Reducing Energy Consumption for Machine Learning Inference on Edge Devices using C++20 Coroutines

BRUCE BELSON, College of Science & Engineering, James Cook University, Cairns, Australia
JASON HOLDSWORTH, College of Science & Engineering, James Cook University, Cairns, Australia
STEVE KERRISON, School of Science and Technology, James Cook University (Singapore Campus), Singapore, Singapore
BRONSON PHILIPPA, College of Science & Engineering, James Cook University, Townsville City, Australia

Increasingly, machine learning inference is implemented on relatively low-powered edge devices, where battery life is a key performance criterion. In this work, we demonstrate how C++20 coroutines can be used to reorganise the execution order of an iterative inference task on an edge device. A Prognostic and Health Management (PHM) application receives streams of vibration data as envelope spectra from a wireless sensor network and processes them locally through an array of Support Vector Machines. In our experiments on ARM Cortex A72 & A53 64-bit SoCs, this method can reduce energy consumption for the task by up to 18%, reduce overall energy use by up to 20% and cut execution time by up to 20.5%. Furthermore, peak power levels are reduced by up to 4.5%, which can increase battery lifespan by reducing wear. We demonstrate that the necessary changes to the C++ code are simple, repeatable and generally applicable to iterative inference tasks.

CCS Concepts: • **Software and its engineering** → **Coroutines**; • **Hardware** → **Power and energy**; • **Computer systems organization** → *Embedded software*; • **Computing methodologies** → *Concurrent algorithms*; *Support vector machines*.

Additional Key Words and Phrases: C++20 coroutines, C++, embedded systems, machine learning inference, edge AI, energy efficiency

1 Introduction

As the volume of data produced by Internet of Things (IoT) devices grows [13, 45], an ever-increasing amount of data is processed locally, on edge devices [33, 39, 41], rather than being transmitted in full to the cloud for remote processing [2, 33]. Since many of these edge devices are battery-powered, it is important to minimise the energy consumption of machine learning (ML) inference models [38] located at the edge.

The bulk of ML inference is executed by code libraries written in C and C++. Although other languages, particularly Python, are used to control ML processing, the underlying libraries, such as those of NumPy [21], PyTorch [34] and TensorFlow [1] are primarily written in C and C++. Furthermore, inference engines designed for microcontrollers and other edge devices, such as TensorFlow Lite for Microcontrollers [15], and uTensor [40] are implemented purely in C++. Therefore, performance and efficiency improvements of C++ implementations are critical for the practical deployment of ML inference on the edge.

In 2020, the C++20 standard introduced coroutines as a native language feature [7]. Coroutines are subroutines that can be suspended and resumed without loss of local data and state [14, 31]. The C++ implementations of

Authors' Contact Information: Bruce Belson, College of Science & Engineering, James Cook University, Cairns, Queensland, Australia; e-mail: bruce.belson@jcu.edu.au; Jason Holdsworth, College of Science & Engineering, James Cook University, Cairns, Queensland, Australia; e-mail: jason.holdsworth@jcu.edu.au; Steve Kerrison, School of Science and Technology, James Cook University (Singapore Campus), Singapore, Singapore; e-mail: steve.kerrison@jcu.edu.au; Bronson Philippa, College of Science & Engineering, James Cook University, Townsville City, Queensland, Australia; e-mail: bronson.philippa@jcu.edu.au.

ACM 1558-3465/2026/6-ART
https://doi.org/10.1145/3821577

coroutines in LLVM and GCC execute efficiently and can therefore be used as extremely lightweight threads [8, 26, 35]. Coroutines can be used to create a "mini-scheduler" which divides large monolithic iterative tasks into smaller sub-tasks. Each of these sub-tasks can benefit from prefetching [26, 35] and improved memory access patterns [8], resulting in appreciably faster data throughput compared to the standard sequential execution pattern.

Our earlier work [8] used micro-benchmarks to investigate the impact of these techniques on the speed of execution of various algorithms employed in inference engines. However, that work was limited to micro-benchmarks and did not consider the applicability of the technique to real-world use cases, whereas this research studies the effects of a coroutine-based execution pattern on a real application and compares not only data throughput but also the energy consumption characteristics of sequential and coroutine-based execution patterns.

This paper's primary contribution is to be the first work proposing and evaluating the use of C++20 coroutines for energy efficiency in ML inference on edge devices. This is achieved through a mini-scheduler algorithm for the use of coroutines, combined with a measurement framework and test harness to evaluate the energy efficiency of coroutines, using Prognostic and Health Management (PHM) as the motivating application. Our analysis and evaluation of the proposed method provides insights that may contribute to broader use of such an approach.

The rest of the paper is organised as follows: Section 2 focuses on the most recent and relevant work in the field of performance enhancement using coroutines. Section 3 describes the methodology of the techniques used to improve performance and the experimental methods we used to study their impact. Section 4 presents the results of our experiments, Section 5 discusses the results and Section 6 contains our conclusions and suggestions for possible future work. Appendices A and B contain the mini-scheduler algorithm and relevant source code sections, respectively. The code for the experiments can be downloaded from an online repository [1].

2 Related work

This section establishes background work that motivates this research, including the drive to perform ML workloads on edge devices, architectural and programming paradigm shifts that improve performance and efficiency, implementations that have demonstrated benefits from these innovations, and the potential impact this can have on energy consumption. We frame this around the application area of PHM, where ML techniques such as Support Vector Machines (SVMs) have for some time been demonstrated as an effective means of predicting device failures or their remaining useful life [47], and more recently can be used in a wider set of use-cases [24].

2.1 Edge computing

The move towards edge processing of IoT data is driven by several different factors, including on the one hand data-related concerns such as privacy, security and data provenance, and on the other hand network-related issues such as latency, reliability and bandwidth.

Edge computing presents both privacy and security benefits as well as challenges. By computing data closer to the source, less data is transferred to more central locations, but edge devices and their intermediate networks become new elements that must ensure adequate security to protect the data they are handling [49], where end-to-end encryption between source and cloud may previously have sufficed. The provenance of data also becomes an issue, requiring data sources and handlers of data to be properly tracked through logging or immutable methods such as blockchain [23].

Performance benefits arise from this shift, too. AI workloads, which may have massive data requirements, may have impractical latencies for real-time applications such as Augmented Reality (AR), Virtual Reality (VR) and live monitoring of physical processes with sensors. Longer communication chains also impose a more significant

[1] https://github.com/bbelson2/coro_edge_energy.git/

latency penalty in the event of needing to retry. One solution to this challenge is edge processing [11], with the possibility of guaranteed latency and reliability parameters [16]. This is not an "either / or" decision: both edge and cloud can collaboratively process data to improve key metrics within the constraints of each system and the bandwidth required to move data between them [30, 36]. Striking a balance is necessary, because edge computing may also introduce performance penalties arising from processing limitations, rather than network latency [2].

2.2 Architectural and language enhancements

ML workloads often require the processing of large datasets. Thus, efficiency in the transfer of data from memory to processing unit is an important factor. The problems and mitigations of memory access patterns have been examined in the context of ML on edge devices, including efforts to optimize frameworks for edge processors [9] and better understanding of workload memory access patterns to maximise the benefits of data pre-fetching [4].

Hardware acceleration of ML workloads has received significant attention with efforts to optimise at both the hardware [12, 17, 27, 43] and software level [44, 48], and through hybrid [9, 43] solutions. Processing-in-memory (PIM) has been proposed as a means of improving speed and reducing unnecessary data movement during local processing of edge data [19, 20], and supporting architectures are now commercially available [28, 29].

Coroutines [14, 31] can potentially help improve performance of these large ML workloads by dividing them into smaller parts that have simpler memory access patterns and benefit from pre-fetching. However, while lower in overhead than full threads or multi-processing, implementing this design pattern still requires effort. Fortunately, programming languages and toolchains can bridge this gap, for example in the extensively used C++ language. The C++20 standard's introduction of coroutines delegates their implementation to the compiler [7]. Thus, the trade-off between code complexity and performance has changed to the advantage of the developer.

Techniques based on coroutines have been used to improve the performance of database engines on large server platforms [22, 26, 35], with the focus of these works on hiding latency due to cache misses arising from dereferencing of multiple pointers, known as "pointer chasing". Coroutines have also been used to implement software-defined network switches [3], employing a deep-pipelining approach that equals or exceeds the performance a more traditional batching / pre-fetching solution.

2.3 Applications

On a Raspberry Pi 4 B — a small platform widely used for edge processing — C++ coroutines were used to implement a "mini-scheduler" that provided up to 65% speed improvements for a range of ML algorithms and transformations, including 8.3% speed improvement for a SVM [8].

Other languages feature coroutines, for example Python since version 3.5. Encheva et al. [18] demonstrated their usefulness for embedded systems programming with MicroPython on an ESP32 microcontroller, testing simple constructs as well as a webcam server and dynamic scripting over the device's serial port, reducing the threading that would normally be handled within the underlying Real Time Operating System (RTOS), simplifying the code.

2.4 Energy efficiency

Performance and energy efficiency tend to correlate strongly because a task that completes more quickly tends to use less energy. This can be complicated by features such as Dynamic Voltage and Frequency Scaling (DVFS), where racing to idle might usually be most energy-efficient, but if some other activity limits how much power-saving can be obtained outside of the task, then stretching the task out over a longer time, at a lower operating voltage and frequency may be beneficial [6]. This is complicated further by modern multi-core systems, which may have multiple operating states at any one time, across different domains within the chip. Loose timing requirements or lower computational accuracy can also be incorporated into the trade-off [5].

While hiding latency may improve performance, the underlying activity of the memory subsystem, including caches, can still impact energy consumption on a different scale to performance alone. For example, a cache miss determination approach that bypasses cache levels identified as unlikely to contain the desired data [32], was shown to have greater energy savings than time savings in some benchmarks. However, the greatest differences were only present in idealised circumstances where the miss prediction was perfect and consumed no energy itself.

2.5 Motivation

Combining the above areas, there is a clear case for improving the efficiency of ML workloads for edge computing scenarios, with coroutines able to serve this objective. While not a new concept, recent language and compiler enhancements have enabled easier adoption of coroutines, and applications in edge and embedded cases have demonstrated their benefits. Performance and energy are related, but often with many hidden complexities, and so in this paper we seek to analyse prior work on performance-enhancing coroutine-based SVMs, with energy efficiency as the primary focus.

3 Methodology

Our methodology begins by examining the target PHM application, then focusing on the gateway component that is run on a Raspberry Pi. We then describe the coroutine implementation method, followed by how tests are run — including the generated dataset that the SVM will process — along with the performance and energy measurement process. Finally, we describe the statistical processes applied to our collected data.

3.1 Application

The studied architecture is shown in Fig. 1. We consider the case of machine learning inference running on a gateway attached to a wireless sensor network (WSN). Each of the nodes of the WSN is attached to an item of machinery whose health is being monitored. The PHM method is based on comparing the instantaneous frequency analysis of the specific item against a set of expected frequencies recorded for each sensor. The sensor node uses an accelerometer to record the vibrations of the machinery for a sample period, then, in order to isolate frequencies that indicate bearing damage, converts the data to the frequency domain using a locally executed Fast Fourier Transform (FFT).

The gateway application receives data transmitted by the S sensors in the WSN. Each measurement contains a vector of F FFT bins, representing a set of F features: it is packed into a single UDP datagram and transmitted to the gateway. M measurements are collected for each sensor. When all data has been received, the gateway processes all measurements for all sensors. Finally, a single status value for each sensor is transmitted to the cloud.

This architecture — with large amounts of input data and small quantities of output — is typical of many applications: local processing avoids cloud dependency, reduces communications costs and assists privacy. In cases such as this one, where the application is deployed remotely, cloud solutions are not feasible.

Each vector of features within the complete collection of measurements is passed through a SVM using per-sensor weights. This calculation is a multi-level iteration process as described in Algorithm 1.

The SVM calculations for a complete data set provide a test for the coroutine-based execution pattern which this paper examines. This multi-layered set of iterations offers an opportunity for a simple transformation to parallelised, multi-threaded execution, using coroutines as extremely light-weight threads, under the model shown in Fig. 2, as demonstrated in earlier work [8].

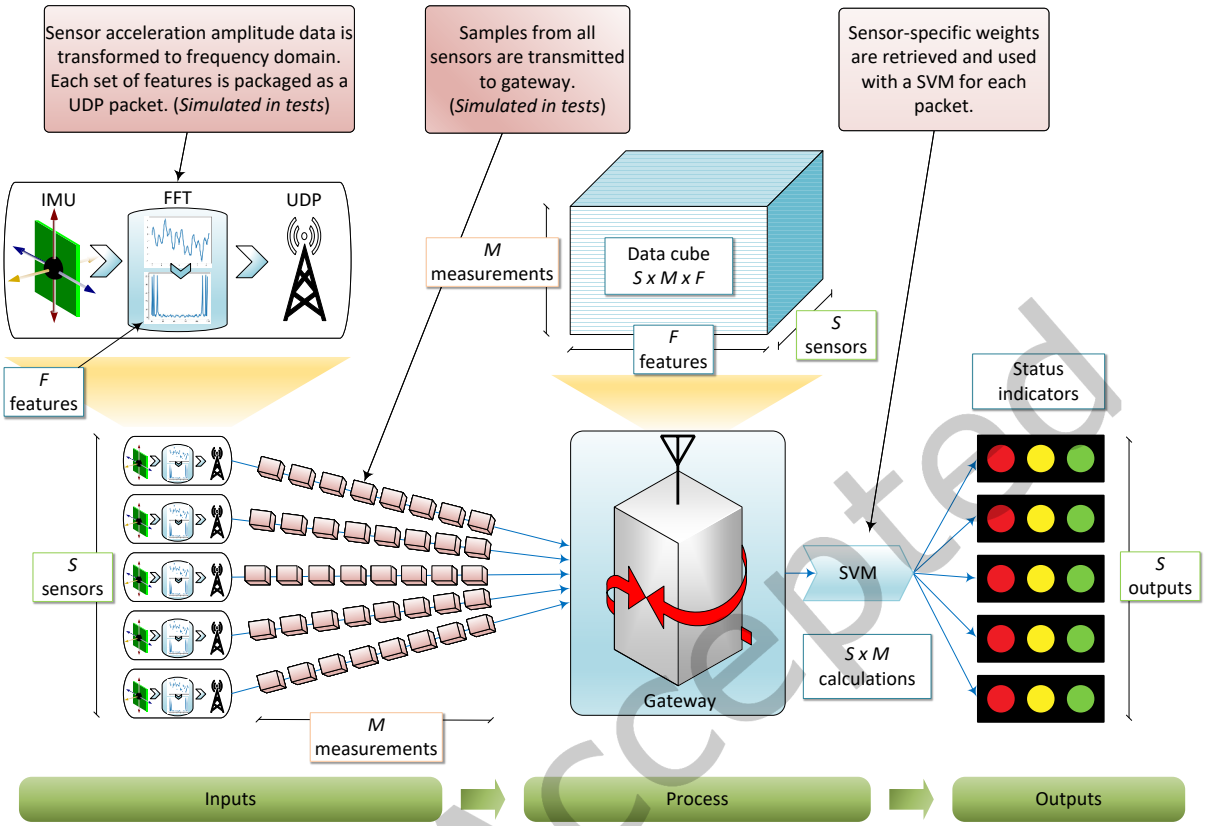


Fig. 1. Networked application. (i) Sensor acceleration data is recorded by inertial measurement unit (IMU) and transformed to frequency domain; (ii) sample vector is sent to gateway as a UDP packet; (iii) gateway retrieves sensor-specific weights and applies SVM to each sample vector.
S sensors each produce M measurements, each containing F features; thus input data is very large ($S \cdot M \cdot F$), but the final output — just one status value for each sensor — is small (S). This architecture is characteristic of edge systems in remote deployments without easy access to the cloud: it avoids cloud dependency, reduces cost and assists privacy. The architecture requires that inference be implemented locally; inference involves many layers of iteration of simple calculations.

3.2 Platform

The gateway computer program was authored in C++20, the first version of C++ to support coroutines [25]. The Clang 12.0 compiler was used, based on its efficient implementation of coroutines [8]. The program was executed on a Raspberry Pi 4 computer [42] with 2GB RAM and a quad core ARM Cortex A72 64-bit SoC with a 2 layer 1 MiB CPU cache. The L1 cache size of this machine is 32 KiB and the L2 cache is 1 MiB. 16-bit fixed point numbers (with 3.13 bits) are used for all real numeric values. A secondary platform - the older Raspberry Pi 3 B+ - was also tested for comparison. (We restricted our focus to the Raspberry Pi ecosystem (i) for simplicity and (ii) as a good fit for available test equipment.) The software platform is summarised in Table 1.

Algorithm 1 Levels of iteration within the application

```
1:  procedure SVMs(sensor_data)                    ▷ Apply SVM to each feature vector in sensor_data
2:      for all s in sensor_data do                ▷ Iterate across sensors - S items
3:          measurements ← measurements[s]         ▷ Retrieve all measurements for this sensor
4:          weights ← weights[s]                   ▷ Look up weights for this sensor
5:          for all m in measurements do           ▷ Iterate across measurements - S x M items
6:              features ← m                       ▷ m contains a vector of features
7:              ▷ Injected code: Initiate a prefetch for [features]
8:              ▷ Injected code: Suspend execution until [features] is loaded into CPU cache
9:              total ← 0
10:             for all f in features do           ▷ Iterate across features - S x M x F items
11:                 total ← total + f.w[i]         ▷ Calculate step of dot product
12:             end for
13:             results[m] ← (total > bias[s])     ▷ Calculate class of this measurement
14:         end for
15:         outcomes[s] ← any(results)             ▷ Calculate outcome for this sensor
16:     end for
17:     return outcomes                            ▷ Return outcomes for all sensors
18: end procedure
```

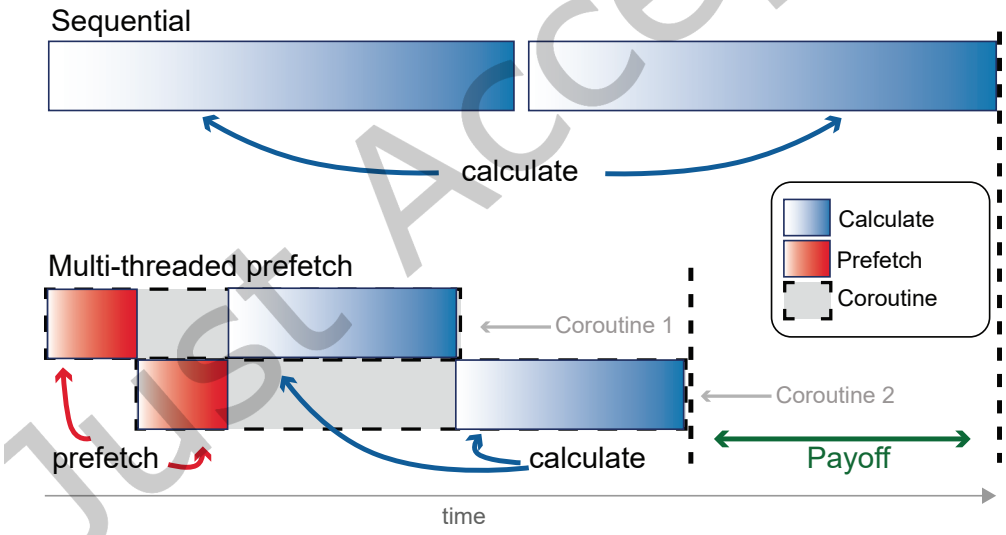


Fig. 2. Coroutine execution model compared with unmodified sequential execution model. (Based on [8] with permission.)

3.3 Coroutine implementation

The coroutine transformation summarised in Fig. 2 and in Algorithm 1.7 improves the speed of iterative tasks that access arbitrarily located blocks of memory: in the case of the SVM operations in the test application, an earlier isolated micro-benchmark [8] demonstrated performance improvements of up to 8.3% on this platform.

Table 1. Software characteristics

Name	Value	Notes
C++ version	C++20 [25]	Earliest C++ version to support native coroutines; only C++ version fully implemented at time of testing
Compiler	Clang 12.0	Most efficient coroutine implementation for this platform [8]
Operating system	Debian Linux 11 (Bullseye)	Minimal implementation (Raspberry Pi OS Lite)
Arithmetic	16-bit fixed-point numbers	3 integer places and 13 fractional places were sufficient for the range and resolution of the data set, which is derived from 12-bit accelerometer readings

Some small modifications to the SVM implementation code are required before it can make use of the coroutine execution model:

(1) The function containing iterations must be transformed to a coroutine. This is done by (i) returning an object that implements the `promise_type` interface and (ii) calling `co_await`, `co_yield` or `co_return` at some point.
(2) Within the iterator two operations must be injected before each new data region is used: (i) initiate a prefetch for the data and (ii) yield to the scheduler by calling `co_await`. Fig. 13a shows the C++ code of the simple modification required for the application code, which is inserted before each SVM operation.

As a result of these changes, the coroutine executing the SVM pauses while the required data is loaded into CPU cache. Loading into cache is executed asynchronously by dedicated machinery independent of the CPU. The mini-scheduler ensures that another task is executed while the cache is loaded. When the scheduler again invokes the waiting task, the data will be in cache and no stall will occur. Furthermore, speed of execution also benefits from an advantageous memory access pattern [8].

3.4 Execution template

The execution context is controlled by a mini-scheduler, implemented as a C++ template. The template is reusable across any multi-level iterative task and is shown in Listing 1 . In order to be called by the scheduler, the coroutine that executes a single step must be reorganised so that it accepts the following parameters: (i) a reference to an application-dependent context class and (ii) the current index into the collection of items to be iterated. This allows the scheduler to manage the list of remaining work efficiently, and without any knowledge of the specific domain or task. An example of this reorganisation is shown in Listing 2.

An instance of the `coroutine_runner` templated class is instantiated and then invoked via its `run()` method (as shown in Listing 3) . The template runs as defined in Algorithm 2, summarised as follows:

(1) The `run()` method creates an collection of coroutines - one for each parallelised lightweight thread.
(2) The method then repeatedly iterates through the collection of coroutines in a round-robin pattern. If a coroutine's current work item is in a wait state it is resumed. If the item is complete then the coroutine is deleted and replaced by a new one, created for the next item in the queue.
(3) The coroutine currently being executed may pause at any time and enter a wait state by calling `co_await`; control then returns to the scheduler's `run()` method, which selects another waiting coroutine.
(4) Once all items have been completed, the `run()` method exits.

Table 2. Execution parameters

Parameter	Symbol	Range	Step	Description
Sensors	S	10 – 50	10	Number of sensors transmitting data packets
Measurements	M	10 – 100	10	Data packets per sensor
Features	F	128 – 2048	64	Number of FFT bins (features) per measurement
Repeats		30		Number of repeats for each test

3.5 Test application

An experimental PHM application already under development was used as the basis for a version created specifically to support the test. The accuracy of inference is not itself on test here, and so the experiments can be de-coupled from real input sensor data, allowing a larger set of tests to be run more readily and repeatably. As such, the sensor nodes were simulated using a pseudo-random series of feature values from a Mersenne twister, MT19937, with a fixed seed. This data is used as UDP traffic representing the sensor network messages. This paper focuses on the gateway part of the WSN and specifically on the energy performance characteristics of the numeric processing required for the local machine learning inference engine.

In addition to the use of simulated data, the code was altered as follows: the two execution models (the standard unmodified sequential model and the modified coroutine-based model) were run alternately for each set of data, in an interleaved pattern; additionally, large regions of memory were modified between each test, in order to flush the CPU cache, and to ensure that each test started without any relevant memory already in cache.

The execution models were run repeatedly as follows:

(1) Receive and collate multiple UDP datagrams from a number of sensors (simulated), then flood cache.
(2) Run test 30 times:
  (a) Apply SVM to each data set using standard sequential execution, then flush cache.
  (b) Apply SVM to each data set using coroutine execution, then flush cache.

The execution parameters were varied as shown in Table 2, for a total of 1,550 tests, each test being run 30 times.

3.6 Performance measurement

The power supplied to the Raspberry Pi was measured using a Joulescope 220 [2], as shown in Fig. 3. The Joulescope sampling rate is 2 MHz (0.5 μs period) and has a sampling precision of 875 μW.

The test application set and cleared general-purpose input/output (GPIO) pins on the Raspberry Pi at the start and end respectively of each processing phase, and this data was passed to the Joulescope to synchronise with the energy measurements, and to be collected in the same result set, as shown in Fig. 4. The GPIO pins were controlled using a library, `wiringPi` [46], which utilises direct hardware access; for this application it displayed latencies of up to 7 μs.

It could be argued that this study might have been more effectively executed by using a simulation of the platform, rather than through the actual time and power measurements used in this paper. However, we decided that building confidence in the actual outcomes — especially with regard to battery life and speed of performance — was important enough to justify the additional time and effort.

[2]https://www.joulescope.com/

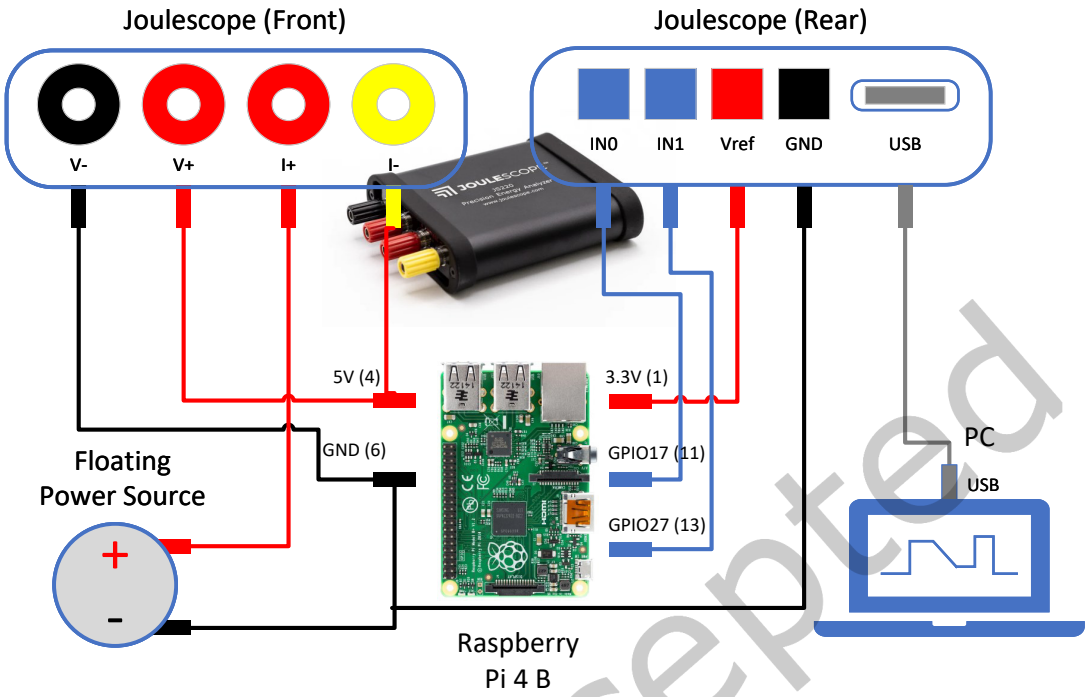


Fig. 3. Wiring layout for experimental procedure. The Joulescope provides power from the floating source to the Raspberry Pi, and measures voltage & current used. The Raspberry Pi provides timing information to the Joulescope through interrupts. All data is merged and passed to a PC via USB.

3.6.1 Time Saving. Time (T) measures the time from the start of the SVM calculations to the end, as shown in Eq. (1). T is the simplest result to measure, since it can be read immediately from the width (along the time-axis) of the rectangular pulse from the appropriate GPIO pin, as shown in the lower two traces in Fig. 4.

$$T = t_1 - t_0 \tag{1}$$

where t_0 and t_1 denote the start and end time respectively.

Tests are examined in pairs: the standard sequential execution and the coroutine execution pattern that follows it. The Time Savings (S_T) for the pair is calculated as shown in Eq. (2). Thus a positive value signifies an improvement in performance (i.e. less time used for the same task).

$$S_T = \frac{T(S) - T(C)}{T(S)} \tag{2}$$

where $T(S)$ and $T(C)$ denote the durations of the sequential pattern and the coroutine pattern respectively.

3.6.2 Overall power and energy savings. Overall Energy (OE) measures the total energy used by the test device from the start of the SVM calculations to the end, as shown in Eq. (3). OE is simple to measure, being the area under the power curve (the upper trace in Fig. 4) between the start and end of the duration's rectangular pulse.

Fig. 4. Example readout from Joulescope, showing the power usage of the device under test (in the upper trace) along with the GPIO pins (in the lower two traces), indicating which execution context is in use.

$$OE = \sum_{t_0}^{t_1} P.\Delta t \tag{3}$$

where P denotes the power measured at Joulescope and Δt denotes the time resolution of the Joulescope.

The energy used by each pair of tests is measured. The Overall Energy Savings (S_{OE}) for the pair is calculated similarly to duration savings, as the reduction in energy divided by the energy of the sequential pattern, as shown in Eq. (4). A positive value signifies an improvement - a reduction in energy used.

$$S_{OE} = \frac{OE(S) - OE(C)}{OE(S)} \tag{4}$$

where $OE(S)$ and $OE(C)$ denote the total energy for the sequential pattern and for the coroutine pattern respectively.

Median Overall Power (OP) for each test is calculated as follows:

$$OP = median([P(t_0)..P(t_1)]) \tag{5}$$

where P denotes power measured at Joulescope and t_0 and t_1 denote the start and end time respectively.

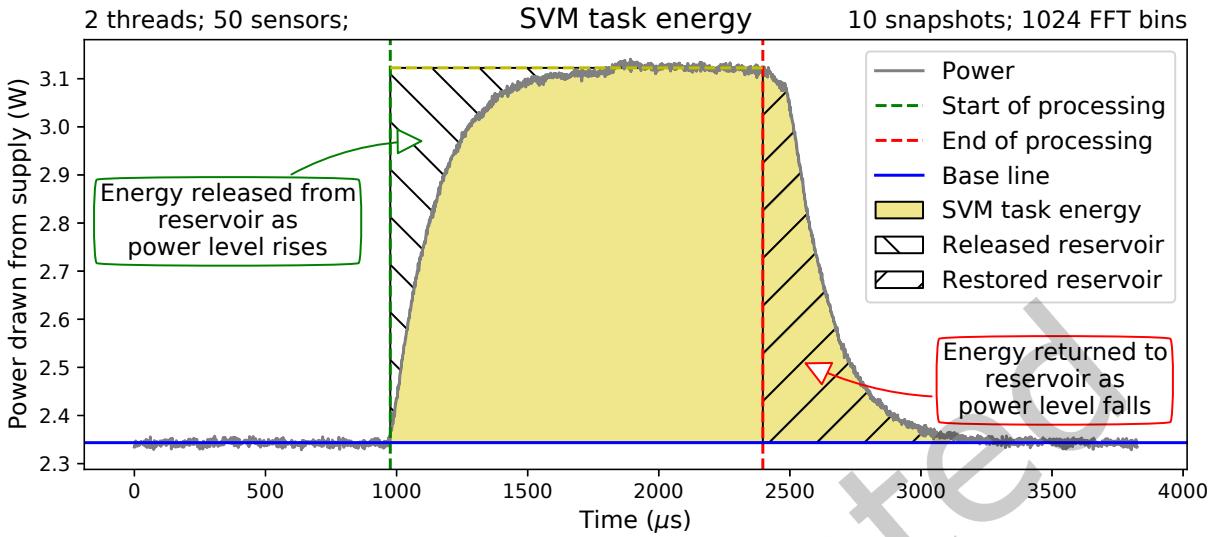


Fig. 5. The pattern of power use for the SVM calculations. Note the delayed rise in power level at the start of processing, due to the release of energy from capacitive reservoirs and the delayed fall after the processing is complete, as the reservoirs are refilled.

The Savings in Median Overall Power (S_{OP}) for the pair is calculated as the negative of the change in power use divided by the power use of the sequential pattern, as shown in Eq. (6). A positive value signifies an improvement - a reduction in median power used.

$$S_{OP} = -\frac{OP(S) - OP(C)}{OP(S)} \tag{6}$$

where $OP(S)$ and $OP(C)$ denote median total power for the sequential pattern and for the coroutine pattern respectively.

3.6.3 SVM task energy. As observed above, Time (T) and Overall Energy (OE) are straightforward to measure; however, the power used *specifically* for the SVM task is more difficult to derive.

Each period of SVM processing caused an immediate rise to a higher power usage level, as shown in Fig. 5. There was a slight delay in reaching the higher level; the length of the delay was consistent, and independent of the duration of the SVM calculations. We attribute this to a discharge of capacitors or other energy storage elements, where the additional power draw was temporarily taken from the capacitors until the voltage regulator was able to respond. There was a similar delay in returning to base power levels at the end of processing, which we attribute to the recharging of the capacitors. The process is described in Fig. 5.

To account for the capacitive-based rise and fall of the measured power, we define the SVM's Task Energy (TE), which is illustrated by the shaded region in Fig. 5. The mathematical definition is as follows.

Starting with a Base Line Power Level (P_{base}) calculated as the median power level in a fixed period before the test begins:

$$P_{base} = median([P(t_0 - t_\delta)..P(t_0)]) \tag{7}$$

where P denotes power measured at Joulescope, t_0 denotes the start time and t_δ denotes the discharge period, we calculate the excess of power use over base line from the start of the calculation and to the point where power

use returned to the base line, as shown in Fig. 5 and in Eq. (8).

$$TE = \sum_{t_0}^{t_1+t_\delta} (P - P_{base}).\Delta t \tag{8}$$

where t_0 and t_1 denote the start and end time respectively, t_δ is the discharge period, P denotes power measured at Joulescope and Δt is the time resolution of the Joulescope.

As with Overall Energy, the SVM's Task Energy Savings (S_{TE}) are calculated as the ratio of the reduction caused by the use of coroutines and the original SVM Task Energy, as shown in Eq. (9).

$$S_{TE} = \frac{TE(S) - TE(C)}{TE(S)} \tag{9}$$

where $TE(S)$ and $TE(C)$ denote the SVM task energy for the sequential pattern and for the coroutine pattern respectively.

3.6.4 Peak power. Peak current — and thus also peak power usage (PP) — is known to have an effect on total battery lifetime [10]. The peak power level during the SVM processing was calculated as follows:

$$PP = \max_{t_0 \to t_1}(P) \tag{10}$$

where P denotes power measured at Joulescope and t_0 and t_1 denote the start and end time respectively.

The savings in peak power level (S_{PP}) were calculated by comparing peak power usage for each test pair as shown in Eq. (11).

$$S_{PP} = \frac{PP(S) - PP(C)}{PP(S)} \tag{11}$$

where $PP(S)$ and $PP(C)$ denote peak power for the sequential pattern and for the coroutine pattern respectively.

3.7 Statistics

Each test was repeated 30 times in order to provide a large enough test set to detect outliers (based on a rule-of-thumb emerging from the Central Limit Theorem). Although the operating system version was "headless" and thus had a minimised number of competing processes and daemons, Raspberry Pi Linux is nevertheless a multi-tasking operating system, and performance of any task will inevitably vary, impacted by the heightened activity of background processes. As well as absorbing CPU cycles for task-switching, these processes can overwrite the cache with their own process-specific data while the task is paused, in which case the cache prefetch will be ineffective, a stall will still occur and an outlier will be recorded.

Median absolute deviation (*MAD*) [37] was used to detect outliers. Within each test, the *MAD* of the duration was calculated for each execution pattern. Results whose deviation from the *MAD* was more than 4 standard deviations (above or below) were excluded. For any pair of results (i.e. consecutive sequential and coroutined executions), the exclusion of either result resulted in the exclusion of both.

Surviving pairs of results were retained, and for each pair the change in performance for time, energy used, median power used, adjusted energy used and peak power level was calculated as described in Eqs. (2), (4), (6), (9) and (11) respectively.

This process resulted in the removal of 8.74% of the test runs for the Raspberry Pi 4 and 15.93% of the Raspberry Pi 3 test runs.

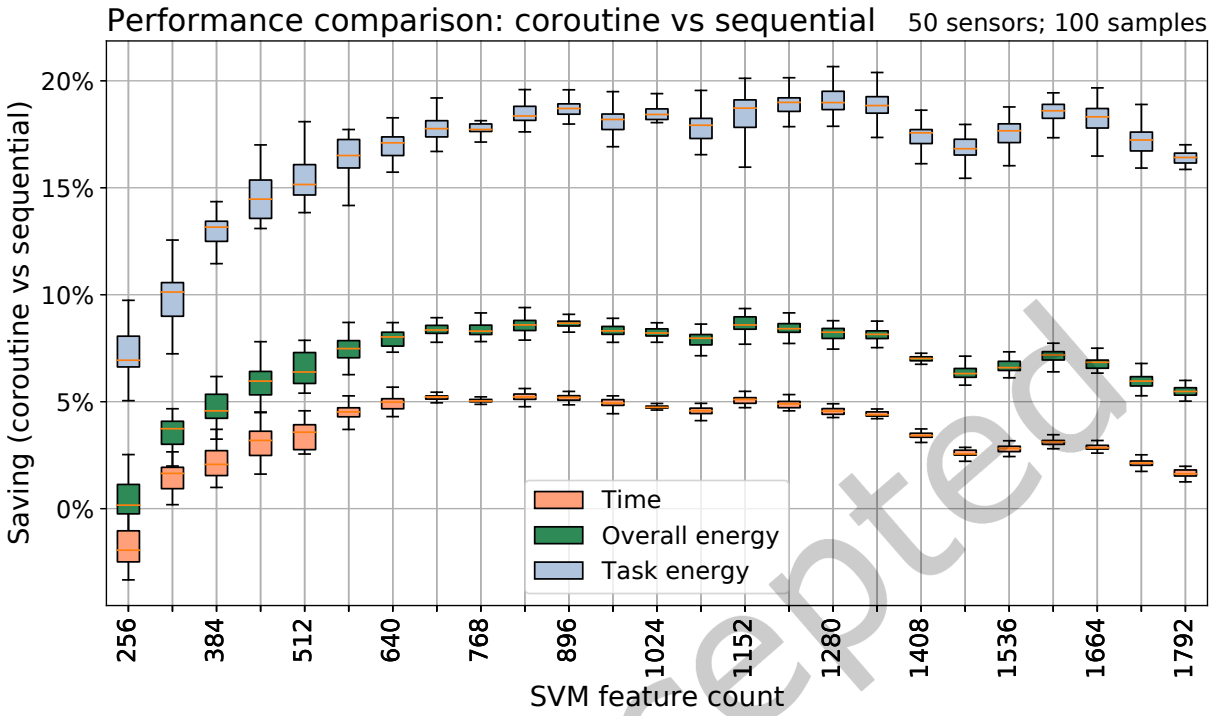


Fig. 6. Summary of performance gains from using coroutines to reorder execution of SVM analyses on Raspberry Pi 4. Statistics displayed represent the reduction in cost divided by the original cost. **Time** is the elapsed time spent on the SVM task; **Overall energy** is the total energy consumed by the edge device during the task; **Task energy** is the marginal energy consumption of the task, after subtracting the base energy use.

4 Results

Our results are presented in terms of time, then power and overall energy, followed by more detailed presentation of the focus area of our efforts — the SVM task — then peak power and platform-specific results, ending with a summary of result consistency.

4.1 Introduction

A typical output from the experiment resembles Fig. 4. The lower two lines on the chart display the rectangular signals supplied directly to the Joulescope by the test application, using the test device's GPIO pins: high signals on pins 0 and 1 signify the normal sequential execution pattern and the coroutined execution pattern respectively. The upper line records the power usage of the test device on the same time axis.

It is immediately apparent — from inspection of test runs such as that in Fig. 4 — that the spike in power use during the coroutined execution is lower than that during sequential execution. Closer examination of the rectangular pulses reveals that the total time for the coroutined operation is also noticeably less than that for the sequential execution.

A typical response to the application of coroutines to the performance in terms of time and energy is summarised in Fig. 6. In this configuration, the use of coroutines resulted in approximately a 5% saving in time and a 19% saving in task energy consumption for feature counts between 1152 and 1280.

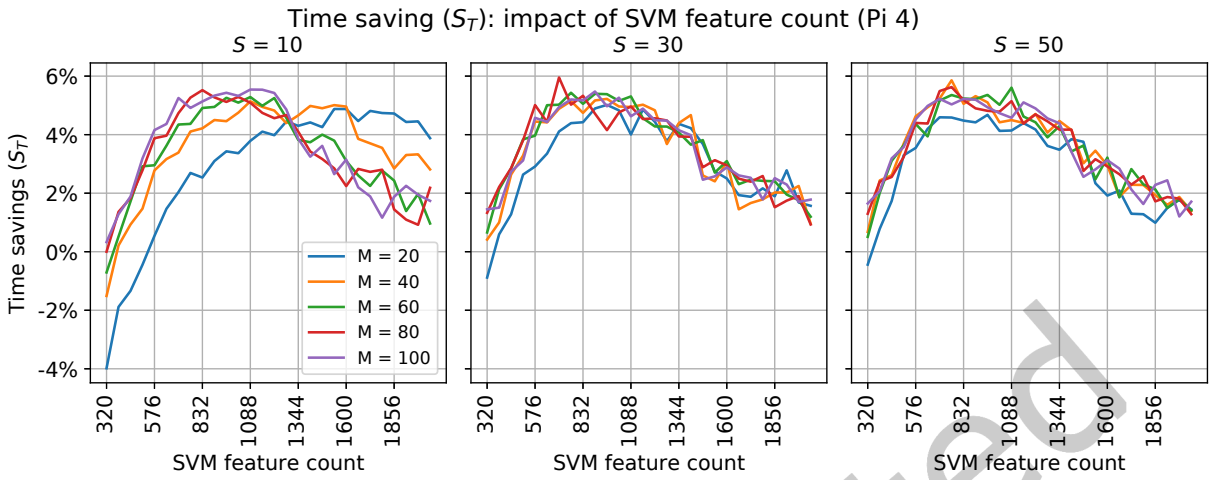


Fig. 7. The effect of SVM feature count on time savings for various sensor counts (S) and measurement counts (M). Each point represents the median value for a single experiment repeated 30 times, with outliers removed as described in Section 3.7. Note the similarity of the response curves across all sample counts. The notable exception is for the smallest sensor and measurement counts (10 x 20 and 10 x 40) which - with a smaller total memory requirement - display less decay for higher feature counts.

Sections 4.2 to 4.7 examine the savings measures individually.

Detailed and summary statistics of the results for each sensor count and measurement count combination can be found in the online repository.

4.1.1 Dimensionality. Many of the figures in this paper, including Figs. 6, 7, 8 and 10, focus on the impact of SVM feature count on performance improvements. SVM feature count in this set of experiments is important in that it represents a class of parameter that cannot easily be manipulated as a configuration setting or a calculated run-time setting, because it is the fixed size of an indivisible unit of work. In this case it represents the length of a feature vector input to the SVM, whose size is a side-effect of the accelerometer used in the sensors.

By contrast, the sensor count (S) and the measurement count (M) represent parameters that *can* be controlled — or tuned — by the application developer; this is similar to a Deep Learning scenario where the developer tunes the batch size in order to optimise the learning task.

4.2 Time

It was expected that SVM feature count (F) would have a major impact on all types of savings, since the cost of pausing and resuming the iterator with `co_await` is incurred exactly once for each feature vector, and this cost is non-trivial.

Fig. 7 confirms this expectation: it shows the impact of SVM feature count on time savings (S_T) for a range of sensor counts (S) and measurement counts (M). It is notable that the pattern of the response with regard to feature count (F) is consistent for almost all sensor counts and measurement counts: a small feature count results in zero time savings; as the size of the vector of features increases the savings increase dramatically, reaching a peak of 5.5% at a feature count of around 960-1088; the savings gradually and steadily fall off thereafter. There is an exception to the pattern for the smallest sensor x measurement count shown (10 x 20), presumably reflecting a total memory size which does not fill the CPU cache.

The poor performance of the coroutine pattern at low feature counts is likely due to two factors. First, there is a fixed cost to using coroutines, which is incurred regardless of the feature count, and which includes splitting tasks, creating coroutine structures, and initializing scheduling state. Second, the cost of pausing and resuming the iterator with `co_await` is incurred exactly once for each feature vector. As the feature count increases, these costs are amortised over a larger amount of work, and thus the savings become more apparent. One possible explanation for the decline in savings at higher feature counts is increased contention for CPU resources and cache capacity as the SVM workload grows.

In summary, there exists - for all except the smallest network sizes - a large region of SVM feature counts where the benefits of the coroutined execution pattern are guaranteed and offer between 4% and 5.5% savings in execution time.

These results are consistent with the time savings found for SVM on a Raspberry Pi 4 B in [8].

4.3 Overall power and median overall energy

A notable outcome of this research is the clear saving in energy usage as a result of replacing the sequential execution pattern with a coroutined execution pattern. Fig. 8a shows the savings (S_{OP}) in overall power usage (OP) between the two execution patterns for a number of different data set sizes and SVM feature counts. Under the modified (coroutined) execution pattern the calculation runs for a shorter time and uses less power while running; outside the smaller feature count zone, the amount of power saved rises steadily, to between 4% and 4.5%.

As shown in Fig. 8b, the effects of time saving and power saving combine to create an overall energy saving (S_{OE}) between 6% and 9%: this saving applies to a usefully large range of feature counts. As would be expected, the impact of feature count on S_{OE} shows similar characteristics to its impact on S_T. The highest energy savings appear for feature counts between 720 and 1024. Once again, there is a consistent shape of response curve for all sensor and measurement counts, and once again there is a notable exception for the smallest sensor and measurement count (10 x 20) which also displays less decay for higher feature counts.

4.4 SVM task energy

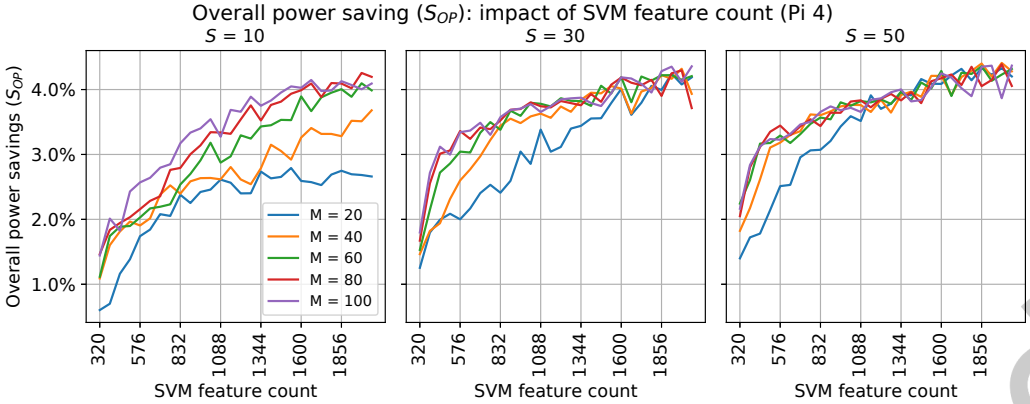
This derived measure represents the energy savings specifically for the SVM task (as described in Section 3.6.3 and in Fig. 5). Fig. 8c shows steady energy savings of over 16% for a wide range of SVM feature counts. The measure follows a similar pattern to the overall energy shown in Fig. 8b. Tests with a feature count between 720 and 1024 show the highest savings, but with a very gradual falling off above 1024.

Fig. 8c shows that the highest range of task energy savings (i.e. >= 16%) coincides with positive time savings (0.5% to 6%) and high total energy savings (5% and above). We observe that there exists a large and reliable range of data sizes where speed, total energy and task energy can all be reliably improved by use of the techniques outlined here.

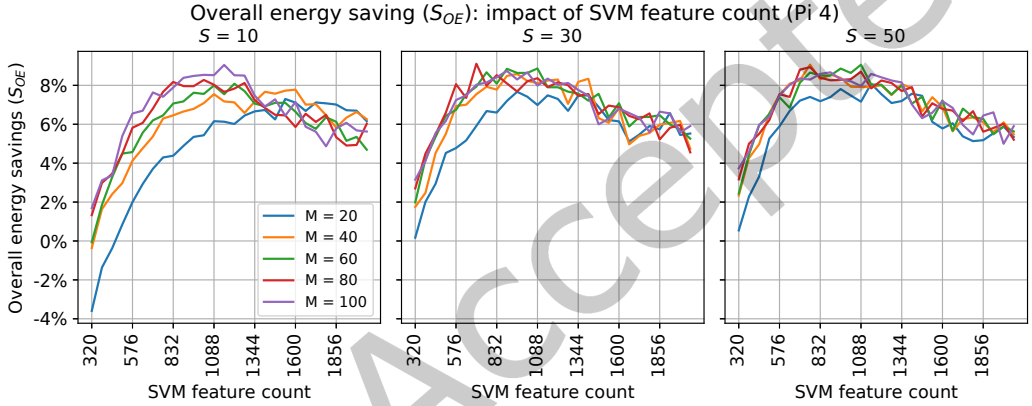
4.5 Peak power

Fig. 9 compares the median peak power savings for each test set with savings in time and average power for the Raspberry Pi 4 platform. The regions of parameter space which display the best average power and time savings —- i.e. the brighter dots (shown in yellow and pale green) on the right-hand-side of the chart —- also exhibit steadily positive peak power savings of between 2% and 4.5%.

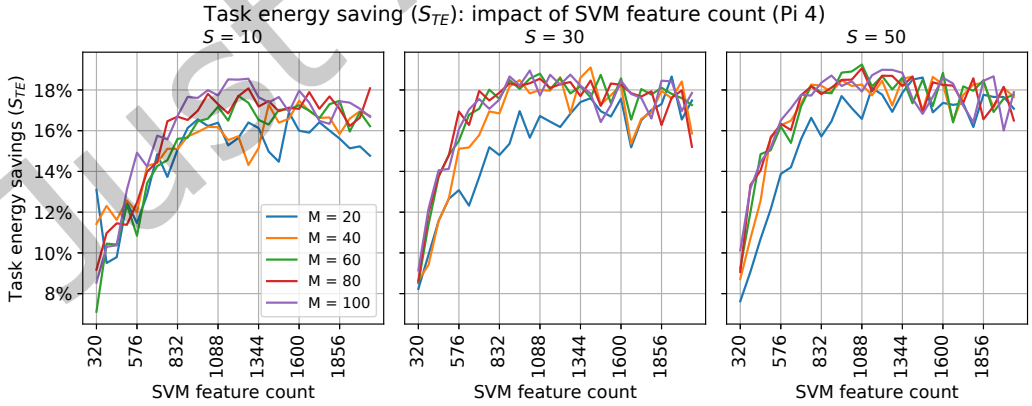
Since peak power levels were between 2.66 W and 3.33 W with a 5V power supply, this saving typically reduced peak current from e.g. 545 mA to between 531 and 520 mA, a saving of between 14 and 25 mA. In percentage terms, this is between 2.6% and 4.6%, with the similarity to the power reduction percentages expected given that current changes will dominate the change in power, while supply voltages should remain fairly stable.



(a) The effect of SVM feature count on overall power savings. The chart shows power savings across all feature counts, rising steadily with feature count, and reaching a plateau at a feature vector size of about 2000.



(b) The effect of SVM feature count on overall energy savings. The effect is similar across all measurement counts except for the data sets with the smallest number of steps in each separate processing task (i.e. 20 measurements).



(c) The effect of SVM feature count on SVM task energy savings for various sensor counts.

Fig. 8. The effect of SVM feature count on energy & power savings measures, for various sensor and sample counts. Each point represents the median value for a single experiment repeated 30 times, with outliers removed as described in Section 3.7.

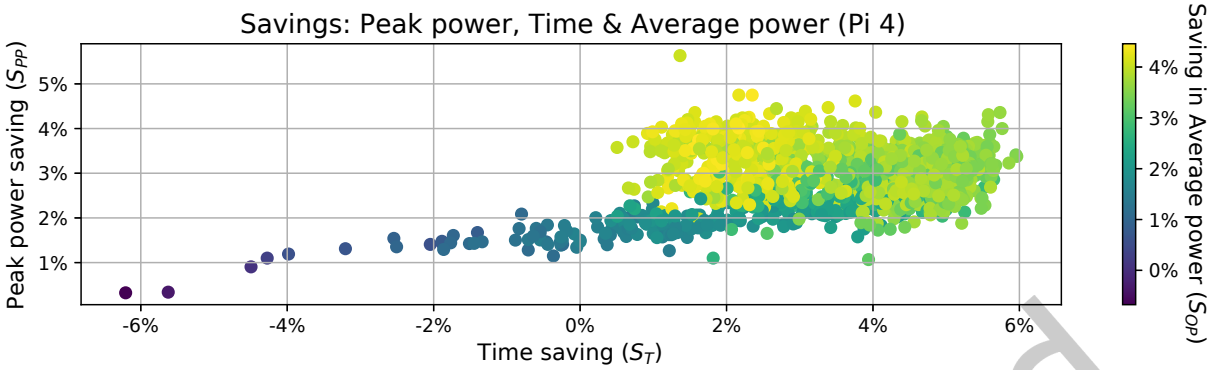


Fig. 9. Comparison of peak power savings with savings in time and average power for the Raspberry Pi 4 B. Each point represents the results for a pair of parameter values: S (sensor count) and M (measurement count). Points to the right of 0% indicate time savings; points that are lighter in colour indicate savings in average power. Notice that the parameter values for S and M which result in useful outcomes for time and power also display useful reductions in peak power — between 2% and 4.5%.

There were no peak power savings for the Raspberry Pi 3 B+ platform: the peak power level was in fact slightly higher on the older platform.

4.6 Comparison with Raspberry Pi 3

To compare base performance across platforms, we used the following calculated statistics:

$$\text{Features per second} = \frac{S \cdot M \cdot F}{T} \tag{12}$$

$$\text{Features per joule (overall energy)} = \frac{S \cdot M \cdot F}{OE} \tag{13}$$

$$\text{Features per joule (task energy)} = \frac{S \cdot M \cdot F}{TE} \tag{14}$$

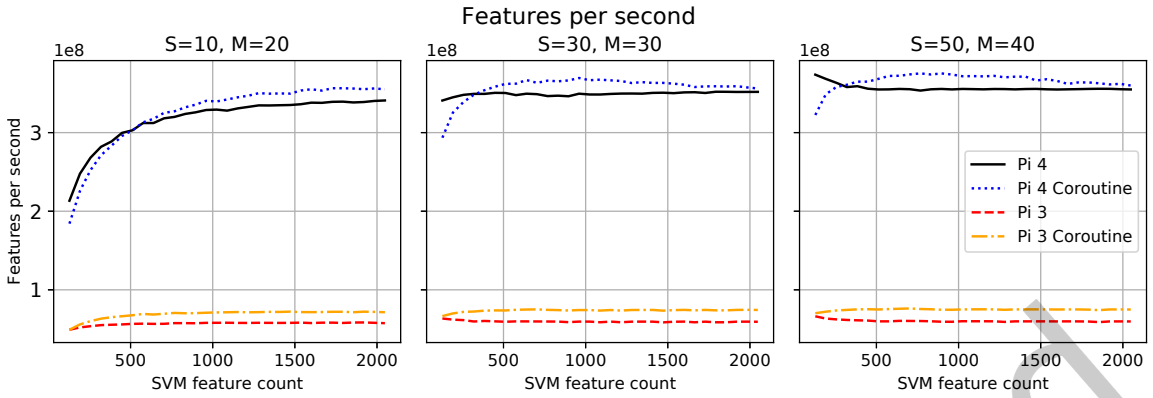
where S is sensor count, M is measurement count, F is SVM feature count and T, OE and TE denote time, overall energy and task energy respectively.

Fig. 10 displays these base performance statistics for both execution patterns on both platforms, across a range of sensor (S) and measurement (M) counts against SVM feature count (F). The consistency of pattern is notable: all performance statistics tend towards a flat response (with regard to feature count) as the count increases beyond 1000.

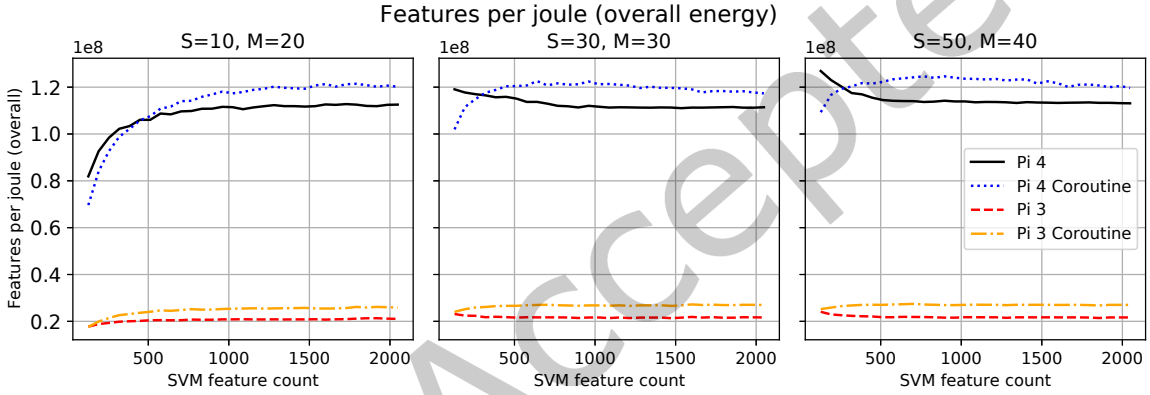
4.6.1 Time. The two platforms have very different performance capabilities. The speed characteristics in Fig. 10a show that the Raspberry Pi 4 B performs the SVM analysis about 6 times as fast as the Raspberry Pi 3 B+.

4.6.2 Overall energy. Fig. 10b shows that the overall energy cost for the newer platform that is around 5.5 times as low as the older platform. This ratio is consistent across sample and measurement counts.

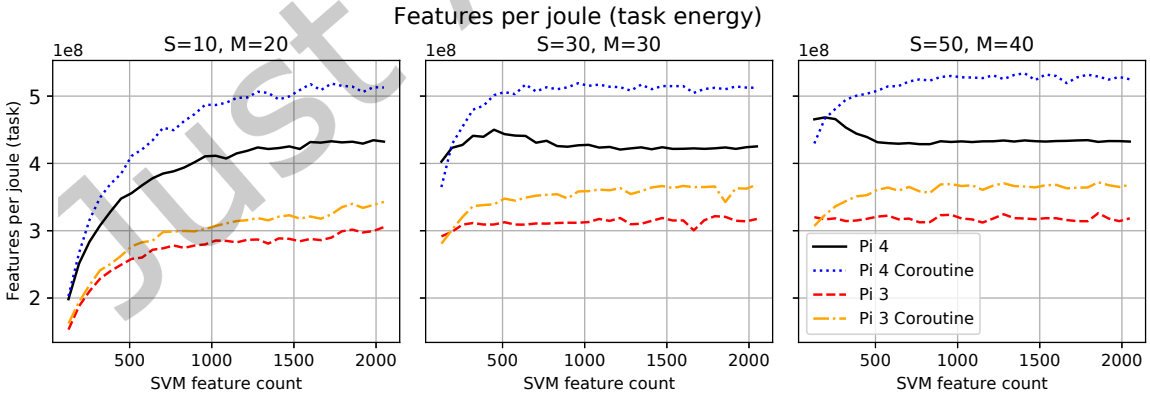
4.6.3 Task energy. We see from Fig. 10c that the difference between the two platforms is much **less** pronounced for the task energy — i.e. after the cost of running the operating systems and other background processes is removed. We might conclude that many of the overall performance improvements gained by enhancements to



(a) The effect of SVM feature count on speed of processing, measured in features per second.



(b) The effect of SVM feature count on overall energy used in processing, measured in features per joule.



(c) The effect of SVM feature count on task energy used in processing, measured in features per joule.

Fig. 10. The effect of SVM feature count on base performance on each platform in terms of speed, overall energy and task energy. Each point represents the median value for a single experiment repeated 30 times, with outliers removed as described in Section 3.7. S: Number of sensors; M: Number of measurements per sensor.

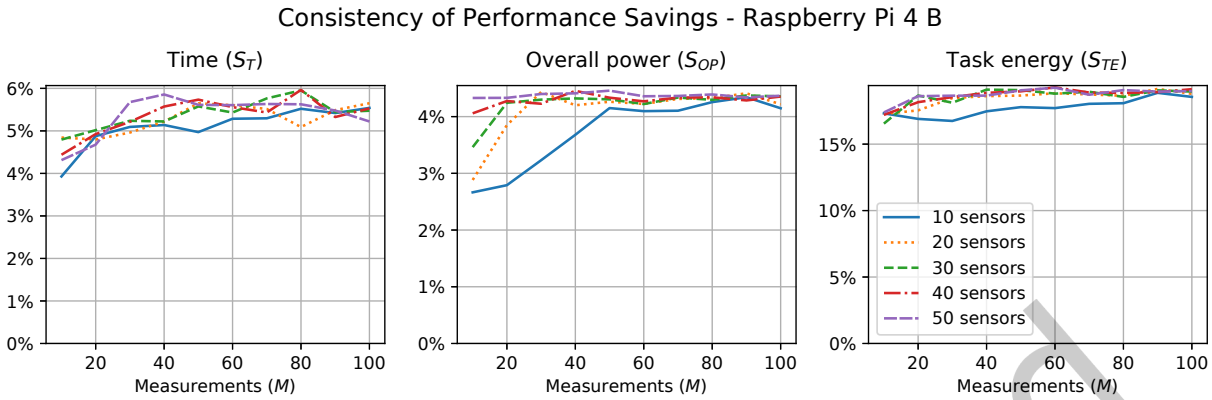


Fig. 11. Summary of best performance savings on Raspberry Pi 4 B. Each chart shows the best performance savings for each metric: S_T, S_{OP} and S_{TE} for the specified sensor and measurement count. These savings are as defined in Eqs. (2), (6) and (9).

the design of the Raspberry Pi 4 do not apply equally strongly to the memory- and CPU-intensive operations that this task requires.

4.6.4 Impact of coroutine execution model. Comparing the impact of the coroutine execution model on the two platforms — as shown in Fig. 10 — we see improvements in all three performance criteria: time, overall energy and task energy. In all three cases — outside of the very low feature counts — there is a clear improvement in performance on both platforms, and the performance enhancement is visible across a wide range of sensor counts, measurement counts and feature counts.

4.7 Consistency of results

Fig. 11 summarises the consistency of the savings achievable in time, overall power and task energy across the various combinations of sensor counts and measurements per sensor, for the Raspberry Pi 4. For each metric the savings were reasonably consistent, with the exception of the cases which had the smallest number of sensors and measurements per sensor. We observe that in general savings for all three metrics rise as the measurement count per sensor is increased.

5 Discussion

Following the presentation of our results, we now examine them and discuss our findings.

5.1 Overall performance savings

Performance gains are summarised in Fig. 12, which compares the range of savings achieved for time (S_T), overall power (S_{OP}), overall energy (S_{OE}) and task energy (S_{TE}) on the two test platforms. Time savings on both platforms are positive and show little variation across different sensor and measurement counts. The time savings on the Pi 3 platform are very large, at around 20.5%, and the time savings for the Pi 4 are lower but still useful at around 3.5%.

The overall power savings on the Pi 4 have a mean of ≈ 3.9% and are asymptotic to 4%; the bulk of results are over 3.2% and even the outliers remain above 2.2%. The overall power usage on the Pi 3 platform is made worse through the application of coroutines: there is a net loss of performance of ≈ 0.5%.

The energy consumed specifically by the SVM calculation task shows important improvements on both platforms: on the Pi 4 the energy use is improved by a mean of 19% and is asymptotic to 18%; there are outliers as

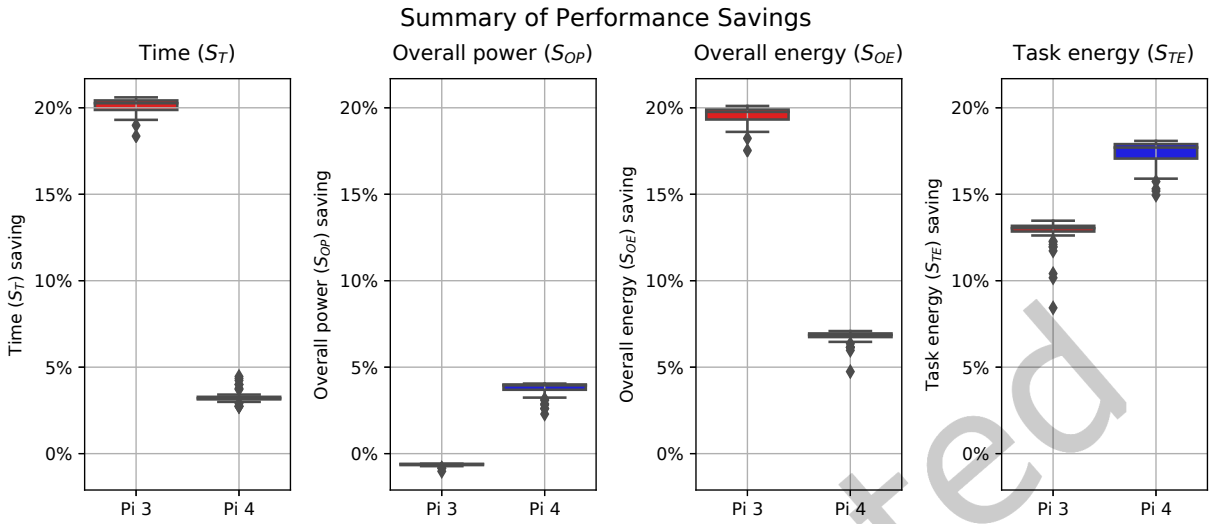


Fig. 12. Summary of best performance savings in time and energy on each platform. The y values represent the mean performance saving for each criterion (S_T, S_{OP}, S_{OE} and S_{TE}) for each specific sensor and measurement count. These savings are as defined in Eqs. (2), (6), (4), and (9). For each combination of sensor count (S) and measurement count (M), the mean performance gain across SVM feature counts > 1024 is collated. This set is shown in the box plot: the box contains the quartiles and the whiskers extend to show the rest of the range, with the exception of outliers, which are shown as dots. (The negative value for S_{OP} on the Pi 3 indicates that the modified algorithm had a net cost - more power was required by the modified coroutine execution pattern than by the unmodified sequential execution pattern.)

low as 15%, which still represents a valuable gain. On the Pi 3 the improvement, with a mean and an asymptote of 13%, is less marked but is still appreciable.

In Fig. 10, we can see a consistent pattern for the impact of SVM feature count on time, overall energy and task energy. In general, savings do not begin until the feature count reaches about 256: this is the point at which the cost — in time and energy — of creating, invoking and managing a coroutine is outweighed by the performance savings attributable to improved memory access patterns. The level of savings develops from 512 to 1024 features and stays fairly static thereafter.

The pattern differs for the smallest data size shown (10 sensors with 20 measurements each): the savings in time and overall energy do not present until the feature count reaches 512, and savings do not stabilise until around 1800. This indicates that — for this smaller number of vectors — the CPU memory cache is not filled until the vectors are proportionately larger.

5.2 Comparison of platforms

While the Raspberry Pi 4 B is capable of performance much superior to its Pi 3 predecessor with regard to both speed of execution of the SVM calculations (Fig. 10a) and overall power used (Fig. 10b), we can observe in Fig. 10c that both platforms use similar amounts of energy specifically for the SVM process. We can also see in Fig. 10c, by comparing the features per joule for the coroutine execution model and the sequential model, that the power savings achieved by applying the coroutine execution model are similar and very clear, tending towards 13% and 18% on the Pi 3 and 4 respectively.

Table 3. Summary of savings in time, power and energy

Platform	Time (S_T)	Overall power (S_{OP})	Overall energy (S_{OE})	Task energy (S_{TE})
Raspberry Pi 3 B+	20.5%	-0.5%	20.0%	13.0%
Raspberry Pi 4 B	3.5%	4.0%	7.5%	18.0%

1. S_T, S_{OP}, S_{OE} and S_{TE} are calculated as defined in Eqs. (2), (6), (4) and (9), respectively.
2. For each sensor-measurements ($S \cdot M$) pair, the SVM feature count with the best median outcome is selected.
3. Values of $S \cdot M < 1024$ are excluded because the curve only becomes asymptotic above this value.
4. For each statistic, the asymptotic value as $S \cdot M$ increases is shown.

```
1 // Apply SVM to each row - suspend each vector
2 for (sample = 0; sample < sample_count; sample++,
      x += row_len, result_ptr++)
3 {
4   x = prefetcher.prefetch(x, data_line_count);
5   co_await std::suspend_always{};
6   *result_ptr = svm_infer(w, x, bias, row_len);
7 }
```

(a) Modification to suspend unconditionally

```
1 // Apply SVM to each row - suspend on demand
2 for (sample = 0; sample < sample_count; sample++,
      x += row_len, result_ptr++)
3 {
4   if (x + row_len > x_pre + data_loaded) {
5     x_pre = prefetcher.prefetch(x, data_line_count);
6     co_await std::suspend_always{};
7   }
8   *result_ptr = svm_infer(w, x, bias, row_len);
9 }
```

(b) Modification to suspend when cache is exhausted

Fig. 13. Modifications to the SVM application code. The added code is shown boxed. In (a) line 4 initiates a memory prefetch for the next input vector; line 5 yields to the scheduler; when other tasks have yielded, control returns to line 6, by which time vector x will have been loaded into CPU cache. In (b) the modification uses a different injection strategy, so that the code only suspends on-demand i.e. when the available prefetched memory is insufficient for the current operation. This strategy was found, experimentally, to be too expensive: the cost of the test outweighed any benefit from memory access pattern improvements.

5.3 Performance trade-offs

The paper compares the performance characteristics of two execution models applied to an iterative calculation task: sequential execution and a switching approach using coroutines to swap cheaply between sub-regions of the task. We explored a substantial test space, varying the sizes of the outer iterations and the amount of memory used, and we observed a variety of performance changes — both positive and negative — across the space explored; we also determined that a large region within the test space reliably offered useful performance improvements.

However, it is important to note that all these tests relied on the use of a simple and unconditional code injection strategy as illustrated in Fig. 13a: the frequency of the task-switching was fixed. However, if a more flexible code injection strategy was used, such as that shown in Fig. 13b — where a test is applied, so that the task switches only when no more precached memory is available — then the cost of executing the injected code becomes unacceptably high. Using this strategy, we were unable to achieve any reliable performance improvements at all, across the same parameter space.

In summary, there is evidence that performance in speed and energy use can be improved by a low-cost task-switching strategy that spreads memory access more evenly across the memory address space of the CPUs studied, but the mechanism that implements the strategy must be carefully managed with regard to its speed and frequency of invocation.

5.4 Dimensionality

As stated in Section 4.1.1 above, many of the results examined here, including those in Figs. 6, 7, 8 and 10, measure the impact of SVM feature count. Because feature count is driven by sensor hardware, it is a parameter that cannot be controlled by the application developer (or at run-time).

The method studied in this paper suits problems where at least some of the dimensionality parameters are controllable; if there is no freedom to tune such parameters, then the technique should not be used.

In other edge computing applications, whether within ML or outside, the efficacy of the execution strategy described in this paper will depend on other parameters — similar to SVM feature count — whose value is fixed by the implementation. It is important to test across the range of likely values for an application instance before investing in the strategy.

5.5 Value of mini-scheduler

We have observed in Section 5.3 that the implementation of the injected code that contains the machinery for task suspension can be the deciding factor in whether the strategy is successful or not; similarly, the implementation of the mini-scheduler will have an important impact on the efficacy of the strategy.

The simple round-robin pattern summarised in Algorithm 2 (and listed in full in Appendix B) has low overheads but does not have enough per-coroutine state information to usefully prioritise between coroutines. It is possible that a more complex prioritisation mechanism would offer further performance benefits; alternatively, a trade-off between complexity and performance might result in the opposite outcome.

6 Conclusions

We have investigated the use of an algorithmic transformation of C++ code to improve runtime performance on an edge device. This transformation uses coroutines and a "mini-scheduler" class to improve the performance of multi-layered highly iterative code — the type of code typically found in machine learning inference applications.

We conducted experiments on two edge gateway devices: a Raspberry Pi 4 B and a Raspberry Pi 3 B+. We measured the impact of the coroutine execution model on the performance characteristics of a support vector machine on a gateway, which locally processed feature vectors passed in by multiple sensors. We varied the number of sensor nodes connected to the gateway device, the number of measurements made by each sensor, and the size of the feature vectors. Table 3 shows a summary of the results.

We observed clear improvements in speed of performance: 3.5% on the Pi 4 and 20.5% on the Pi 3. Notably, we also observed - on the Pi 4 only - a substantive reduction in the overall power used by the system while calculating the SVM: 4.0%.

Combining the impact of time savings and overall power savings, we observed an overall energy saving of 7.5% on the Pi 4 and 20% on the Pi 3. (The energy saving on the Pi 3 is solely a side-effect of the time saving).

Separating out the power used specifically by the SVM calculation, we saw a reduction of 18.0% on the Pi 4 and 13% on the Pi 3.

We observed that the technique offers useful benefits on both of the platforms studied, but in differing ways, as summarised in Table 3: the Pi 4 offers savings in the 3.5% to 4.0% region for both time and overall power, whereas the Pi 3 offered time savings up to 20.5% and a slight power cost.

This paper was restricted to testing on Raspberry Pi devices. Our earlier work [8] established that Intel platforms could also show speed benefits; it would be useful to investigate whether Intel platforms — particularly the small devices used for edge processing — will also display power usage improvements.

The use of this transformation on existing application code offers considerable cost savings. The throughput improvements observed would allow a proportional reduction in the number of gateways in a WSN, with a

positive impact on equipment and deployment costs. The energy improvements — up to around 18% — offer large increases in battery life, with consequent deployment and maintenance savings.

The benefits discussed here are part of a trade-off that places increased code complexity against reduced execution time, energy use and peak current. We assert that - with the use of C++20 - the increase in code complexity is known and manageable: the scheduler code in Listing 1 is easily and immediately applicable to other problem domains; the alterations to the existing iteration code in Listing 2 are small and simple; and the invocation of the iterator/scheduler , as shown in Listing 3, is transparent and short.

6.1 Further work

This initial work has explored the energy benefits of using a specific language's coroutine implementation against a particular ML problem on two Raspberry Pi platforms. As a result, there are several avenues worth exploring:

(1) Testing on further platforms, particularly the now widely available Raspberry Pi 5, as well as competing Single Board Computer (SBC) platforms, including those that use alternative SoCs and architectures such as RISC-V.
(2) Consider other datasets, such as those within PHM that are larger than the scenario used in this work, or data from other application areas, to determine whether similar energy reductions are achievable with coroutine scheduling.
(3) Working with existing C++ ML implementation libraries for edge devices to provide operating system-specific versions of the implementation template and to apply them to the libraries' inference code. We note that in previous work [8], while SVM performance benefitted from prefetch and therefore coroutine scheduling, this was not always the case for other methods.
(4) An investigation into the behaviours underlying the performance improvements, including the use of performance tools to provide analyses of the impact of the technique on (i) cache misses and stalls (affecting instruction cache as well as data cache), (ii) memory access patterns, (iii) CPU utilisation and (iv) dynamic voltage and frequency scaling.
(5) Studying the impact of the technique when used on multiple cores (including platforms with asymmetric cores).
(6) Comparing the benefits of the technique against the use of threads and thread pools.
(7) Identifying other iterative, compute-intensive edge computing tasks that would benefit from the technique, both within and outside ML.
(8) Developing and evaluating the effectiveness of a compiler extension or source-to-source preprocessor that automatically translates sequential code to coroutines when requested. Such approaches would mitigate potential code complexity arising from manual implementation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data can be found, along with source code, at an online repository: https://github.com/bbelson2/coro_edge_energy.git/.

Acknowledgements

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors. However, we would like to acknowledge the James Cook University Singapore 2025 Cross-Campus Research Mobility Grant for supporting the involvement of additional authors.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] Ahmed Ali-Eldin, Bin Wang, and Prashant Shenoy. 2021. The Hidden cost of the Edge: A Performance Comparison of Edge and Cloud Latencies. In *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC '21)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3458817.3476142

[3] Hirochika Asai. 2019. Deep Pipelining: Efficient Pipelining of Network Function Chains with Coroutines. In *2019 IEEE Conf. on Network Softwarization (NetSoft)*. 324–332. doi:10.1109/NETSOFT.2019.8806673

[4] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 513–526. doi:10.1145/3373376.3378498

[5] Muhammad Waqar Azhar, Madhavan Manivannan, and Per Stenström. 2023. Approx-RM: Reducing Energy on Heterogeneous Multicore processors under Accuracy and Timing Constraints. *ACM transactions on architecture and code optimization* 20, 3 (2023), 1–25. doi:10.1145/3605214

[6] M. Waqar Azhar, Miquel Pericàs, and Per Stenström. 2019. SaC: Exploiting Execution-Time Slack to Save Energy in Heterogeneous Multicore Systems. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP '19)*. Association for Computing Machinery, New York, NY, USA, Article 26, 12 pages. doi:10.1145/3337821.3337865

[7] Bruce Belson, Jason Holdsworth, Wei Xiang, and Bronson Philippa. 2019. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 18, 3 (June 2019), 21:1–21:21. doi:10.1145/3319618

[8] Bruce Belson and Bronson Philippa. 2022. Speeding up Machine Learning Inference on Edge Devices by Improving Memory Access Patterns using Coroutines. In *2022 IEEE 25th International Conference on Computational Science and Engineering (CSE)*. IEEE, 9–16. doi:10.1109/CSE57773.2022.00011

[9] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F. Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. 2021. Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks. In *Proc. 30th Int. Conf. Parallel Architectures Compilation Techn. (PACT)*. IEEE, 159–172. arXiv:2109.14320 doi:10.1109/PACT52795.2021.00019

[10] Rebecca Carter, Andrew Cruden, and Peter J. Hall. 2012. Optimizing for efficiency or battery life in a battery/supercapacitor electric vehicle. *IEEE Trans. Veh. Technol.* 61, 4 (2012), 1526–1533. doi:10.1109/TVT.2012.2188551

[11] Zhuoqing Chang, Shubo Liu, Xingxing Xiong, Zhaohui Cai, and Guoqing Tu. 2021. A Survey of Recent Advances in Edge-Computing-Powered Artificial Intelligence of Things. *IEEE Internet Things J.* 8, 18 (2021), 13849–13875. doi:10.1109/JIOT.2021.3088875

[12] Yu Hsin Chen, Tien Ju Yang, Joel S. Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE J. Emerg. Sel. Top. Circuits Syst.* 9, 2 (2019), 292–308. arXiv:1807.07928 doi:10.1109/JETCAS.2019.2910232

[13] Cisco. 2016. Cisco Global Cloud Index: Forecast and Methodology, 2015–2020. https://www.cisco.com/c/dam/m/en_us/service-provider/ciscoknowledgenetwork/files/622_11_15-16-Cisco_GCI_CKN_2015-2020_AMER_EMEAR_NOV2016.pdf

[14] Melvin E Conway. 1963. Design of a separable transition-diagram compiler. *Commun. ACM* 6, 7 (July 1963), 396–408. doi:10.1145/366663.366704

[15] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, and Others. 2021. Tensorflow lite micro: Embedded machine learning for TinyML systems. *Proc. Mach. Learn. Syst.* 3 (2021), 800–811.

[16] Mohammed S. Elbamby, Cristina Perfecto, Chen Feng Liu, Jihong Park, Sumudu Samarakoon, Xianfu Chen, and Mehdi Bennis. 2019. Wireless Edge Computing With Latency and Reliability Guarantees. *Proc. IEEE* 107, 8 (2019), 1717–1737. arXiv:1905.05316 doi:10.1109/JPROC.2019.2917084

[17] Murali Emani, Zhen Xie, Siddhisanket Raskar, Varuni Sastry, William Arnold, Bruce Wilson, Rajeev Thakur, Venkatram Vishwanath, Zhengchun Liu, Michael E. Papka, Cindy Orozco Bohorquez, Rick Weisner, Karen Li, Yongning Sheng, Yun Du, Jian Zhang, Alexander Tsyplikhin, Gurdaman Khaira, Jeremy Fowers, Ramakrishnan Sivakumar, Victoria Godsoe, Adrian Macias, Chetan Tekur, and Matthew Boyd. 2022. A Comprehensive Evaluation of Novel AI Accelerators for Deep Learning Workloads. In *2022 IEEE/ACM International*

Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). IEEE, 13–25. doi:10.1109/PMBS56514.2022.00007

[18] Sylvia Encheva, Sharil Tumin, and Yuhua Luo. 2021. Cooperative Dynamic Programmable Devices Using Actor Model for Embedded Systems of Microcontrollers. In *Cooperative Design, Visualization, and Engineering*. Lecture Notes in Computer Science, Vol. 12983. Springer International Publishing AG, Switzerland, 126–137. doi:10.1007/978-3-030-88207-5_13

[19] Gabriel Falcao and João Dinis Ferreira. 2023. To PiM or Not to PiM. *Commun. ACM* 66, 6 (2023), 48–55. doi:10.1145/3589995

[20] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. *SIGMETRICS Perform. Eval. Rev.* 50, 1 (2022), 33–34. doi:10.1145/3547353.3522661

[21] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. doi:10.1038/s41586-020-2649-2

[22] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. Corobase: Coroutine-oriented main-memory database engine. *Proc. VLDB Endow.* 14, 3 (2020), 431–444. arXiv:arXiv:2010.15981v1 doi:10.14778/3430915.3430932

[23] Rui Hu, Zheng Yan, Wenxiu Ding, and Laurence T. Yang. 2020. A survey on data provenance in IoT. *World Wide Web* 23, 2 (2020), 1441–1463. doi:10.1007/s11280-019-00746-1

[24] Hong Zhong Huang, Hai Kun Wang, Yan Feng Li, Longlong Zhang, and Zhiliang Liu. 2015. Support vector machine based estimation of remaining useful life: current research status and future trends. *Journal of Mechanical Science and Technology* 29, 1 (2015), 151–163. doi:10.1007/s12206-014-1222-z

[25] ISO/IEC. 2020. C++ Draft International Standard - N4860. https://isocpp.org/files/papers/N4860.pdf

[26] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the "killer nanoseconds". *Proc. VLDB Endow.* 11, 11 (2018), 1702–1714. doi:10.14778/3236187.3236216

[27] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA) (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3079856.3080246

[28] Young Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang Hyuk Kwon, Je Min Ryu, Jong Pil Son, O. Seongil, Hak Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun Sung Shin, Jin Kim, Beng Seng Phuah, Hyoung Min Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, Soo Young Kim, Eun Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, Joon Ho Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *Proc. IEEE Int. Solid-State Circuits Conf.*, Vol. 64. 350–352. doi:10.1109/ISSCC42613.2021.9365862

[29] Daewoong Lee, Jaehyeok Baek, Hye Jung Kwon, Dae Hyun Kwon, Chulhee Cho, Sang Hoon Kim, Donggun An, Chulsoon Chang, Unhak Lim, Jiyeon Im, Wonju Sung, Hye Ran Kim, Sun Young Park, Hyoung Joo Kim, Hoseok Seol, Juhwan Kim, Jungbum Shin, Gil Young Kang, Yong Hun Kim, Sooyoung Kim, Wansoo Park, Seok Jung Kim, Chan Yong Lee, Seungseob Lee, Tae Hoon Park, Chi Sung Oh, Hyodong Ban, Hyungjong Ko, Hoyoung Song, Tae Young Oh, Sang Joon Hwang, Kyung Suk Oh, Jung Hwan Choi, and Jooyoung Lee. 2023. A 16-Gb T-Coil-Based GDDR6 DRAM with Merged-MUX TX, Optimized WCK Operation, and Alternative-Data-Bus Achieving 27-Gb/s/Pin in NRZ. *IEEE J. Solid-State Circuits* 58, 1 (2023), 279–290. doi:10.1109/JSSC.2022.3222203

[30] Jialei Liu, Ao Zhou, Chunhong Liu, Tongguang Zhang, Lianyong Qi, Shangguang Wang, and Rajkumar Buyya. 2022. Reliability-Enhanced Task Offloading in Mobile Edge Computing Environments. *IEEE Internet Things J.* 9, 13 (2022), 10382–10396. doi:10.1109/JIOT.2021.3115807

[31] Christopher D Marlin. 1979. *Coroutines: A Programming Methodology, a Language Design and an Implementation*. Number 95 in Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

[32] G. Memik, G. Reinman, and W.H. Mangione-Smith. 2003. Just say no: benefits of early cache miss determination. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* 307–316. doi:10.1109/HPCA.2003.1183548

[33] Jianli Pan and James McElhannon. 2018. Future Edge Cloud and Edge Computing for Internet of Things Applications. *IEEE Internet Things J.* 5, 1 (2018), 439–449. doi:10.1109/JIOT.2017.2767608

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, Soumith Chintala, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H Wallach, H Larochelle, A Beygelzimer, F d'Alché-Buc, E Fox, and R Garnett (Eds.), Vol. 32. Curran Associates, Inc. arXiv:1912.01703

[35] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *VLDB J.* 28, 4 (2019), 451–471. doi:10.1007/s00778-018-0533-6

[36] Jinke Ren, Guanding Yu, Yinghui He, and Geoffrey Ye Li. 2019. Collaborative Cloud and Edge Computing for Latency Minimization. *IEEE Trans. Veh. Technol.* 68, 5 (2019), 5031–5044. doi:10.1109/TVT.2019.2904244

[37] Peter J. Rousseeuw and Christophe Croux. 1993. Alternatives to the median absolute deviation. *J. Am. Stat. Assoc.* 88, 424 (1993), 1273–1283. doi:10.1080/01621459.1993.10476408

[38] Farzad Samie, Lars Bauer, and Jörg Henkel. 2019. From Cloud Down to Things: An Overview of Machine Learning in Internet of Things. *IEEE Internet Things J.* 6, 3 (June 2019), 4921–4934. doi:10.1109/JIOT.2019.2893866

[39] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Ramón Caceres, and Nigel Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Comput.* 8, 4 (2009), 14–23. doi:10.1109/MPRV.2009.82

[40] Nikolaos Schizas, Aristeidis Karras, Christos Karras, and Spyros Sioutas. 2022. TinyML for Ultra-Low Power AI and Large Scale IoT Deployments: A Systematic Review. *Future Internet* 14, 12 (2022). doi:10.3390/fi14120363

[41] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* 3, 5 (2016), 637–646. doi:10.1109/JIOT.2016.2579198

[42] Ahmet Ali Süzen, Burhan Duman, and Betul Şen. 2020. Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn. In *Proc. Int. Congr. Hum.- Comput. Interact., Optim. Robot. Appl.* IEEE, 1–5. doi:10.1109/HORA49412.2020.9152915

[43] Vivienne Sze, Yu Hsin Chen, Tien Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017), 2295–2329. arXiv:1703.09039 doi:10.1109/JPROC.2017.2761740

[44] Gaurav Verma, Yashi Gupta, Abid M. Malik, and Barbara Chapman. 2021. Performance Evaluation of Deep Learning Compilers for Edge Inference. In *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*. IEEE, 858–865. doi:10.1109/IPDPSW52791.2021.00128

[45] Melvin M Vopson. 2020. The information catastrophe. *AIP Advances* 10, 8 (2020), 085014. doi:10.1063/5.0019941

[46] WiringPi. 2026. WiringPi. https://github.com/WiringPi/WiringPi

[47] Bo-Suk Yang and Achmad Widodo. 2008. Support Vector Machine for Machine Fault Diagnosis and Prognosis. *Journal of System Design and Dynamics* 2, 1 (2008), 12–23. doi:10.1299/jsdd.2.12

[48] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A runtime-based computational performance predictor for deep neural network training. In *2021 USENIX Annual Technical Conference*. 503–521. arXiv:2102.00527

[49] Jiale Zhang, Bing Chen, Yanchao Zhao, Xiang Cheng, and Feng Hu. 2018. Data Security and Privacy-Preserving in Edge Computing Paradigm: Survey and Open Issues. *IEEE Access* 6 (2018), 18209–18237. doi:10.1109/ACCESS.2018.2820162

A Algorithms

Algorithm 2 Operation of mini-scheduler for coroutines

```
1:  procedure RUN(cc, n, coro)          ▷ Run coroutine 'coro' for each of 'n' items, using 'cc' coroutines.
2:      coros ← [1, ..., cc]                                  ▷ Create an array of coroutines
3:      done ← [1, ..., cc]                                   ▷ Array of completion flags for each coroutine
4:      for i = 1, ..., cc do
5:          coros[i] ← new coroutine(coro, i)                 ▷ Create new coroutine to process item i
6:          done[i] ← false                                   ▷ Coroutine's initial state is 'incomplete'
7:      end for
8:      next ← cc + 1                                         ▷ Index for next item to be processed
9:      completed ← 0
10:     while completed < n do                                ▷ Continue until all work items are completed
11:         for i = 1, ..., cc do                             ▷ Inspect each coroutine in turn – round-robin
12:             if not done[i] then                           ▷ If any work remains
13:                 if not coros[i].is_complete() then
14:                     coros[i].resume()                     ▷ If not complete – continue
15:                 else
16:                     if next == n then                     ▷ Test whether any work items remain
17:                         done[i] ← true                    ▷ No work remains – mark this slot as complete
18:                     else
19:                         coros[i] ← new coroutine(coro, next)   ▷ Replace with new instance
20:                         next ← next + 1
21:                     end if
22:                     completed ← completed + 1
23:                 end if
24:             end if
25:         end for
26:     end while
27: end procedure
```

B Source code

Listing 1 contains the mini-scheduler used in these experiments, a generalised C++ template class. Listing 2 shows the SVM iteration algorithm to which the scheduler was applied, both before and after the changes that were required by the scheduler. Listing 3 demonstrates the invocation of the scheduler for the batch of SVM tasks. The full source code is managed at https://github.com/bbelson2/coro_edge_energy.git/.

```
1 // Generalised runner for parallelising iterative
2 // tasks across multiple coroutines.
3 template <typename REFDATA_T>
4 class coroutine_runner {
5   public:
6   typedef resumable_t (*coro_fn_t)(
7   const prefetcher_t &prefetcher,
8   REFDATA_T &refdata, size_t coroutine_index);
```

```
9
10  coroutine_runner(const prefetcher_t &prefetcher,
11  REFDATA_T &refdata)
12  : prefetcher_(prefetcher), refdata_(refdata) {}
13
14  void run(size_t coroutine_count, size_t item_count,
15  coro_fn_t coro_fn) {
16    // A collection of parallelised coroutines
17    std::vector<resumable_t> tasks;
18    // Status of all items
19    std::vector<bool> done(coroutine_count, false);
20    size_t incomplete = item_count;
21
22    // Create coroutines, ready to run. There will always be at most
23    // coroutine_count of them, and each will be deleted and replaced
24    // by a new one when its item is complete.
25    for (size_t b = 0; b < coroutine_count; b++) {
26      tasks.push_back(coro_fn(prefetcher_, refdata_, b));
27    }
28
29    // Work through all tasks until all are done
30    size_t next_item = coroutine_count;
31    while (incomplete > 0) {
32      for (size_t c = 0; c < tasks.size(); c++) {
33        if (done[c]) {
34          continue;
35        }
36        resumable_t &t = tasks[c];
37        if (t.is_complete()) {
38          if (next_item < item_count) {
39            tasks[c] = coro_fn(prefetcher_, refdata_,
40            next_item);
41            next_item++;
42          } else {
43            done[c] = true;
44          }
45          incomplete--;
46        } else {
47          t.resume();
48        }
49      }
50    }
51  }
52
53  protected:
54  const prefetcher_t &prefetcher_;
55  REFDATA_T &refdata_;
```

```
56 };
```

Listing 1. Generalised C++ template to apply the coroutined execution context to any function after it has been converted to a coroutine with three parameters: (i) prefetcher class, (ii) an application-dependent context class and (iii) an index into the collection of sub-tasks (as defined by `typedef coro_fn_t` in lines 5-7).

```
// SVM iteration algorithm, before being refactored as a coroutine
void infer_sensor_sequential(runtime_data &rt_data, uint32_t sensor_index)
{
  const data_item_t *x, *w;

  // Resolve weights & bias for this sensor
  w = rt_data.resolve_w(sensor_index);

  // Inspect w data
  data_item_t bias = w[0];
  w++;

  // Get sensor data base
  const data_vector_t& x_vec = rt_data.resolve_x_vec(sensor_index);
  x = x_vec.data();
  auto row_len = rt_data.rt.sv_len;
  auto sample_count = x_vec.size() / row_len;

  // Get result base
  std::vector<result_t>& results = rt_data.resolve_results_vec(sensor_index);
  result_t* result_ptr = results.data();

  for (uint32_t sample = 0; sample < sample_count; sample++, x += row_len, result_ptr++)
  {
    *result_ptr = svm_infer(w, x, bias, row_len) ? 1 : 0;
  }
}

// Controller to run task sequentially
void run_infer_sequential(runtime_data &rt_data)
{
  for (uint32_t i = 0; i < rt_data.rt.sensor_count; i++)
  {
    infer_sensor_sequential(rt_data, i);
  }
}

// SVM iteration algorithm, after refactoring as a coroutine
static resumable infer_sensor_coro(const prefetcher_t &prefetcher, runtime_data &rt_data,
size_t coroutine_index)
{
  uint32_t sensor_index = (uint32_t)coroutine_index;
  co_await CORO_STD::suspend_always{};

  const data_item_t *x, *w;

  // Calculate prefetch sizes
  size_t weights_size = rt_data.rt.w_len * sizeof(data_item_t);
  size_t weights_line_count = to_pf_line_count(weights_size);

```

```
  // Resolve weights & bias for this sensor
  w = rt_data.resolve_w(sensor_index);
  w_next = prefetcher.prefetch(reinterpret_cast<const char*>(w), weights_line_count);
  co_await CORO_STD::suspend_always{};

  // Inspect w data
  data_item_t bias = w[0];
  w++;

  // Get sensor data base
  const data_vector_t& x_vec = rt_data.resolve_x_vec(sensor_index);
  x = x_vec.data();
  auto row_len = rt_data.rt.sv_len;
  auto sample_count = x_vec.size() / row_len;

  // Calculate prefetch sizes
  size_t data_size = row_len * sizeof(data_item_t);
  size_t data_line_count = to_pf_line_count(data_size);
  size_t results_size = sample_count * sizeof(result_t);
  size_t results_line_count = to_pf_line_count(results_size);

  // Get result base
  std::vector<result_t>& results = rt_data.resolve_results_vec(sensor_index);
  result_t* result_ptr = results.data();
  result_next = prefetcher.prefetchw(reinterpret_cast<char*>(result_ptr), results_line_count);

  for (uint32_t sample = 0; sample < sample_count; sample++, x += row_len, result_ptr++)
  {
    x_next = prefetcher.prefetch(reinterpret_cast<const char*>(x), data_line_count);
    co_await CORO_STD::suspend_always{};
    *result_ptr = svm_infer(w, x, bias, row_len) ? 1 : 0;
  }
}
```

Listing 2. Sample task - `infer_sensor_sequential()` - to calculate SVM for each of a set of $n_s \cdot n_p$ vectors. The task is reorganised as a coroutine — `infer_sensor_coro()` — ready to be run by the mini-scheduler `coroutine_runner::run()`.

```
// Declare instance of platform-dependent memory prefetcher
prefetcher_t prefetcher;

// Declare an instance of the execution context template,
// which references the application-dependent runtime data
// rt_data.
coroutine_runner<runtime_data> runner_with_prefetch(
  prefetcher, rt_data);

// Invoke the execution context instance to run the
// coroutined function across all instances of the sensor
// data sets.
runner_with_prefetch.run(rt_data.task_count,
  rt_data.sensor_count, infer_sensor_coro);
```

Listing 3. Typical usage of C++ template to run a coroutine across iterative tasks.

Received 19 December 2025; revised 19 May 2026; accepted 28 May 2026