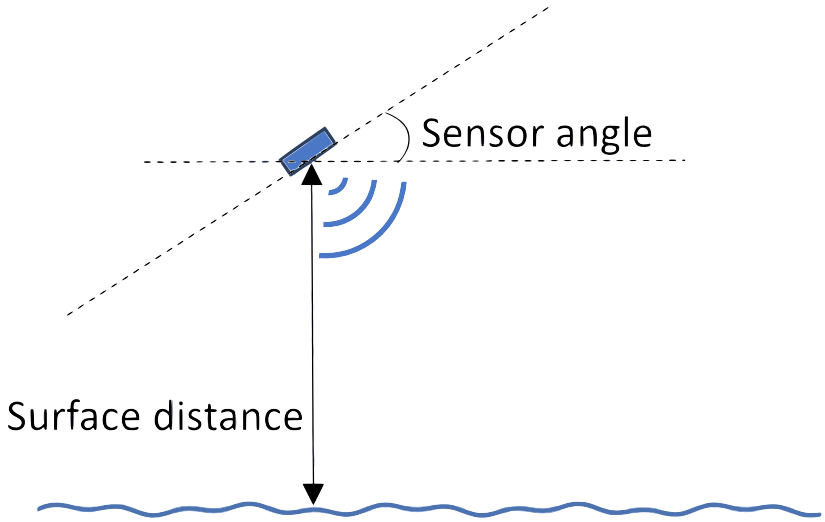

Figure 2.1: Illustration of how to use the surface velocity application by Acconeer [3].

2.2 C

C, a popular programming language developed in the early 70s, is a weakly typed and compiled language. It is compact and versatile, which has led to it being used in an array of industries. Despite being decades old, it still ranks highly in popularity, and according to the TIOBE index, C is ranked as the second most popular programming language [15]. It is often considered the most appropriate language for embedded applications due to its efficiency, low-level control, and portability. It provides the possibility to directly manipulate hardware peripherals, as well as to access memory addresses using pointers directly.

2.3 MicroPython

MicroPython is an implementation of Python for microcontrollers and constrained systems. It aims to be compact and efficient. Even though it includes only a subset of the Python standard library, it is possible to bring Python code directly to a microcontroller through MicroPython [12]. It's developed as open-source software (OSS) and is available on GitHub [7]. MicroPython offers built-in functionality to run directly on several ports (Unix, nRF, esp32, etc.) and also the opportunity for users to add their own preferred port or submodule with extensions to the MicroPython library. MicroPython is written with the same syntax and with the same file extension as Python `.py`. MicroPython is compiled with the MicroPython compiler to un-optimised byte-code in `.mpy`-files, which is interpreted by the MicroPython interpreter directly on the microcontroller unit (MCU). Apart from just running bytecode, MicroPython offers the possibility to flash an MCU with the compiler, interpreter, and a Read–eval–print loop (REPL), making it possible to write code in real-time directly on an MCU through a communication medium.

Contrary to C, (Micro)Python is a dynamically typed, interpreted language that uses a garbage collector for memory management and offers limited access to low-level structures such as pointers and control of data types. Python has a large standard library containing data structures and concepts, such as classes and list comprehensions, that are not available in C. By its nature Python generally performs worse with respect to computation time and memory utilisation than compiled languages such as C and C++ [5]. According to previous work, this is also true for MicroPython [14] [9]. These papers show that for certain algorithms MicroPython performs worse than other languages used for embedded development, which is expected. However, none of these utilises the acceleration of user modules such as Ulab (as we will see in section 2.5). We believe this approach is not true to the practical use cases of MicroPython and attempt to prove that the execution of MicroPython on embedded systems can be considerably better than shown in these studies.

There are several reasons why Python generally would perform worse than C, but as always, performance is very application-specific. Following are some differences between C and Python that could affect performance:

Execution C is compiled, while Python is interpreted. Compilers often include several optimisation techniques to speed up the execution of source code. The same optimisations could be impossible to perform in an interpreted language.

Garbage collection Python runs with a garbage collector that has to dynamically find unused allocations in memory and remove them to make space for new allocations. C has no garbage collector, the memory management is placed in the hands of the developer, who has to allocate and free memory "by hand". The lack of a garbage collector introduces errors such as memory leaks but also allows controlling how much memory is being used.

Types In Python you can't declare types of variables or parameters, which is obligatory in C. Not declaring types can ease the process for beginners, but it also means that you can't limit a variable to a certain size. From this follows that a Python program will generally occupy more space in memory compared to C for the same operations. In C there exist different types of different sizes, a `int16_t` for ints of 16 bits or `int32_t` for ints of 32 bits are two examples. Using these sized types in a controlled manner can reduce the footprint in memory

2.4 Heap Memory Managment

C

C provides the user with control over what is added or removed from the heap. Memory can dynamically be requested during runtime, and these allocations will then persist until they are asked to be freed. This means the user must keep track of the "in use" status of objects on the heap to not run out of memory or do illegal memory accesses. However, it provides flexibility and can lead to a more efficient program, since memory allocations and frees are only done when needed.

MicroPython

Unlike C, MicroPython utilises a garbage collector. This means that the memory management is hidden from the user and allocations and frees are done automatically. When memory is needed on the heap, the MicroPython garbage collector will do a scan to find unused blocks it can free. The free/in use status of the blocks are stored in a special bitmap, for faster scanning [11]. The search is done using a mark-and-sweep method, which starts by traversing the root set. The root set consists of objects referenced on the stack, that are currently in registers or are pointers to peripherals. These are set to `in use`. All other objects referenced by the objects in the root set are also marked as in use. Once the marking stage is done, the garbage collector performs a sweep stage, where all objects not marked as in use are cleared.

Garbage collecting reduces the workload for the user, however, it adds execution time and can lead to memory fragmentation. Memory fragmentation occurs when free memory blocks are scattered throughout the heap, leading to inefficient memory usage. This typically happens when memory is allocated and freed in such a way that the available memory is divided into small, non-contiguous blocks. Too much fragmentation causes problems since there could technically be enough memory for a new allocation, however it is spread out in smaller segments throughout the heap. One common way to solve this issue is to include a moving stage. This means that the garbage collector will push existing in use allocations together, which leaves all the following memory blocks free. This creates more overhead since the memory addresses stored in the object pointers have to be updated. Due to the MicroPython garbage collector (GC) occasionally mistaking primitive values for pointers, the GC has no way of moving memory allocations on the heap, to avoid accidental overwrites of variables in use [13]. This means MicroPython users still have to be mindful of how and when they create objects to avoid fragmentation. One way could be to reuse variables, however, this means that the scope of the variable will be extended, which blocks that memory region from being garbage collected and used for something else in between. MicroPython gives the user the possibility to control the garbage collector in several ways. It is possible to add memory usage thresholds for when the garbage collector should collect. It is also possible to turn it off altogether and manually ask for collections.

2.5 Ulab

Ulab is a NumPy-like array manipulation library. It is compatible with MicroPython and it's derivative CircuitPython. We use Ulab in our project to be able to execute more complex operations on large sets of data in C, through the comfort of Python syntax.

Like MicroPython, Ulab is a community-driven, open-source software that is not fully compatible with NumPy or SciPy. Though lacking some functionality, Ulab is a starting point for MicroPython developers working with, for example, data manipulation. Ulab enables users to easily include complex algorithms and data manipulation techniques that are faster compared to plain MicroPython. Using Ulab in a project will not only accelerate the execution of the code but also provide a more *Pythonic* way of writing Python. In Python, libraries such as NumPy and SciPy are widely used when handling complex data structures and algorithms. With this in consideration, Ulab brings MicroPython closer to the feel of ordinary Python.

2.6 Profiling

When measuring the performance of software, there are several approaches and techniques to be used. We are interested in execution time, memory management, and power consumption and will therefore focus on suitable profiling methods. Since embedded modules are very minimal and don't run superfluous processes such as a graphical user interface (GUI), power consumption can be measured as the total energy over the whole unit during execution. For measuring execution time, one can use language-supported methods reading values from the processor or an integrated real-time clock, RTC. Along with execution time, tools such as *valgrind* can be used to analyse the total instruction count or cache misses. When measuring memory there are different techniques to consider if measuring stack or heap usage. Following is a description of techniques used to measure memory utilisation.

2.6.1 Memory

Memory is sometimes a limited resource in embedded systems, which has to be reflected in the software running on these devices. Potential risks when running a program are allocating the whole heap memory causing overwrites or errors when running, or filling up the stack with instructions, causing a stack overflow stopping the execution of the program.

Heap usage

The MicroPython heap is allocated as one big block at the start of program execution. It is then up to the garbage collector to ensure that unused memory is marked as available within this block. MicroPython provides a built-in function that shows heap block usage at that given moment. There is no built-in function for monitoring maximum heap usage during an execution.

Acconeer provides a tool for heap tracking. It works by registering whenever memory for an Acconeer-implemented object is being allocated for or freed. By doing this, the tool can keep track of the maximum heap usage of the C application.

Stack usage

One common way of analysing stack usage in embedded systems is with stack painting. This is done by initialising the stack area with a repeated pattern, as shown in Figure 2.2. Once the program has finished running, it is possible to examine the stack and see where the pattern ends. This is a high watermark and indicates the maximum stack usage during runtime. This technique is used by both FreeRTOS and Zephyr RTOS to measure stack usage [6][17]. One downside to this dynamic approach is that it will not give any information on worst-case stack usage. Since it is a runtime-based analysis, it will only show stack usage for that specific execution path. Therefore, one would have to exercise all possible execution paths to get the full picture.

Acconeer provides a function for checking maximum stack usage which is based on stack painting. MicroPython provides a function showing stack usage during runtime. This works

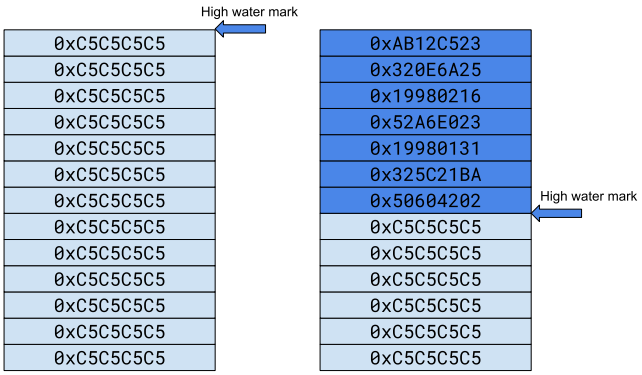

Figure 2.2: Representation of stack painting in memory. A high watermark represents the value read as the max stack usage.

by keeping a reference to the top of the stack, declaring a dummy variable, and checking the distance between the two memory addresses. It is possible to separate the Python stack from the C stack. If enabled, the Python stack will be allocated on the heap instead. The function will then return the stack usage of the Python code.

A different approach is to analyse the stack statically. This can be done either with tools analysing binary or assembly code or at compile time. GCC (GNU Compiler Collection) gives the option to compile with `-fstack-usage`, which generates a `.su` file that contains information on stack sizes of the target functions. The benefit of static stack analysis is that it shows the maximum stack usage of all the functions and not only those on a specific execution path. However, this method cannot accurately determine stack usage for operations whose stack usage is determined during runtime, such as variable length arrays and recursive or indirect function calls.

2.7 Optimisation

When optimising code there are different aspects to take into consideration and different techniques that might not work together in a final product. When working with MicroPython there are some special design choices to take into consideration due to its effect on execution time and memory usage. If the goal is to minimise execution time one would want to write most of the data handling functions in C and if you focus on development time you could let the complicated parts of the program be written in MicroPython.

MicroPython provides a guide in their documentation on how to optimise a MicroPython program for execution time [10]. Many of the optimisations described below would be done automatically by a C compiler at compile-time. However, for a MicroPython user, these have to be implemented manually. Following is a quick summary of the recommended optimisations presented by MicroPython.

Algorithm optimisation Optimising your algorithms is an essential part of optimising memory management and execution time of your code.

RAM allocations and buffers Creating objects one time and then reusing them rather than creating new ones, wasting time allocating memory. Avoiding appending elements to lists, since this will trigger new allocations and potentially garbage collections.

Memoryview To avoid creating copies of objects, for instance when passing slices of arrays to functions, one can use Memoryview which is a small, fixed-size object that points to a slice of an object.

Native code emitter By using the native code emitter, the MicroPython compiler will emit native opcodes, i.e. machine code, instead of bytecode. The bytecode would be executed by the MicroPython virtual machine, whereas the native code is executed directly by the central processing unit (CPU). MicroPython claims that this is roughly twice as fast [10].

Viper code emitter Similarly to the native code emitter, the Viper code emitter will produce machine code. In addition to that§, it will also perform further optimisations related to integer arithmetic and bit manipulations. It also provides support for the use of pointers. It is however not fully compliant with all Python code.

Caching Preloading library functions or object variables into local variables can reduce the look-up time when for example iterating over a data structure.

Constants In MicroPython it is possible to use global constants of an integer value that are optimised during compilation. This is done by wrapping the value in `const()`. This works similarly to the `define` keyword in C.

Another optimisation technique not recommended by MicroPython but commonly used is loop unrolling. This technique is used to increase speed by eliminating some of the instructions that control a loop and the branch conditions on each iteration. It is done by expanding the iterations of the loop, i.e. repeating the statements inside the loop. Often, loop unrolling is done automatically by the compiler. For the GCC compiler, loop unrolling is done at optimisation level O3 [8].

Chapter 3
Method

In this chapter, we will describe the method used to implement a working MicroPython implementation of the surface velocity algorithm and what measures we took to optimise our solution. Further, we will explain which metrics we chose and how we measured our results.

3.1 Hardware

We have used the XM126 radar module on an XB122 breakout board. The XM126 is a reference module that is built with Nordic nRF52840 System on a Chip (SoC) and A121 Pulsed Coherent Radar (PCR) sensor. The nRF52840 SoC is equipped with a 32-bit 64MHz Arm® Cortex®-M4 CPU, 256 KB SRAM, and 1 MB flash memory [2].

For the standard nRF MicroPython port, the stack is given 8 kB of RAM, and roughly 247 kB is left for the heap. When running our application on hardware, we compile our MicroPython files before flashing onto hardware using a *frozen manifest*. This means the files will be frozen into the firmware at deployment and there is no need to load the Python code separately into a filesystem. Compiling before flashing eliminates the need to include the MicroPython compiler and REPL.

To build and deploy we used the nRF makefile available in the MicroPython repository. We included the path to our own C modules by setting the `USER_C_MODULES` variable.

3.2 Extending MicroPython

We have used MicroPython v1.22.1 on the nRF port for an XM126 board. We have also implemented a sub-module that enables the Acconeer C-library to run through MicroPython. This

sub-module can be described as a translation layer between C and MicroPython, represented as the `c_mp_layer.c` block in Figure 3.1. In the translation layer, pointers to for example functions and data structures in C are wrapped in special MicroPython object-related types. By wrapping pointers and not the value directly, we can save the time and memory of copying each value or struct when traversing between the layers. With our objects and functions finished, our submodule allows communication and data to flow between the Python and the C layers.

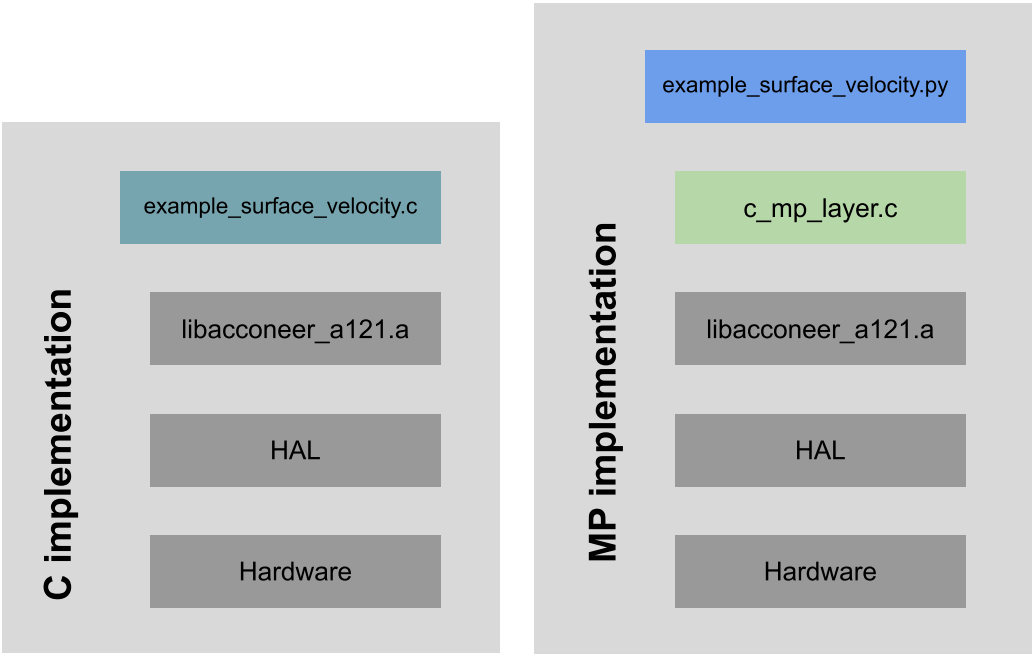

Figure 3.1: Visualisation of the code stack of C and MicroPython implementation respectively. HAL stands for hardware abstraction layer.

When extending MicroPython we define the specifics of the board being used. This includes a pin mapping, general board information, and a configuration file. When extending MicroPython with a new module, code is written in C or C++ to enable using certain features from the C scope in the MicroPython scope. All C code in this project, including the reference C implementation, was compiled with optimisation level O3.

3.2.1 Ulab

From the Ulab library we extensively used the `NDarray` data structure and its accompanying methods. These allowed us to handle large data sets effectively. NDarrays can also be used to construct a `memoryview`, which is used to reduce the need to copy data in memory. This can reduce both the size in memory and runtime of a program. Since the *welch* function is not included in the Ulab library, we constructed our implementation using the `ulab.utils.spectrogram()` method. The Ulab function does a fourier transform, as well as calculates the squared magnitude. It does not split the data into segments and multiply them with the hann window, and it also does not calculate averages over the segments.

3.3 Optimisation

When implementing the surface velocity algorithm we took a naive approach and made it as similar to the existing Python implementation in Acconeer Exploration Tool as possible, to make the starting point of our measurements as close to those of a real-life implementation as possible.

With a functioning implementation of the algorithm, we defined different techniques for optimising MicroPython. By implementing a version of the algorithm for each defined optimisation we could find the best techniques to use and finally combine them to make a more efficient version of the algorithm.

The final implementations used for comparison were the following:

Standard This is the reference implementation. It was made to look as similar to the Acconeer Exploration Tool application as possible.

All This implementation combines multiple optimisations. The final version was a combination of *Opt read*, *Native*, *Cached*, *Less lookup*, *Const* and *Opt alg*.

Opt read Three different optimisations related to data reading and storage were combined.

- **Data buffer** Unpacking of the data buffer was done more compactly and efficiently, using list comprehension and slicing instead of with a for loop and appends. The unpacking was also moved out of the main function and into the process function.
- **Memoryview** Instead of passing slices of objects to function calls, Memoryviews were utilised in applicable places.
- **Specifying data types** The NDarray keyword `dtype` was specified to int16 for all arrays containing data in integer format.

Native The Native decorator was added above all functions to emit native CPU opcodes instead of the universal bytecode.

Loop unrolling The inner for loop of the *welch* function was unrolled into four separate loops which meant the outer loop could be removed entirely.

Cached Certain repeated calculations used in the *welch* function, for instance, the hann window, were calculated once and then saved in the containing object to remove unnecessary calculations.

Less lookup Caching library variables into local variables in functions to reduce the number of expensive lookups.

Const Global variables were wrapped in the `const()` declaration.

Transposed The time series were processed in a transposed format compared to the reference implementation. This was done to avoid slicing over the matrix columns.

Opt alg The algorithms were looked through and optimised to avoid unnecessary steps and loops during the processing.

C This is the reference C implementation provided by Acconeer.

3.4 Profiling

To get comparable, deterministic results, prerecorded data was used. The data consisted of 34 radar frames that had been collected by recording flowing water. The data was copied into a buffer using `memcpy`. For the MicroPython implementation, this was done in the C layer, to have a similar effect on the performance as for the pure C implementation.

To validate the results of the data processing done by the algorithms, the recorded data was processed in the Acconeer Exploration Tool application. The values calculated by the reference applications were recorded and used to assure correctness. This was done similarly for both the MicroPython and the C implementation, using the `isClose` function, with a tolerance of 10^{-6}.

3.4.1 Execution time

Measurements of the MicroPython implementation were conducted using the `time` library function `timed_function` to measure the execution time of functions in the algorithm. Each different version of our algorithm, each with different levels of optimisation, was measured. To get an average, each program was executed 10 times. Measurements were done both for the execution time of the whole main function and separately for the execution time of each sub-function called by the main function to profile the program.

`timed_function` is a decorator function, a function that's applied to other functions to modify its behaviour. In this case, the target function is wrapped by time measurements using `utime.ticks_us()` and the time delta calculated by `time.ticks_diff()`. We choose to modify the built-in method to produce a dictionary containing all timed functions mapped to their time measurements during multiple runs of the function to be able to calculate a mean value of the function's execution time.

For the C implementation, we used the RTC clock of the `Zephyr` RTOS. In the same manner, as in the MicroPython case, we measured the execution time of the functions over 10 executions and calculated the mean execution time of each measured function.

Average execution times for the different implementations were also measured for the Unix port on a Linux x86 system. This was done similarly as for the XM126, however for the C implementation, we used the `clock()` function provided by the `time.h` header file, featured in the standard C library. To get a comparable idea of where the program spends most of its time, the average execution time of each profiled function was multiplied by the number of function calls.

3.4.2 Memory

When measuring memory we focused on the stack and heap usage of our program. In C we used functions from the Acconeer RSS to measure the stack and in MicroPython we implemented our stack painting function based on the same logic that was used in the Acconeer example. The stack painting was done in the C layer. The memory addresses for the start and end of the stack were read from the linker file. This area was then filled with a repeating pattern. At the end of the execution, the high watermark was evaluated.

For heap usage in the C program, we utilised Acconeer's heap measurement functions. For the MicroPython implementation, we had to use a different approach. During normal usage, the heap will only be garbage collected once it is close to being full. Because of this, we chose to manually control the GC during measurements. The approach was to request a collection before allocating a MicroPython object and then check the heap usage immediately after the allocation. This was done alongside all major allocations to find the place where the maximum amount of heap space was being used.

3.4.3 Power

To measure the average power needed by the different implementations, we used Joulescope and its accompanying software. The programs were executed 10 times and the power was averaged over the performances. The average time to process one frame was read from the waveforms and the energy needed was calculated using the formula $P * t = W$.

3.4.4 Instruction count

To measure the number of instructions generated by the implementations, Valgrind with the tool Cachegrind was used on a Linux machine. We ran the programs one time each with Valgrind and Cachegrind enabled and noted the generated statistics. The resulting output files were then analysed further using `cg_annotate` to get more detailed profiling data.

Chapter 4

Results

This section presents the results of C and MicroPython implementations. The execution times of the different implementations are presented in figures, both for the total execution time and also the execution time for each function of the algorithm, showing how different optimisations affect the execution time. All times presented are total times, i.e. an accumulation of all calls to that function during runtime. Instruction count, stack usage, and power consumption are presented in tables and figures show power consumption measured on the XM126 board running different implementations of the algorithm. In the presented results, a special focus was made on the *All*, *Standard*, and *C* implementations.

4.1 Time

This section presents the execution time of the different implementations of the surface velocity algorithm. Both the total execution time of the program and the execution time for each function in the algorithm are presented.

Figure 4.1 and Table 4.1 show the results from profiling each function for three different implementations. As can be seen in the graph, the *process* and the *welch* functions take the majority of the runtime. Furthermore, the difference between the *Standard* and *All* optimisations differ wildly between different functions. In *process*, the optimised MicroPython version runs almost four seconds faster than the standard version, while in the *welch* function, the optimised version runs slightly slower.

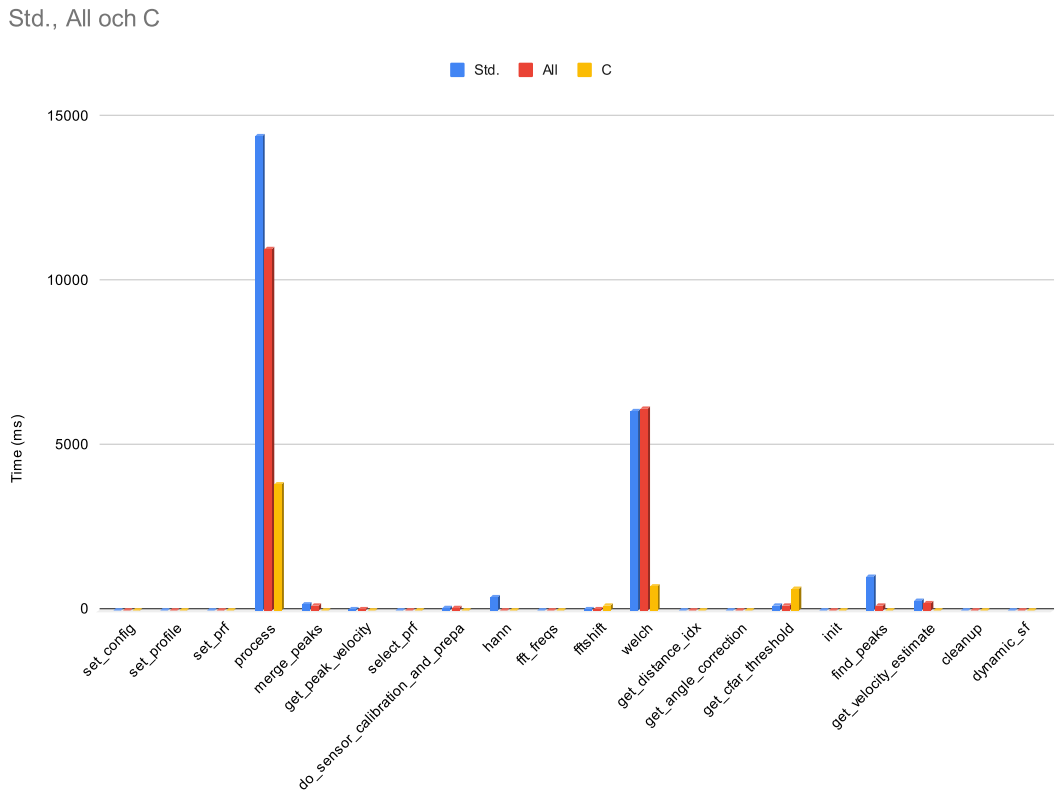

Figure 4.1: Execution time for each profiled function in the surface velocity algorithm. This graph identifies the most troublesome functions to focus on when optimising the program. *Process*, *welch* and *find_peaks* are the functions that consume the most time in our program.

Function	Std. (ms)	All (ms)	C (ms)
set_config	0.3784	0.2076	0.006484
set_profile	1.385	1.086	0.006109
set_prf	2.701	1.895	0.012265
process	14460	11030	3861
merge_peaks	190.6	166	0.4038
get_peak_velocity	54.99	44.25	0.1553
select_prf	2.121	1.526	0.001875
do_sensor_calibration_and_prepare	89.26	90.18	15.28
hann	408.2	9.558	0.4643
fft_freqs	1.862	2.197	0.02533
fftshift	60.46	59.5	176.2
welch	6064	6156	748.2
get_distance_idx	22.1	24.1	5.294
get_angle_correction	6.301	7.076	0.2295
get_cfar_threshold	155.6	146.3	669.4
init	21	25.6	4.311
find_peaks	1052	170.6	4.281
get_velocity_estimate	293.7	241.2	2.966
dynamic_sf	6.577	7.288	0.009554

Table 4.1: Execution time for each profiled function in the surface velocity application. *Process*, *welch* and *find_peaks* consume most of the execution time in the *standard* implementation.

In Figure 4.2 we can see that the execution time for the profiling gets affected differently by different optimisations. Notably, the *Opt read* optimisation made *process* run slower. The slower execution is expected since this version moved the unpacking of data from the main function to process.

In Figure 4.3 we can see that the *welch* function was barely affected by optimisations, it is only the *cached* optimisation that visibly made the algorithm run faster. *Opt read* performed worse since it contained an additional cast of an int16 NDarray to a float NDarray. Other functions were affected more by optimisations, such as *find_peaks* presented in Figure 4.4. As can be seen, both *Opt read* and *Opt alg* substantially reduced the running time of the function compared to the standard version of the code. In the end, it was optimised to run in less than a fifth of the original running time.

Figure 4.5 gives a closer look at the difference in running time of *find_peaks* for C and the fully optimised MicroPython implementation. Notice the logarithmic scale on the Y-axis. The MicroPython version runs almost 100 times slower than the C version.

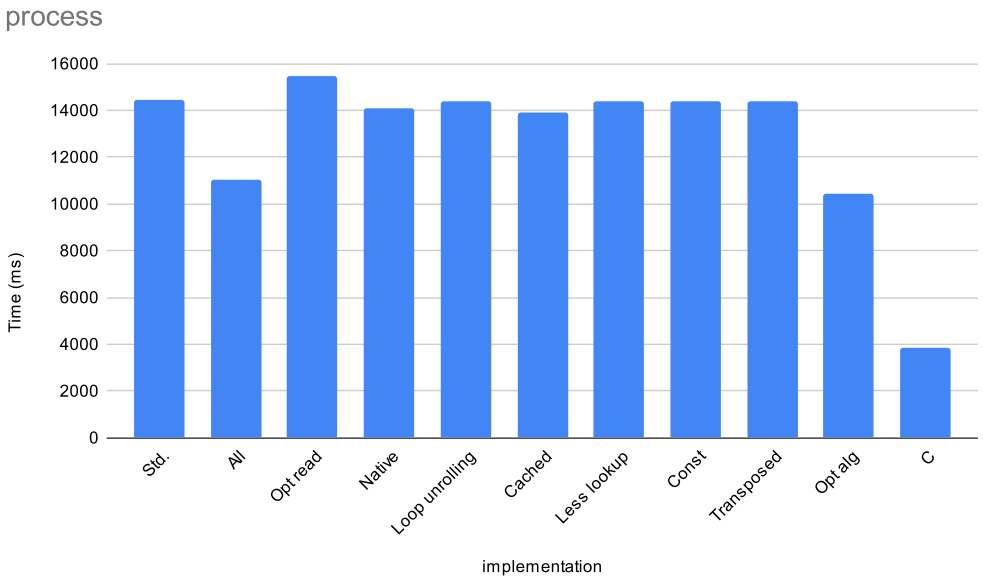

Figure 4.2: Execution time of the *process* function for different implementations. *Opt alg* and *all* performed well for this function while *opt read* performed worse than the *standard* implementation.

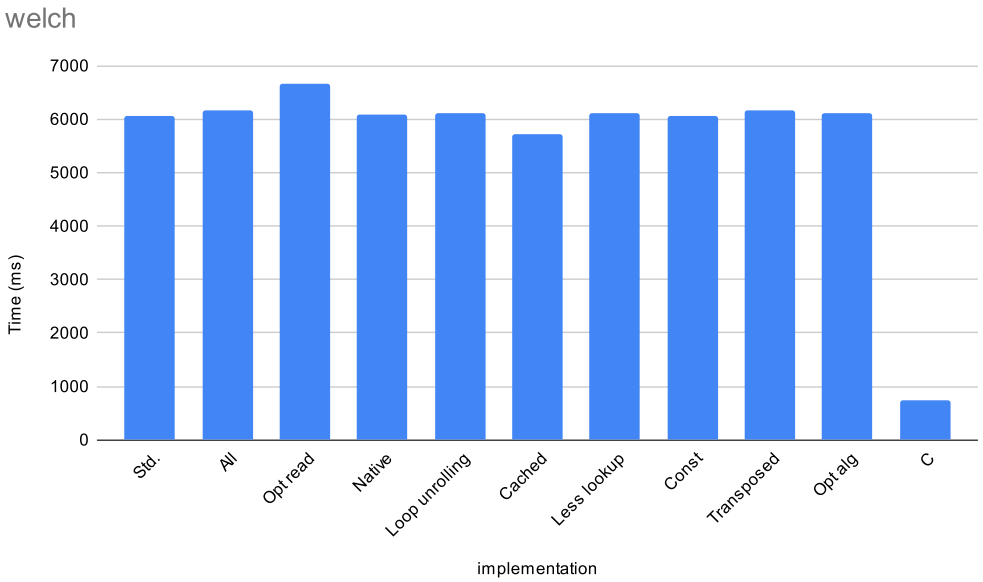

Figure 4.3: Execution time of the *welch* function for different implementations. *Cached* was the only version that performed slightly better than the standard implementation, while the execution time was roughly the same or slightly worse for all other implementations.

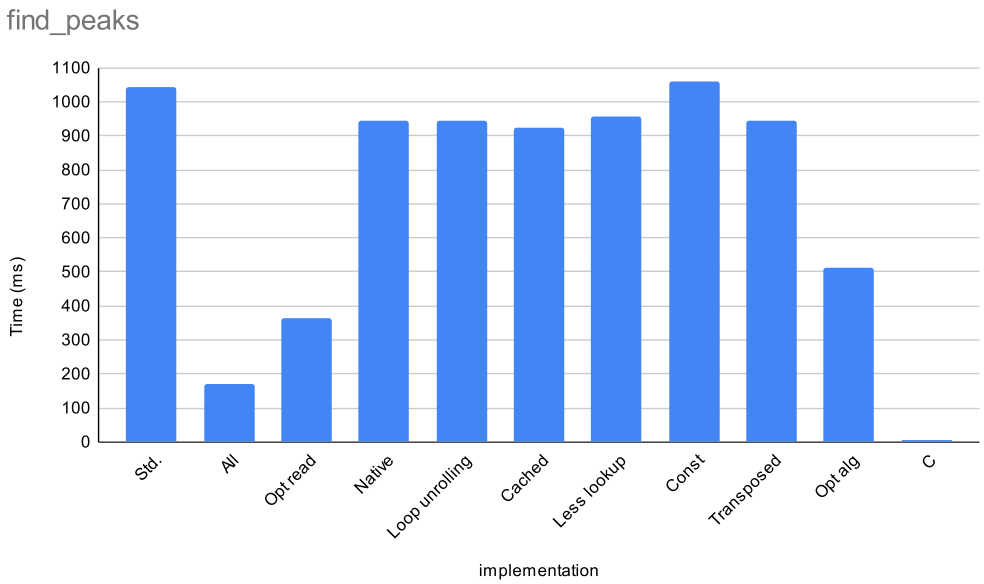

Figure 4.4: Execution time of the *find_peaks* function for different implementations. All implementations except *const* reduced the execution time for this function. *Opt read* and *opt alg* more than halved the execution time. At the same time, the C implementation is outperforming MicroPython in such a way that it is hardly noticeable in the graph.

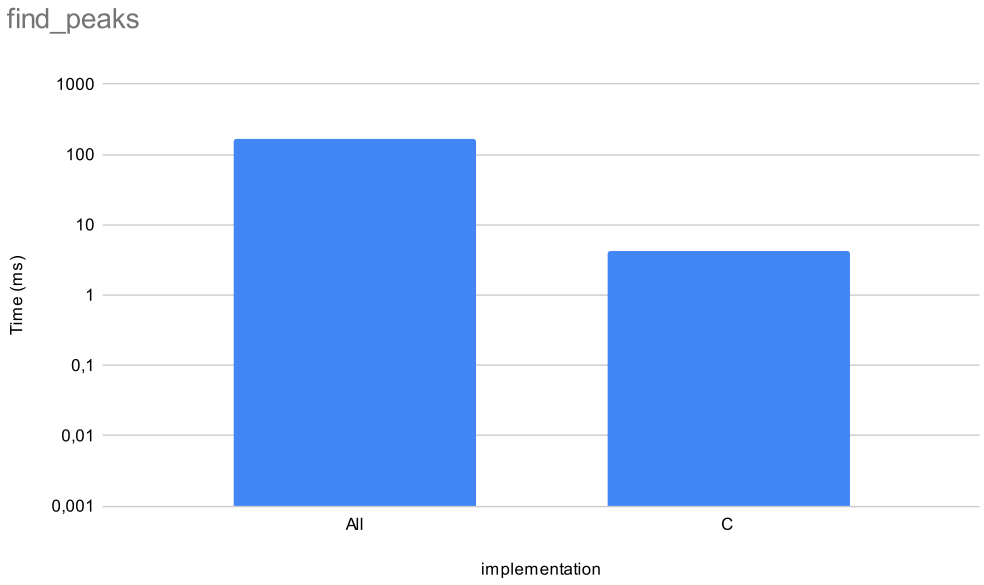

Figure 4.5: Average execution time for a whole frame of the *find_peaks* function for two implementations. Note the logarithmic y-axis.

In Figure 4.6 and Figure 4.7 we see total execution time for the entire program when run with the different optimisations. Besides the *all* implementation, *Opt alg* performs the best. This is evident on both the XM126 and Linux system.

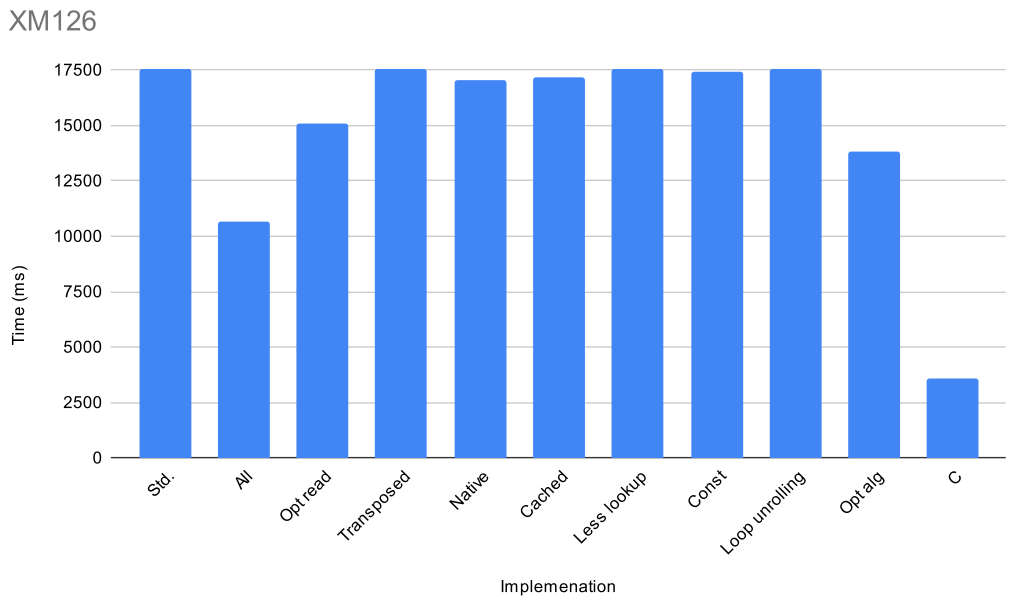

Figure 4.6: Total execution time when running one frame at different levels of optimisation on XM126

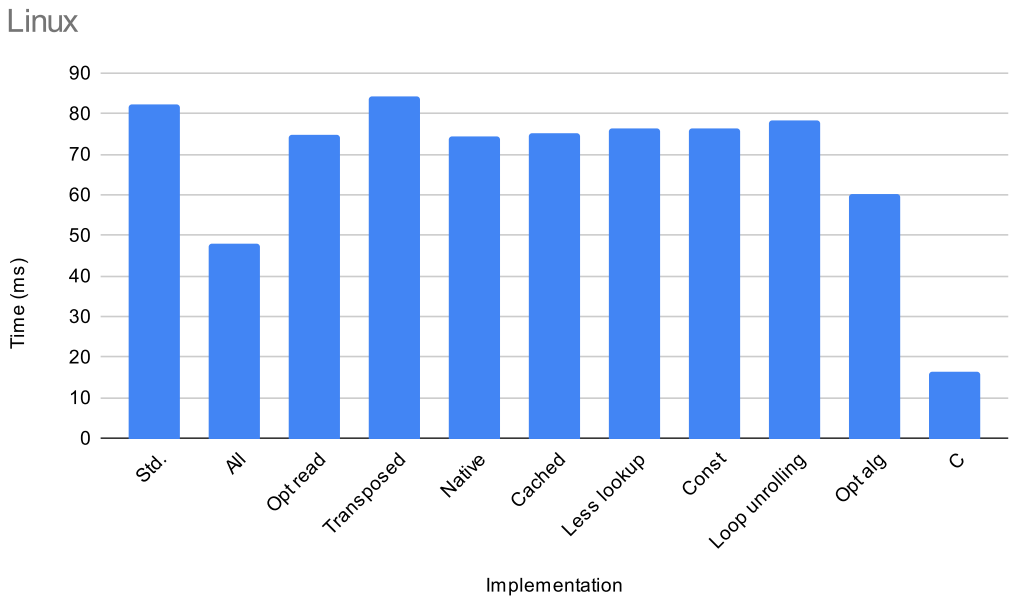

Figure 4.7: Total execution time when running one frame at different levels of optimisation on a Linux x86 system

4.2 Memory

We measured the stack and the heap usage of the *C*, the *Standard* and the *All* optimisations implementation in MicroPython.

Table 4.2 presents the measured maximum stack usage of three different implementations

Program	Max stack usage (B)
C	2448
Std.	2676
All	2732

Table 4.2: Maximum stack usage of three different implementations of the surface velocity algorithm. The *standard* implementation increased the stack size by approximately 2KB and the *all* implementation increased the stack size further by approximately 1KB.

of the surface velocity algorithm. The C program has the smallest stack, and the optimised MicroPython has the largest. Table 4.3 presents the total heap usage of three different imple-

Program	Max heap usage (B)
C	33624
Std.	155264
All	153728

Table 4.3: Max heap usage in Bytes of three different implementations. MicroPython introduces a large overhead for the heap usage.

mentations. The C implementation shows the smallest heap usage and the standard MicroPython implementation the biggest heap usage. From the table, we can see that MicroPython uses almost 5 times more heap memory than the C implementation.

4.3 Power

In this section, we present the results of power consumption measurements on the XM126 module when running three different implementations. The algorithm handles a lot of data, which results in new measurements being triggered as soon as the computations are done. Over time, the C implementation, which is running faster than the MicroPython implementations, will trigger more measurements per time unit, each consuming a lot of power, while the MicroPython versions spend more time in a lower power state, doing computations.

Program	Power (mW)	Time (mS)	Energy (nJ)
C	136.1	114	15.5
Std.	141.1	515	72.7
All	141.6	309	43.8

Table 4.4: Estimated power, time, and energy needed to process one frame. Both the decrease in time and total energy consumption for the *All* implementation is approx. 40%.

In Table 4.4 we can see that the power consumption to process one frame is pretty similar across all three implementations, but when considering the processing time, we see that the C implementation consumes less energy per frame than the MicroPython implementation.

By looking at the figures featured in 4.8, 4.9 and 4.10 it is noticeable that the peak caused by the sensor doing its measurement is higher for the MicroPython implementations. We believe the reason for this is how the `wait_for_sensor_interrupt` is implemented in the different programs. In the C implementation, the processor is allowed to go into sleep mode until the finished measurement causes an interrupt. For the MicroPython implementation, the processor instead busy waits in a loop.

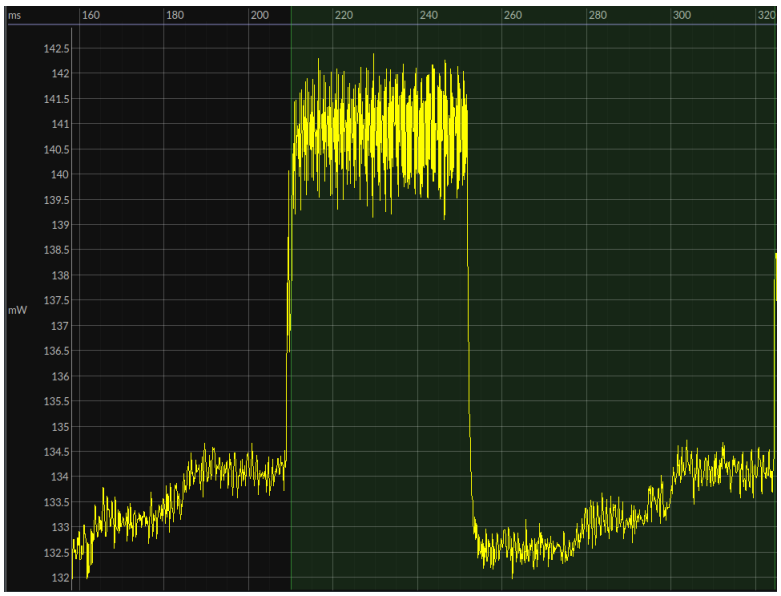

Figure 4.8: Power consumption during one frame for the C implementation. Elapsed time can be seen in the top row and consumed effect in the left column. It should be noted that this graph uses the time unit *ms* in contrast to *s* used in the other power graphs. The higher middle part of the graph represents the sensor doing measurements and the lower parts of the graph when the application is doing calculations between measurements.

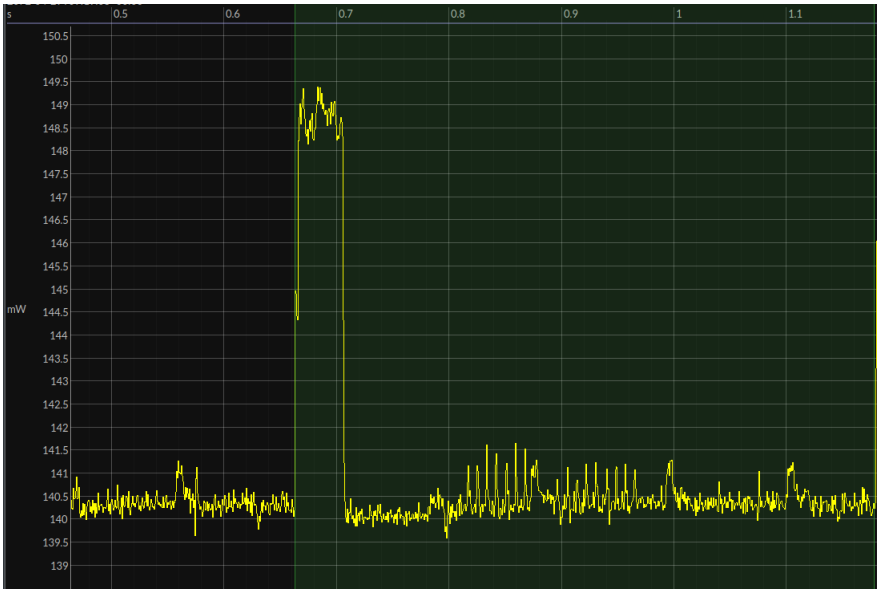

Figure 4.9: Power consumption during one frame for the *Standard* implementation. Elapsed time can be seen in the top row and consumed effect in the left column. The higher middle part of the graph represents the sensor doing measurements and the lower parts of the graph when the application is doing calculations between measurements

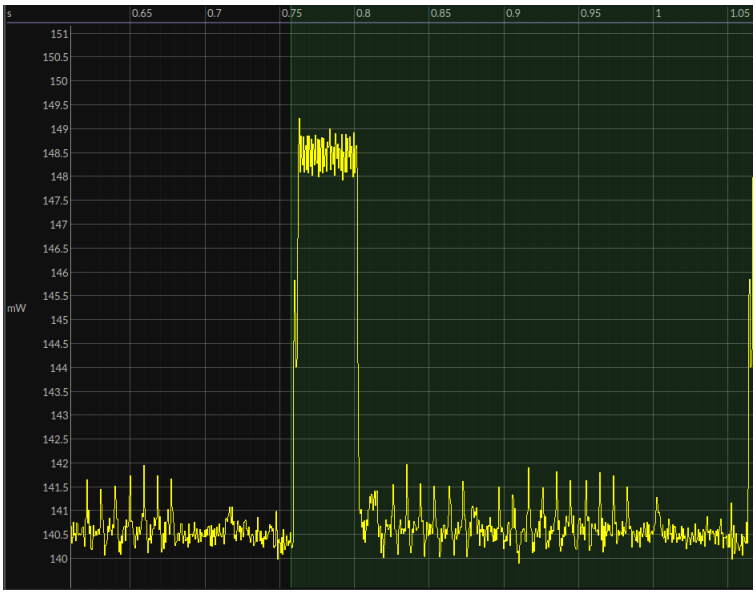

Figure 4.10: Power consumption during one frame for the *All* implementation. Elapsed time can be seen in the top row and consumed effect in the left column. The higher middle part of the graph represents the sensor doing measurements and the lower parts of the graph when the application is doing calculations between measurements

4.4 Instruction Count

From Table 4.5 we can see that the number of instructions executed is larger for the MicroPython implementations. The *Standard* implementation has roughly **4, 75** times the instructions of the *C* implementation, and the *All* implementation has **3, 03** times the instructions. If we compare this to the execution time, we get a similar result, where the *Standard* implementation is **4, 91** times slower and the *All* implementation is **2, 99** times slower.

Table 4.6 shows statistics of the instruction counts for the five most called functions in the *standard* and *native* implementations. We have used this data to analyse what impact bytecode has on the instruction count. This is done most effectively by looking at the *native* implementation which only affects this type of instructions without optimising other aspects.

The three functions that influence the number of instructions the most are the same for both programs, with similar percentages. The standard configuration of the Unix port enables multithreading which is why we see the use of functions like `pthread_mutex_lock`. This is not the case when running on the nRF port, where multithreading is disabled. The *standard* implementation features the function `mp_execute_bytecode` which is missing from the *native* implementation.

Program	Instruction count
C	141,957,489
Std.	674,307,185
All	429,782,708

Table 4.5: Executed instructions, as counted by Cachegrind, on a Linux x86 system. The *standard* implementation introduces a lot of overhead compared to the C implementation, while the *all* implementation reduces the instruction count by apprx. **36%** from the increase introduced by *standard*.

Std.	Native
gc_alloc (16.94%)	gc_alloc (16.06%)
gc_collect_end (10.50%)	gc_collec_end (11.00%)
pthread_mutex_lock (5.90%)	pthread_mutex_lock (6.18%)
mp_execute_bytecode (5.08%)	pthread_getspecific (4.48%)
pthread_getspecific (4.42%)	mp_binary_op (4.01%)

Table 4.6: Top five most executed functions and percentage of total instruction count for *Standard* and *Native* implementations, calculated by cg_annotate. *Native* eliminates the *mp_execute_bytecode* instructions.

Chapter 5
Discussion

In this chapter, we discuss the results and what we can conclude from them. We discuss time, power, and memory respectively, and try to answer questions that arose during the study.

5.1 Time

Some optimisations made certain functions slower, while at the same time reducing the overall runtime and were included in the final *All* implementation. One example of this is the *welch* function in the *Opt read* implementation. An `int16` to `float` cast caused the runtime of the *welch* function to increase, while, as evident in Figure 4.6, the total execution time was decreased.

Find_peaks was an especially interesting function to optimise since it didn't contain calls to any C-based function and could be considered a more or less pure Python implementation. This means the performance of this function is directly related to the efficiency of MicroPython itself. In *Opt read*, `memoryview` was utilised, which influenced the execution time substantially and could be considered a relatively easy thing to add to one's implementation. The execution time in *Opt alg* was reduced due to the possibility of removing all `is_nan` checks. These statements are present in the Acconeer Exploration Tool implementation since the *find_peaks* function is designed to be universal for all algorithms, however, it is not necessary for the surface velocity algorithm and they are not part of the C implementation. There is a lot of variance in the execution time for *find_peaks* in implementations where this function wasn't changed, for example in the *Transposed* and *Loop unrolling* implementations. This could be caused by normal variations in execution time. Despite *All* taking a fifth of the time to execute as the *Standard* implementation, it is still many times slower than the C function. This is shown especially well in Figure 4.5 where it is possible to see the difference on a logarithmic scale. This highlights the potential benefit of implementing computationally heavy algorithms in C.

The seemingly small performance benefit of using the native decorator was initially unexpected. As stated previously, MicroPython claims in their guide that this would potentially halve the execution time, compared to standard bytecode [10]. In Figures 4.6 and 4.7 there is a slight performance increase, however, it is not twice as fast. This could be explained by analysing Table 4.6. Here it is evident that the function `mp_execute_bytecode` makes up **5.08%** of the instructions of the Standard implementation. Naturally, it is not featured in the function list of the *Native* implementation at all. The performance increase could therefore be thought to come from the removal of these specific instructions. Since they make up a relatively small portion of the total program, the total execution time will not be as dramatically affected as initially thought.

When comparing Figure 4.7 and 4.6 it is possible to see that the optimisations gave a similar effect on the execution time for both XM126 and the Linux system. A benefit to this consistency is the possibility to more easily and swiftly experiment with different optimisations on the Linux system, and then expect similar results once deployed on an embedded unit. Running MicroPython locally on your PC is a great feature since it could save time in cases where you have more processing power on that machine than on your embedded device, and eliminates the hassle of flashing the MCU every time a change in source code is introduced.

For the measurements done on the Linux system, the *Transposed* implementation was the only one to perform worse than the Standard implementation. The idea behind transposing the data was to allow for slicing and calculations to be made row-wise, instead of column-wise. This can theoretically be more efficient since it is common for matrices to have row elements stored contiguously in memory, which can be utilised by caching the entire row, eliminating cache misses as the whole row can be fetched from the cache at once. However, this did not lead to any decrease in execution time. We think this could be caused by the way Ulab has implemented NDarrays, however, there was no effort to research this further. Another reason could be that the overhead of doing the transpose outweighs the benefit of exploiting spatial locality. There wasn't any benefit of transposing the data for the XM126 implementation either. This device does not have cache memory, however, there could still be a potential speed increase if the data is read in burst mode. The lack of impact could be explained by other accesses being made in between the row operations, which would remove the positive effect of burst mode.

Loop unrolling was only tried in the *welch* function. As stated previously, the outer loop was removed entirely, which meant the inner loop was repeated four times. In Figure 4.3 it is evident that loop unrolling had no significant effect on the performance of the program. In this attempt, only four loop condition checks per function call could be removed, which seems to not have been enough overhead removed to have any noteworthy effect on execution time. In an attempt to reduce the number of iterations of the inner loops, it was required to add additional list slicing. This resulted in worse performance, so this approach was abandoned. Although not effective as an optimisation in this case, loop unrolling could have a positive effect in other algorithms and is worth exploring if applicable. It should be noted that it is not necessary to, as in our case, unroll all iterations of a loop. If the number of iterations is very high, unrolling the loop only a handful of times could still have a positive

impact since this means the loop control instructions will be reduced to a fraction.

There was no significant performance benefit of the *Less lookup* implementation. A reason for this could be that when compared to the total amount of execution time, the time spent doing lookups is only a small fraction. Similarly, using `const()` had no major effect on execution time. This implementation is related to *Less Lookup* since they both are approaches to minimise lookup time, and the reason for the *Const* implementation not having any effect could be attributed to the same reason. To understand what kind of operations make up most of the execution time, we can look at Table 4.6. A significant amount of the instructions are related to heap management, in the form of allocations and garbage collections, which could indicate that the most effective optimisations would be related to that area.

The largest performance improvement was achieved by *Opt alg*, as can be seen in Figure 4.6. This highlights the need to be mindful of how the algorithms are implemented to avoid expensive and unnecessary steps. Similarly, *Opt read* also had a positive impact on performance since this reduced the need for some time-consuming allocations. Some algorithm optimisations done for *Opt alg* could also be applied to the Acconeer Exploration Tool example.

Some potential optimisation strategies were never implemented. The Viper code emitter was omitted entirely. The background to this decision was its noncompliance with standard Python code and its use of special types. For example, Viper supports different types of pointers and ints. An angle to this research has been to see if MicroPython is a viable choice when considering developing Proof of Concepts on embedded systems. Because of this, an important aspect to have in mind is accessibility and ease of use for a standard Python developer, and rewriting the code to take advantage of all the potential optimisations provided by the Viper emitter felt like too big of a step.

When measuring time, no consideration was taken into probe effect and the potential overhead created by the measurement itself. This could affect the comparison if it is faster to measure execution time in C than in MicroPython or vice versa, and should be taken into account when looking at the results.

5.2 Power

A conclusion to draw from the results of the power measurements is that execution time and power are closely related. Apart from the busy wait discussed previously, it is not possible to lower the energy consumed when the sensor measures. If this is considered a fixed consumption, the only place to implement optimisations is during the processing of the data. In Figure 4.9 and Figure 4.10 it is possible to see that the power level during the lower part of the graph, i.e. the processing parts, is relatively similar between the implementations. However, the time spent in that state differs, where the *All* implementation spends far less time processing. This results in lower overall energy consumption, which can be seen in Table 4.4. With that said, the C implementation spends even less time in that state and as a result, consumes the least amount of energy per frame.

Looking at Table 4.4 we can observe an approximately 40% decrease in energy consump-

tion by optimising for execution time. However, this could still be considered too small of an improvement if used in an energy-critical application where the application needs to be very low power. In that case, one would have to find ways to further optimise the MicroPython code, or switch to a better-performing programming language.

For this specific application, there are no hard limits on time. Doing data processing faster will result in more measured frames per time unit and a more continuous representation of the velocity. However, achieving more measurements per time unit is more resource-intensive since our power consumption is highest during a measure. On the contrary, if we let the sensor measure only once every fixed time interval and go into sleep mode once finished, we can save energy by doing the data processing fast. This highlights a trade-off between accuracy and energy, and the winner in both cases is the faster C implementation.

5.3 Memory

The results of the dynamic stack analysis are expected. The number of variables and function calls are relatively similar between the implementations, however, the MicroPython implementations contain an additional C layer necessary to make calls from the Python application down to the Acconeer radar functions written in C. This layer will add additional instructions on the program stack. There is a slight increase of the stack between the *Standard* and *All* implementations, as seen in Table 4.2. Because of the *Less lookup* optimisation, there are more local variables in the *All* implementation and this could be the reason for the increase in stack size.

Some efforts to analyse the static stack usage were made, however, did not result in any valuable insights. For this type of application, it was determined that there was no obvious risk in only using dynamic stack analysis, since the possible execution paths are straightforward and will be exhausted during normal execution.

There is a significant increase in heap usage between the C and the MicroPython implementations. One reason for the rise is the wrapping of C objects. Some objects are needed both in the MicroPython application and in the C layer and are therefore wrapped in a `mp_object` type which will put additional data on the heap. Furthermore, the MicroPython implementations contain class objects not featured in the C implementation. One example of this is the Processor object, which contains functions and variables needed in the processing algorithm. As previously mentioned, the XM126 unit was given roughly 247 kB of heap memory, which meant there was around 90 kB left at the moment of maximum heap usage for both the *Standard* and *All* implementations.

As presented in Table 4.3, it only differed approximately 2 kB between the *Standard* and *All* implementations, which could seem daunting for memory-critical applications. Optimisation was done with execution time in mind, a great difference in heap size was not expected. To decrease the maximum heap usage further, we could use only bytearrays and force collec-

tions of the garbage collector, which most likely would increase the execution time. Studying a minimal implementation in MicroPython is an interesting subject for future studies, and we hope our guidelines (see Chapter 6) can be useful.

5.4 Future Work

In the current implementation, an extensive C module featuring radar-specific functions, as well as the Ulab module, is used. To further utilise the performance benefit of C, additional functionality could be placed in the C layer. For example, the *Welch* function could be entierly implemented in C. This would make it more similar to the Acconeer Exploration Tool implementation, where the *Welch* function is a part of the SciPy library.

For this project, no regard for program flash size was taken, considering the substantial size of the nRF52840's flash memory. Due to the substantial size of the nRF52840's flash memory, no regard for program flash size was taken into consideration in this project. Program flash size could be an issue on hardware with memory constraints, and therefore could be an interesting topic for future research. Theoretically, there could be large variations in program flash size of the different builds, depending on factors such as Python files being emitted as bytecode or machine code, or if the Python files aren't compiled at all before flashing.

Another interesting aspect would be to do a user study to see how developers experience using MicroPython for embedded development. Developer preference could have a big impact on how feasible it is to involve MicroPython in the development process.

5.5 Conclusion

In this study, we implemented a radar-specific algorithm, containing the handling of large data sets and signal processing algorithms, in MicroPython, and optimised it with a focus on execution time. We found that the most effective optimisations were achieved by altering the algorithm and data handling. Combining all optimisations, we achieved to reduce both execution time and power consumption by approximately 40% and lowered the max heap size used by 2 kB. We show that with relatively standard optimisation techniques, it is possible to optimise a high level programming language for embedded use cases. The final MicroPython program did not measure with the C implementation, however, contrary to previous research, we suggest that MicroPython can be a part of the commercial market as a prototyping tool for embedded system development.

Chapter 6
Guidelines

When working with MicroPython there are different aspects to consider depending on what kind of project you are working on.

Hardware could make it or break it for your project. A good thing to start with is to decide what hardware to work on and check if an appropriate port for that platform exists. If not, is it feasible, given your budget, time, and knowledge, to port it yourself?

Memory What are your restrictions on memory utilisation? For some operations and depending on the application, it could be hard to minimise memory usage. In those cases, C, or some other high-performance programming language could be the only viable option.

Time If your application is time-critical, it could be good to start with implementing it in C and work towards MicroPython from there.

Power The energy consumption will most likely be higher. This could be a problem if the device is supposed to run on battery for a very long time.

Competence within the team will give you a hint of whether making the switch is worth it or not. If the application developers are senior C programmers, it could be more efficient to stick with C.

Although a subset of the Python standard library is integrated into MicroPython, many of the popular libraries used in Python are not. For example, NumPy and SciPy are not available in MicroPython. To solve this we have used the Ulab extension (see Section 2.5) to be able to use efficient data structures and algorithms, similar to NumPy and SciPy.

For us to build the example application, foundational work in C had to be done. This is because of the translation layer, written in C, that is needed to work with the Acconeer RSS

API. Once this is established, the applications could be written in Python. One has to be aware of this initial investment in writing C code before deciding on whether it is worth using MicroPython or not. It is possible to minimise the translation layer by rewriting the functions of the API in pure Python, however, this would demand an even bigger initial investment, and the performance would be even more lacking. We would recommend against this unless the project is starting from scratch. In that case, it could be worth using MicroPython to quickly create a working version, and then incrementally move performance-critical parts into C code.

When writing a typical Python program, memory management is seldom a problem, this changes when the goal is to execute the code on a MCU with highly restricted memory space. Regular Python data structures are pretty large and when considering the limitations of the garbage collection a lot of memory will be filled with data not in use. To decrease the footprint of your program, consider helping the garbage collector by deleting variables and restricting the copying of data structures when other means are available.

Furthermore, it could be hard to even reach the desired performance results in pure Python code. If this is the case one have to consider expanding the MicroPython library with an external user module, where one can add new functionality through writing C code.

6.1 Using MicroPython

Following is a brief description of how to get started using MicroPython and different techniques to optimise your implementation.

Clone the git repository from the official GitHub page.

Find the appropriate port for the MCU you are planning to use. Depending on what port you are using, you have to provide additional information such as board-specific configuration. If your port does not previously exist you have to supply everything from toolchain, boot configuration, and basic drivers for development.

Build external C modules if needed. External modules make it possible to call self-defined C functions from MicroPython. These can be used to run heavy calculations in C, communicate with the system through C, or build an API for existing C code to be used in MicroPython. To include your external modules, you add the path to your external modules to the `SRC_USERMOD_C` when building MicroPython with `make`.

Running MicroPython code on your MCU can either be done when building MicroPython, by stating the files to be compiled and included in a `manifest.py` file, or by sending it to the MCU over the chosen communication channel.

Including Ulab is done by cloning the official Ulab repository and including it on the build path, similarly to how you include an external C module.

When up and running with development, it is time to plan for optimisations. Optimisation is highly implementation-dependent and relies on budget, time, and application-specific requirements. Keep in mind that in this study, we have focused on reducing execution time and

power consumption. The following section would probably look different if we optimised to reduce memory footprint, and it is up to each developer to find the right method for their specific application. The techniques we found effective when working in MicroPython for our specific application are presented below.

Profiling your code is essential when optimising. Identifying the problematic areas of your code enables you to know where to start. It could be difficult to discern straight from the source code where the program will spend the most time; therefore, it is always a good idea to check even if you believe you have a hunch.

Optimising your algorithms is perhaps the most straightforward way to potentially reduce execution time significantly. Before attempting to optimise with complex techniques that may make the code intricate and restrictive, reworking the logic and algorithms is essential.

Using Ulab is recommended, especially when working with a lot of data or heavy computations. The optimisations achievable using Ulab depend on your implementation. The following is a description of what we learned from using Ulab:

NDarrays are extremely helpful when working with data and data manipulation. We found the accompanying data manipulation functions helpful, and the ability to specify the data type can greatly reduce memory footprint and execution time.

Using Ulab functions whenever possible is essential for reducing execution time and also operates very similarly to `NumPy`, which facilitates an easy transition.

Bytearrays are useful when a buffer is needed or generally when handling data minimally.

Memoryview objects are excellent for reducing the footprint when sending data between functions. They are small objects that point to a part or a whole section of data in memory. A memoryview can only be constructed from data structures implementing the buffer protocol, such as bytearrays, arrays, or even Ulab's NDarrays. Memoryviews can be sliced and passed as arguments to eliminate the need to copy large sets of data in memory when using them.

Caching certain repeated operations or variables from other namespaces can potentially reduce execution time by removing the overhead of redoing calculations or searching for functions in other libraries.

The native code emitter reduces overhead by compiling to machine code instead of bytecode. The effect of this technique seems to vary between different use cases and might not always significantly affect runtime. Thankfully, it is a low-effort optimisation, only requiring the use of the decorator function `native`.

Not all optimisations can be recommended, and we encourage people to follow the Zen of Python. If an optimisation makes the code more complicated or destroys the readability, consider not using it. An example of this is loop unrolling, which we would advise against.

References

[1] Acconeer AB. A121. https://developer.acconeer.com/home/a121-docs-software/. Accessed: 2024-04-09.

[2] Acconeer AB. A121. https://www.mouser.se/datasheet/2/1126/Acconeer_XM126_Datasheet-3367955.pdf. Accessed: 2024-04-09.

[3] Acconeer AB. Surface velocity. https://docs.acconeer.com/en/latest/exploration_tool/algo/a121/examples/surface_velocity.html. Accessed: 2024-05-07.

[4] Acconeer AB. System overview. https://docs.acconeer.com/en/latest/handbook/a111/system_overview.html. Accessed: 2024-04-09.

[5] William Bugden and Ayman Alahmar. The safety and performance of prominent programming languages. *International Journal of Software Engineering and Knowledge Engineering*, 32(05):713–744, 2022.

[6] FreeRTOS. uxtaskgetstackhighwatermark. https://www.freertos.org/uxTaskGetStackHighWaterMark.html. Accessed: 2024-05-3.

[7] Damien George. Micropython. https://github.com/micropython/micropython, 2024.

[8] GNU. Options that control optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed: 2024-05-21.

[9] Valeriu Manuel Ionescu and Florentina Magda Enescu. Investigating the performance of MicroPython and C on ESP32 and STM32 microcontrollers. In *2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME)*. IEEE, October 2020.

[10] MicroPython. Maximising micropython speed. https://docs.micropython.org/en/latest/reference/speed_python.html. Accessed: 2024-05-3.

[11] MicroPython. Memory managment. https://docs.micropython.org/en/latest/develop/memorymgt.html. Accessed: 2024-05-3.

[12] MicroPython. Micropython project homepage. https://micropython.org/. Accessed: 2024-04-10.

[13] Jim Mussared. The micropython garbage collector. https://www.youtube.com/watch?v=H_xq8IYjh2w. Accessed: 2024-05-3.

[14] Ignas Plauska, Agnius Liutkevičius, and Audronė Janavičiūtė. Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller. *Electronics*, 12(1):143, 2022.

[15] TIOBE. Tiobe index for may 2024. https://www.tiobe.com/tiobe-index/. Accessed: 2024-05-13.

[16] Sebastian Wurl, Markus Faehling, Hanna Vivien Werner, and Martin Langer. Fast micropython controller for flight faults (fmcff). *2023 IEEE Aerospace Conference, Aerospace Conference, 2023 IEEE*, pages 1 – 8, 2023.

[17] Zephyr. Kconfig search. https://docs.zephyrproject.org/latest/kconfig.html#CONFIG_INIT_STACKS. Accessed: 2024-05-3.

Appendices

Appendix A

Pseudocode

In this appendix we will present pseduocode that describes the functionality of the *process* function featured in the surface velocity example.

```
function process(result):
    # Get the data segment using double buffering frame filter
    data_segment = double_buffering_frame_filter(result.frame)

    # Update time series data
    shift time_series by -sweeps_per_frame along axis 0
    set the last sweeps_per_frame entries of time_series to data_segment

    # Compute power spectral density (PSD) using Welch's method
    psds, _ = scipy_welch(time_series, sweep_rate)

    # Update low-pass filtered PSDs
    if update_index * sweeps_per_frame < time_series_length:
        lp_psds = psds
        lp_psds = lp_psds * psd_lp_coeff + psds * (1 - psd_lp_coeff)

    # Get the index corresponding to the distance of interest
    distance_idx = get_distance_idx(lp_psds)
    set distance_idx to the obtained index
    distance = distances[distance_idx]
    psd = lp_psds[:, distance_idx]
    bin_vertical_vs = bin_rad_vs * get_angle_correction(distance)

    # Apply CFAR threshold to PSD and find peaks
    psd_cfar = get_cfar_threshold(psd)
    psd_peak_idxs = cfar_peaks(psd_cfar, psd)

    if length of psd_peak_idxs > 0:
        if maximum absolute value of bin_vertical_vs at psd_peak_idxs > bin_vertical_vs at slow_zone:
            vertical_v, peak_idx, peak_width = get_velocity_estimate(bin_vertical_vs, psd_peak_idxs, psd)
        else:
            vertical_v, peak_idx = get_velocity_estimate_slow_zone(bin_vertical_vs, psd_peak_idxs, psd)
            peak_width = 0

        if absolute value of lp_velocity > 0 and vertical_v / lp_velocity < 0.8:
            if wait_n < max_peak_interval_n:
                vertical_v = lp_velocity
                increment wait_n by 1
            else:
                reset wait_n to 0
        else:
            reset wait_n to 0
    else:
        if wait_n < max_peak_interval_n:
            vertical_v = lp_velocity
            increment wait_n by 1
        else:
            vertical_v = 0
            peak_idx = None
            peak_width = 0

    # Apply dynamic scaling factor to low-pass filtered velocity
    sf = dynamic_sf(velocity_lp_coeff, update_index)
    if update_index * sweeps_per_frame > time_series_length:
        lp_velocity = sf * lp_velocity + (1 - sf) * vertical_v

    increment update_index by 1


    return lp_velocity, distance
```

Figure A.1: Pseudocode for the processing function

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-06-05

EXAMENSARBETE MicroPython Integration for Radar Specific Application: Is it Worth it?
STUDENT Malin Åstrand, Felix Apell Skjutar
HANDLEDARE Jonas Skeppstedt (LTH), Anders Buhl (Acconeer)
EXAMINATOR Sven Robertz (LTH)

Förenklad Utveckling av Radarapplikationer med MicroPython

POPULÄRVETENSKAPLIG SAMMANFATTNING **Malin Åstrand, Felix Apell Skjutar**

I vårt projekt utvärderar vi MicroPythons effektivitet i radarapplikationer jämfört med C. Våra fynd indikerar att MicroPython, trots lägre hastighet, kan vara ett fördelaktigt alternativ för snabb och enkel prototyputveckling.

Föreställ dig att du bygger en cool ny pryl, t.ex. ett radarsystem som mäta ythastigheten på ett vattenflöde. Vanligtvis skriver ingenjörer instruktionerna för prylen med ett programmeringsspråk som heter C. Det är väldigt lätt för den lilla datorn i prylen att förstå C, men det kan ibland vara svårt för människor att göra sig förstådda, speciellt om man vill testa något helt nytt på begränsad tid.

Python är ett annat populärt programmeringsspråk. Det är lätt att lära sig och låter dig skriva program snabbt. Därför föredrar många människor Python när de vill prova nya idéer. Ofta vill man se att idéen fungerar men bryr sig inte lika mycket om att programmet kör lika snabbt som om det skulle vara skrivet i t.ex. C.

MicroPython är en speciell version av Python som är designad för att fungera på små datorer inuti prylar, också kallade inbyggda system, som vårt radarsystem. Det gör programmeringen av dessa prylar lika enkel som att skriva Python på en vanlig dator.

I vårt projekt ville vi se om MicroPython kunde användas för att skapa en radarapplikation på ett effektivt sätt. Tidigare har Acconeer skrivit radarmjukvaran i Python först, och sedan översatt den till C för att köra på prylen. Vi undrade om vi kunde hoppa över det andra steget och använda MicroPython direkt på en radar.

Vi jämförde hur bra vår version av denna radarapplikation skriven i MicroPython fungerade gentemot den skriven i C. Vi testade olika knep för att snabba upp MicroPython genom att använda olika optimeringstekniker och bibliotek. Vi upptäckte att med noggrann planering och optimering kunde vi göra MicroPython endast tre gånger långsammare än C. Detta är inte tillräckligt för att ersätta C, men bra nog för att testa hur väl en ny idé fungerar på inbyggda system.

Till slut skapade vi en guide för andra att använda när de ska besluta om MicroPython är rätt för deras projekt. Om du gör en snabb demo eller prototyp kan MicroPython spara mycket tid och ansträngning, även om det inte är lika snabbt som C. För dem som är nya inom programmering av inbyggda system kan detta göra det mer tillgängligt och roligt att skapa innovativa prylar.