Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Eirik Vale Aase

Minimizing the Energy Consumption of Soft Real-Time Applications on a Multi-Core Ultra-Low-Power Device

Master's thesis in Computer Science
Supervisor: Magnus Jahre
June 2020

NTNU
Norwegian University of
Science and Technology

Eirik Vale Aase

Minimizing the Energy Consumption of Soft Real-Time Applications on a Multi-Core Ultra-Low-Power Device

Master's thesis in Computer Science
Supervisor: Magnus Jahre
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Assignment Description

Internet of Things (IoT) represents a technology paradigm shift in which computers are embedded inside everyday machines, often times smaller in size and consuming less energy than traditional computers, but usually unable to perform resource-demanding tasks. Supporting this paradigm shift are Ultra-Low-Power (ULP) devices. This thesis focuses on the subset of ULP devices that are characterized as soft real-time systems, i.e., they must complete tasks before a given deadline. Recently, IoT devices have started focusing on applications in the machine learning (ML) domain which tend to push ULP devices to their performance limits. Thus, multi-core architectures become attractive since they have the potential to increase performance and save energy. More specifically, the improved performance of multi-cores can be exploited with race-to-halt strategy – i.e., run at maximum performance until the result is ready and then enter a sleep mode – or a slowdown strategy – i.e., choose the minimal frequency/voltage operating point that meets the deadline.

In this thesis, the student should investigate race-to-halt and slowdown strategies in the context of ML-applications on multi-core ULP devices. The student should first parallelize the SeeDot framework on the prototype ULP device provided by Nordic Semiconductor to investigate the relationship between key parameters such as the number of cores, clock frequency, and supply voltage. The student should then use these results as a basis for performing an architectural exploration to determine under which conditions race-to-halt and slowdown strategies are more appropriate.

Abstract

Internet of Things (IoT) sees billions of ultra-low-power (ULP) devices deployed into all parts of our society. Such devices perform a variety of functions, periodically reading in data through their sensors, possibly processing this data in some way and transmitting them before the period is over and finally entering a sleep mode. These devices pose an interesting intersection of conflicting objectives. They often sport minimal resources when it comes to memory and energy budgets, while at the same time being increasingly often times tasked with running compute-intensive machine learning (ML) applications that must adhere to real-time performance requirements. In recent times increasingly more ULP devices also come equipped with multiple processing cores that increase the computing capabilities, but challenge the energy budget. It is therefore important that these devices utilize their resources in terms of clock frequency, supply voltage and the number of cores in such a way as to minimize their energy consumption while still meeting the performance requirements. Allowing ULP devices to perform processing-intensive work locally instead of communicating with Internet servers enable a range of interesting applications that avoid the security concerns and latencies of Internet communication.

In this work we consider slowdown through dynamic voltage-frequency scaling (DVFS) versus race-to-halt as strategies for minimizing energy consumption under performance requirements for a multi-core ULP device in a soft real-time context. We implement three applications that are deployed to a prototype provided by Nordic Semiconductor. The first makes use of the SeeDot framework for deploying quantized neural networks (QNNs) on ULP devices and the other two are matrix multiplication applications. All applications are implemented in a single-core and multi-core context. We measure the performance and energy consumption in terms of clock frequency, supply voltage and the number of cores. We also analyze the theoretical benefit of using multiple scaled-down cores versus a single scaled-down core.

We find that a slowdown strategy using the lowest configuration that still meets the performance requirement minimizes energy consumption the most, and that supporting *dynamic* scaling of supply voltage and clock frequency might involve more engineering work without any real benefit to energy savings. This assumes that the workload performed within the time period does not exhibit significant variation. We also find that a race-to-halt strategy allows the device to finish as early as possible by consuming the most energy without improving the performance of the application in any meaningful way given the performance requirement. In the analysis of using multiple scaled-down cores versus a single scaled-down core we found that there is potential for performance improvements and significant energy savings when using multiple scaled-down cores.

Sammendrag

Tingenes Internett (Internet of Things (IoT)) gjør at milliarder of ultra-lav-effekt-enheter (ultra-low-power (ULP) devices) blir deployert i alle deler av samfunnet vårt. Slike enheter utfører en rekke funksjoner, periodisk leser inn data fra sensorene deres, potensielt prosesserer og sender denne dataen videre før de går inn i en sovemodus. Disse enhetene representerer en interessant krysning av motsigende mål. De har ofte minimalt med ressurser når det kommer til minne og energibudsjett, samtidig som de i større grad enn før får i oppgave å utføre beregningsintensive maskinlæringsapplikasjoner (machine learning (ML) applications) som må holde seg innenfor sanntids-ytelseskrav. I det siste har ULP-enheter også blitt utstyrt med flere prosesseringsenheter som øker beregningsevnene, men som også utfordrer energibudsjettet. Det er derfor viktig at disse enhetene utnytter sine ressurser med tanke på klokkefrekvens, forsynningsspenning og antall kjerner på en slik måte at de minimierer energiforbruket samtidig som de møter ytelseskravene. Å tillate ULP-enheter å utføre prosesserings-intensivt arbeid lokalt i stedet for å kommunisere med Internett-servere muliggjør en rekke interessante applikasjoner som unngår sikkerhetshensyn og forsinkelser assosiert med Internett-kommunikasjon.

I dette arbeidet betrakter vi slowdown gjennom dynamisk spennings- og frekvensskalering (dynamic voltage-frequency scaling (DVFS)) kontra race-to-halt som strategier for å minimere energiforbruket gitt ytelseskrav for en flerkjernet ULP-enhet i en myk sanntidskontekst. Vi implementerer tre applikasjoner som blir deployert på en prototype fra Nordic Semiconductor. Den første bruker SeeDot-rammeverket for å deployere kvantiserte nevralnett (quantized neural networks (QNNs)) på ULP-enheter og de to andre er matrisemultiplikasjon-applikasjoner. Alle applikasjonene er implementert i en en- og flerkjernet kontekst. Vi måler ytelse og enerigforbruk relatert til klokkefrekvens, forsynningsspenning og antall kjerner. Vi analyserer også den teoretiske gevinsten av å bruke flere nedskalerte kjerner i stedet for en enkelt nedskalert kjerne.

Vi finner at en slowdown-strategi som bruker den laveste konfigurasjonen som fortsatt møter ytelseskravet minimerer energiforbruket mest, og at å støtte dynamisk skalering av forsyningsspenning og klokkefrekvens kan kreve ingeniørarbeid uten å bidra til å redusere energiforbruket i noen særlig grad. Dette antar at den utførte arbeidslasten innenfor tidsperioden ikke endrer seg vesentlig. Vi finner også at en race-to-halt-strategi gjøre at enheten avslutter raskest mulig ved å forbruke mest energi uten å forbedre ytelsen på en meningsfull måte gitt ytelseskravet. I analysen av å bruke flere nedskalerte kjerner i stedet for en enkelt nedskalert kjerne så finner vi at det er potensial for ytelsesforbedringer og signifikante energibesparelser ved å bruke flere nedskalerte kjerner.

Preface

This work represents the masters thesis written during my last semester in order to fulfill the graduation requirement at the Norwegian University of Science and Technology (NTNU) Computer Science — Computers and System Software. This thesis is prepared individually with the supervision of Associate Professor Magnus Jahre and in cooperation with Nordic Semiconductor. The work was carried out from January to June 2020.

The work of implementing, running and measuring the applications we concern ourselves with on the prototype was generally challenging. New problems surfaced seemingly every day, stemming from various limitations in the prototype and false assumptions about how it worked. This struggle has to a great extent revealed for me the magic of how computers work under the hood. And when these issues were slowly getting resolved, suddenly the COVID-19 pandemic hit the world which created exceptional circumstances for us all. As we depended on a physical device and all work was conducted in the Nordic Semiconductor offices in Trondheim, the nation-wide regulations prohibited us from physical access. It was a joyous moment when we were told after many weeks that we could borrow the necessary equipment such that we could conduct the remaining experiments and complete the work with the data we needed.

I would like to thank Associate Professor Magnus Jahre for his much-valued support and guidance during this work. I also wish to thank Cletus Pasanha and Frode Pedersen from Nordic Semiconductor for answering our many questions and providing valuable support in using and understanding their prototype. Further I wish to express gratitude towards Nordic Semiconductor for providing equipment that allowed us to conduct our experiments on a physical device, and also for making the necessary arrangements in the face of the COVID-19 pandemic in order for us to finish our experiments on time.

I would also like to thank my friends, and partner, Ingunn, who have listened to my rantings and often made me realize something without uttering a single word. And to my parents, for always believing in me and supporting me in whatever way possible.

Eirik Vale Aase

Trondheim, June 11th, 2020

Contents

Assignment Description **i**

Abstract **ii**

Sammendrag **iii**

Preface **iv**

Contents **v**

List of Figures **ix**

List of Tables **xi**

List of Algorithms **xiii**

Acronyms **xv**

Glossary **xvii**

1 Introduction **1**
1.1 Internet of Things 1
1.2 Soft Real-Time ULP Devices 2
1.3 Example Scenario 3
1.4 Assignment Interpretation 4
1.5 Contributions 4
1.6 Chapter Outline 5

2 ML on ULP Devices **7**
2.1 ML Basics and SeeDot 7
2.2 Matrix Multiplication 8
2.3 Alternative Parallel Matrix Multiplication Algorithms 9

2.4 SeeDot . . . 10

3 Applications on the Nordic Semiconductor Prototype **13**
3.1 SeeDot . . . 13
3.1.1 Single-Core . . . 13
3.1.2 Multi-Core . . . 14
3.2 Distributed Matrix Multiplication . . . 16
3.3 Ideal Matrix Multiplication . . . 18

4 Experimental Setup **21**
4.1 Prototype . . . 21
4.2 Scenario . . . 22
4.3 Transferring Images to Prototype for Classification . . . 23
4.4 Configuring Supply Voltage and Clock Frequency . . . 23
4.5 Operating Points . . . 25
4.6 Measurements . . . 27
4.6.1 Execution Time . . . 27
4.6.2 Current Consumption . . . 27
4.6.3 Power Consumption . . . 28
4.6.4 Energy Consumption . . . 29

5 Results **31**
5.1 Execution Time . . . 31
5.2 Power Consumption . . . 33
5.3 Energy Consumption . . . 33
5.4 Architecture-Specific Challenges . . . 35
5.5 Sensitivity Analysis . . . 36

6 Minimizing Energy Consumption in Soft Real-Time Systems **37**
6.1 Single-Core Execution . . . 37
6.1.1 Rate of Change for Execution Time, Power and Energy . . . 37
6.1.2 Simulated Sleep Energy with Soft Real-Time Deadline . . . 39
6.1.3 Slowdown versus Race-to-halt . . . 41
6.2 Multi-Core Execution . . . 43
6.3 Energy Proportionality . . . 46

7 Related Work **49**
7.1 ML Applications on ULP Devices . . . 49
7.2 Task Scheduling in Soft Real-Time Systems . . . 50
7.3 Slowdown versus Race-to-halt . . . 51

8 Conclusion and Future Work **53**

Bibliography **55**

Appendices **59**

A Effect of the COVID-19 Pandemic **59**

B Multi-Core Analysis Table **61**

C Source Code **63**

List of Figures

1.1 Typical life cycle of IoT devices 2
1.2 Three strategies for saving energy in computing systems 3
1.3 Outline of chapters 6

2.1 Matrix in row-major order 9
2.2 High-level overview of the SeeDot pipeline 11

4.1 Block diagram of the compute domain 22
4.2 Typical system configuration 23
4.3 Block diagram of the experimental setup 25

5.1 Execution time versus clock frequency 32
5.2 Power consumption versus supply voltage 34
5.3 Energy consumption versus supply voltage 35

6.1 Step-wise change of performance, power and energy consumption . . 38
6.2 Energy consumption with soft real-time deadlines 40
6.3 Energy savings versus supply voltage 42
6.4 Scaled-down multi-core energy savings versus scaled-down single-core 45

List of Tables

4.1 Key numbers for the compute domain 21
4.2 Operating points 26

6.1 Equivalent multi-core performance for a single-core configuration . . 44

B.1 Scaled-down multi-core energy savings versus scaled-down single-core 61

List of Algorithms

1 `main` function of the SeeDot single-core application 14
2 `main` function of the SeeDot multi-core application 15
3 Excerpt from `seedot_fixed.c` of the SeeDot multi-core application 16
4 Matrix multiplication applications 17
5 Transferring features and label to prototype 24

Acronyms

k-**NN** *k*-Nearest Neighbors. 7

CNN convolutional neural network. 10

DMA direct memory access. 46

DNN deep neural network. 50

DOSC digital oscillator. 21–24, 26

DSL domain-specific language. 10, 13

DVFS dynamic voltage-frequency scaling. ii, iii, 2, 4, 37, 41–43, 50, 51, 53

IoT Internet of Things. ii, iii, 1

ML machine learning. ii, iii, xvii, 1, 4, 5, 7, 8, 10, 11, 13, 22, 49, 50, 53

QNN quantized neural network. ii, iii, 49, 50

ULP ultra-low-power. ii, iii, 1–5, 7, 10, 11, 22, 46, 47, 49, 50

Glossary

ProtoNN ML algorithm used to generate a corresponding ProtoNN ML model that can be deployed on resource-constrained devices. 7, 10

SeeDot High-level framework used in this work to guide the production of ML models that are small enough to run on resource-constrained devices, as well as a compiler that compiles SeeDot programs into efficient C code. ii, iii, ix, xiii, 4, 5, 7, 8, 10, 11, 13–18, 23, 26, 28, 31–36, 39–42, 50, 53

Chapter 1

Introduction

This introductory chapter frames the context of this work and lists the objectives in the form of concrete tasks and the resulting contributions.

1.1 Internet of Things

Internet of Things (IoT) represents a technological shift in which everyday objects are connected to networks, i.e. the Internet, and are able to perform tasks that require computational effort and artificial intelligence. A typical life cycle can be seen in Figure 1.1 — typically such devices would read input from its environment, perform some action and enter a sleep mode before restarting the process (Sethi and Sarangi, 2017). *Act* means to possibly process data and then transmit them. Processing data can be done in a multitude of ways, and an emerging candidate is machine learning (ML) (Din et al., 2019). IoT devices can be deployed into a variety of domains, like smart houses, health care and the industry (Khan et al., 2020). There are currently billions of IoT devices connected to the Internet, and it is expected that this number will increase by the billions in the coming years (Din et al., 2019; Khan et al., 2020).

The environment in which IoT devices is deployed may significantly affect the properties of the device, like the physical size. Because of this they might be heavily resource-constrained in terms of memory and energy budget, having as little as a few KiB of RAM and a current consumption in the order of a few µA or mA depending on the operating mode (Sethi and Sarangi, 2017; Atmel, 2015; Gopinath et al., 2019). Because of this we will refer the IoT devices we consider in this work as ultra-low-power (ULP) devices.

The development of ULP devices require a strong emphasis on energy-aware resource consumption. A way of characterizing the resource consumption is with the concept of energy proportionality as defined by Barroso and Hölzle (2007). The idea is that a system uses few to none resources when the workload is little to none, and

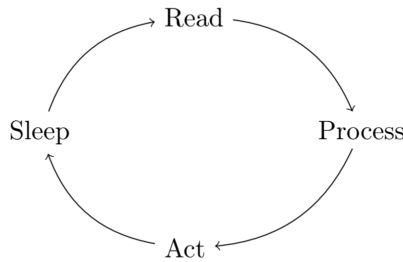


Figure 1.1: Typical life cycle of IoT devices. A sleeping device is notified at regular intervals to read data from its sensors. These data are processed and acted upon before the device enters a sleep mode again.

consumes proportionally more resources as the workload increases. Barroso and Hölzle argues that servers designed with energy proportionality in mind would see significant energy savings. Energy proportionality should therefore be a goal for ULP devices.

1.2 Soft Real-Time ULP Devices

There is a subset of ULP devices that run real-time applications. This means the device must adhere to certain performance requirements that depend on the type of real-time context. For instance, the device might have an imposed deadline. This deadline can either be a soft or hard real-time deadline. An example of the former might be a wearable gesture-based application and the latter a flight control system. Missing a single soft real-time deadline might not be a problem, but repeatedly missing deadlines might degrade the performance. This is in contrast to a hard real-time deadline, where missing a deadline might be fatal. In this work we concern ourselves with soft real-time applications. The energy consumption of real-time applications is of utmost importance. One must reduce the energy consumption as much as possible to lengthen the battery life of the device, but still have enough resources to complete the execution of tasks before the deadline.

There are traditionally two strategies for reducing the energy consumption of ULP devices: Slowdown through dynamic voltage-frequency scaling (DVFS) or race-to-halt. Race-to-halt simply means that the device uses all available resources to finish program execution as quickly as possible and then entering a sleep mode (Das et al., 2015; Imes and Hoffmann, 2015). When it comes to slowdown there are subtle nuances. Traditionally it is defined as using a scaled-down configuration of clock frequency and supply voltage that allows the device to meet the performance requirements (Das et al., 2015; Imes and Hoffmann, 2015). In a soft real-time context the device might finish before the deadline even on the lowest configuration, and as such we also have the possibility of the device entering a sleep mode as it is unnecessary to idle when program execution has completed. These three strategies are seen in Figure 1.2. It is not apparent which of these strategies is the optimal

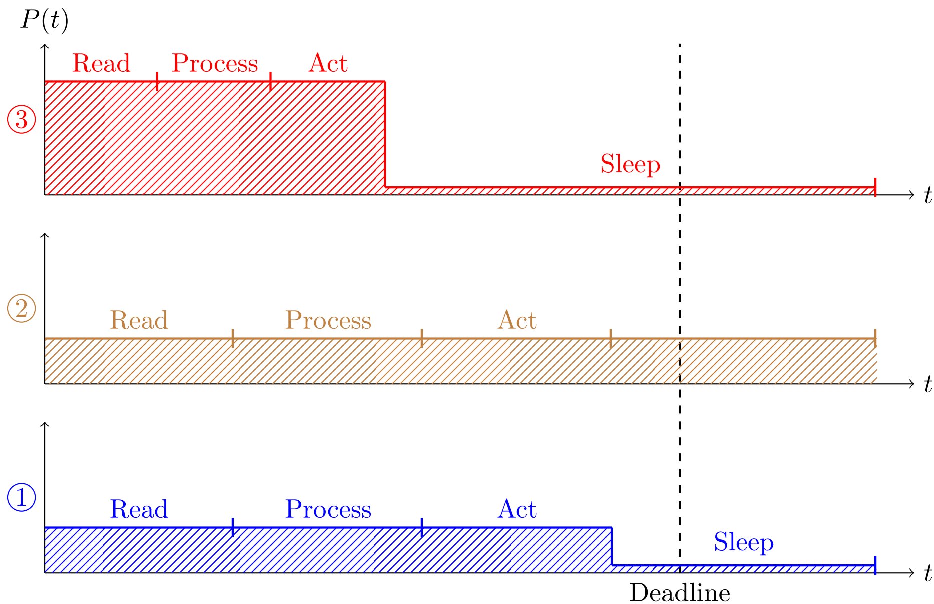


Figure 1.2: Three strategies for saving energy in computing systems. Power consumption versus time. 1) Slowdown through DVFS and sleeping after program execution. 2) Slowdown through DVFS without sleeping after program execution. 3) Race-to-halt.

one, as it might depend on the device architecture and executed applications (Das et al., 2015; Imes and Hoffmann, 2015).

An added dimension of minimizing energy consumption through the use of the slowdown or race-to-halt strategies is the use of multiple processing cores. The idea is that instead of pushing the device to the limit on the highest configuration in terms of clock frequency and supply voltage, you divide the work between multiple cores that operate on a lower configuration. Given that the application is sufficiently parallelizable, the multi-core application would then increase performance and reduce the energy consumption compared to the single-core application.

1.3 Example Scenario

We will now exemplify the discussion above with a scenario of multiple ULP devices equipped with sensors and varying amounts of available resources working together to compute some task.

There are several examples of ULP devices being used in the health industry. One example is the use of tiny monitoring patches that can be worn on the skin and used for continuously monitoring vital aspects of the patient (Sethi and Sarangi, 2017). Even though it is only intended to be used for a few days, it will still have to consume little energy and it will not have the ability to perform any significant

computations. If we compare this device to the life cycle in Figure 1.1 we see that the device would have to read data from its sensors regularly, say the blood pressure, and then act upon these data by wirelessly transmitting them to the patient or doctor before going back to sleep.

One can also imagine that multiple sensors attached to the patient can be used for collecting multiple types of data and based on the combined data stream make inferences about the condition of the patient. To process this data stream we also imagine there being another ULP device that could be carried by the patient that collects this data and makes use of a ML application that is trained to recognize certain patterns.

1.4 Assignment Interpretation

Based on the assignment description we establish a set of tasks T for this work

(T_1) We will parallelize the SeeDot example application on a single prototype with multiple processing cores provided by Nordic Semiconductor.

(T_2) We will perform experiments measuring performance and energy consumption in terms of clock frequency, supply voltage and the number of processing cores.

(T_3) We will discuss whether a slowdown through DVFS or race-to-halt strategy is more appropriate given an objective of minimizing the energy consumption under a performance requirement like a soft real-time deadline in relation to the prototype architecture.

We note that when the assignment description describes the strategies of slowdown versus race-to-halt, it is slightly inaccurate in that it does not open for allowing the slowdown strategy to sleep at the end of program execution before the deadline is reached. In this work we will explicitly open for this strategy in our analysis.

1.5 Contributions

Based on the defined tasks T we list the set of contributions C from this work

(C_1) We parallelized the SeeDot example application and two matrix multiplication applications on a single prototype with multiple processing cores provided by Nordic Semiconductor.

(C_2) We performed experiments measuring performance and energy consumption in terms of clock frequency, supply voltage and the number of processing cores.

(C_3) We discussed whether a slowdown through DVFS or race-to-halt strategy was more appropriate given an objective of minimizing energy consumption with a performance requirement like a soft real-time deadline in relation to the

prototype architecture. We found that a slowdown strategy with the lowest configuration that still met the deadline minimized energy consumption the most. The race-to-halt strategy consumed the most energy. We also found significant energy saving opportunities when using multiple scaled-down cores compared to a single scaled-down core.

Because of the COVID-19 pandemic that broke out during the work on this masters thesis, we lost access to the Nordic Semiconductor equipment for several weeks. See Appendix A for more details.

1.6 Chapter Outline

Figure 1.3 outlines the chapters and the related contributions in this work. Chapter 1 introduces the problem domain, tasks and contributions. Chapter 2 introduces the field of ML and the SeeDot framework. Chapter 3 presents the applications that have been implemented on the Nordic Semiconductor prototype. Chapter 4 details the prototype and the experimental setup. Chapter 5 presents the results obtained from the experiments. Chapter 6 discusses slowdown versus race-to-halt strategies as means for saving energy in soft real-time systems in a single-core and multi-core context and the concept of energy proportionality. Chapter 7 presents the related work when it comes to deploying ML applications on ULP devices, task scheduling in real-time systems and the question of slowdown versus race-to-halt. We present our conclusion and the proposed future work in Chapter 8.

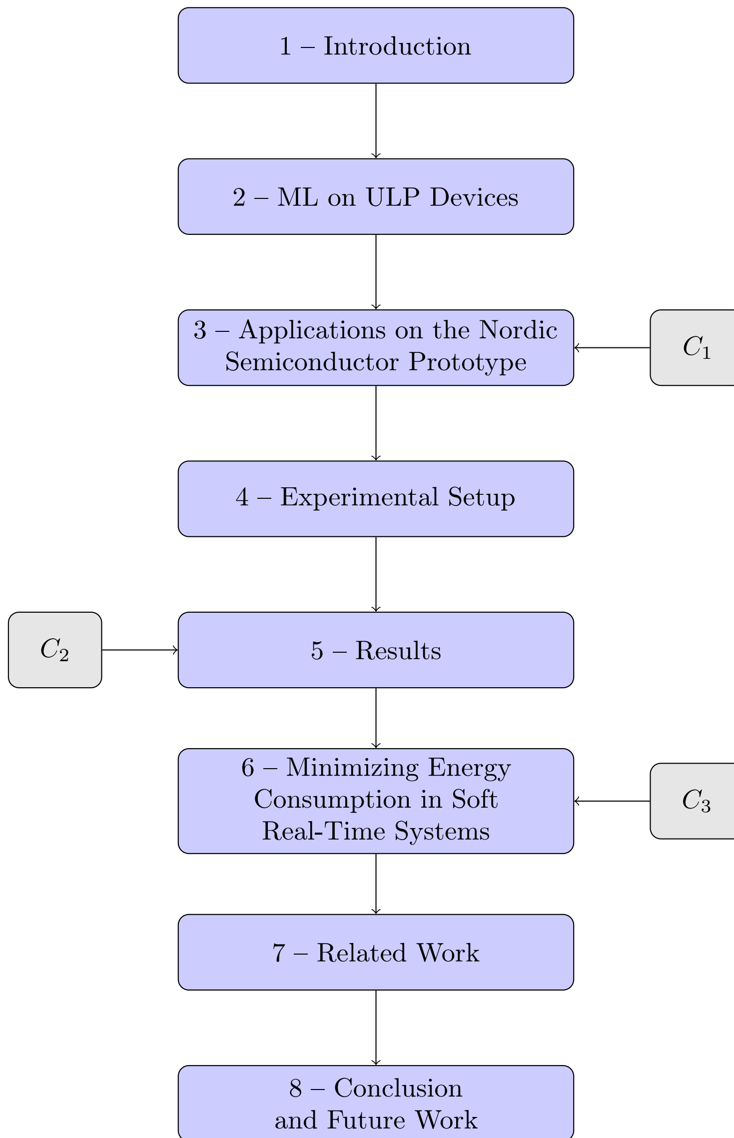


Figure 1.3: Outline of chapters. Blue boxes are chapters in order. Gray boxes with C_1, C_2 and C_3 refer to the individual contributions in Section 1.5.

Chapter 2

ML on ULP Devices

In this chapter we summarize the relevant aspects of ML and the important matrix multiplication operation that is central to the parallelization of our applications. We also explain the aspects of SeeDot that enables us to run a ML application on a ULP device like the Nordic Semiconductor prototype.

2.1 ML Basics and SeeDot

ML is the process of making a program improve its performance from experience, i.e. it should learn from data. We refer to the application scenario and data as an *example* (or *data point*) that consists of *features*, usually denoted as a vector x and *labels* denoted as a vector y. The learning and evaluation steps are separated, and we use different examples for each, i.e. we use a *training set* when learning and a *testing set* when evaluating the performance. A data set, either testing or training, is then a collection of many examples. The *task* is executed with a ML algorithm (Goodfellow et al., 2016).

There are different types of ML algorithms, and one of these are supervised learning algorithms. Supervised means that we instruct the ML algorithm to associate some input x with some output y, such that y_i corresponds to the correct output for a given example or data point x_i. When the ML task is about figuring out what kind of input belongs to a specific output category, we call this a *classification task* (Goodfellow et al., 2016). In the case of SeeDot, the example is a vector of features, where each feature x_i is the individual pixels that make up the image of a handwritten digit. Then, the classification task is the process of classifying which digit the image represents. We use SeeDot with the ProtoNN algorithm, which is a kind of k-Nearest Neighbors (k-NN) inspired ML algorithm that is designed for supervised classification tasks.

Since each digit is on the form $y \in \{1, 2, \ldots, 9\}$, we call this specific classification task for a *multiclass classification*. This is in contrast to a binary classification,

where the output value can be one of two values (Goodfellow et al., 2016; Murphy, 2012). The performance of the ML algorithm on a classification task is usually measured in terms of *classification accuracy*, i.e. the fraction of correct classifications (Goodfellow et al., 2016).

The neural network in the SeeDot example application performs multiple matrix operations, for instance matrix multiplication and matrix subtraction. We implement an application that only performs matrix multiplication, and such an application is referred to as a *kernel* in ML. Matrix multiplication is used extensively in computer graphics, robotics, graph theory and image processing (Alqadi and Aqel, 2008; Akpan, 2006).

2.2 Matrix Multiplication

We will now define the concept of a matrix and how matrix multiplication works mathematically. We will then look at how matrices are stored in memory and how we enable multiple processing cores to cooperate on a matrix multiplication task in parallel without interfering with each other.

Equation 2.1 defines a $m \times n$ matrix $\mathbf{A}$ to be a (possibly) multi-dimensional list with m rows and n columns. A one-dimensional list, also called an array, is then defined to be a $1 \times n$ matrix, with n elements.

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \tag{2.1}$$

We consider two matrices $\mathbf{A}$ and $\mathbf{B}$ of dimensions $m \times n$ and $n \times p$, respectively. Then $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ is the multiplication of $\mathbf{A}$ with $\mathbf{B}$ resulting in the $m \times p$ matrix $\mathbf{C}$. This operation is only valid if $\mathbf{A}$ has as many columns as $\mathbf{B}$ has rows. $\mathbf{C}$ is then defined to be

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \begin{cases} 1 \leq i \leq m \\ 1 \leq j \leq p \end{cases} \tag{2.2}$$

We will now look at how matrices are stored in memory. We will consider two simplified 2×2 matrices $\mathbf{A}$ and $\mathbf{B}$ defined as

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \tag{2.3}$$

There are traditionally two ways in which matrices like $\mathbf{A}$ and $\mathbf{B}$ are stored in memory. *Row-major order* involves storing the elements of each row next to each

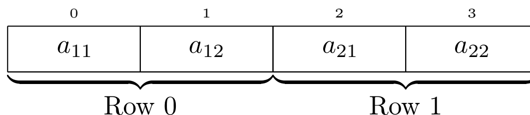


Figure 2.1: Matrix **A** laid out in memory, row-major order

other in memory. This is shown in Figure 2.1. The other way of storing these matrices in memory is called *column-major order* and involves storing the elements of each column next to each other in memory. If we were to store matrix **A** in Figure 2.1 in column-major order instead of row-major order, then the two elements a_{12} and a_{21} would switch places. In this work all matrices are stored in row-major order.

We will now show how two cores c_0 and c_1 can cooperate on multiplying two matrices **A** and **B** as in Equation 2.3. The output matrix **C** is calculated as

$$c_{ij} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} \tag{2.4}$$

Since we have two cores available, ideally we want to distribute the computations evenly between the two. Therefore we would like to have c_0 be responsible for the first half of **C**, and c_1 the other half. For c_0 to compute the first row of **C**, it would only have to read in the first row of **A**, the whole of **B** and write to the first row of **C**. The same is true for c_1, only it would read the second row of **A** and write the second row of **C**. The two cores can then write to their respective rows in **C** in parallel without interfering with each other. If we consider arrays then dividing the work between the two cores simplifies to splitting the arrays evenly and tasking each core with computing half of the arrays.

2.3 Alternative Parallel Matrix Multiplication Algorithms

A naive single-core matrix multiplication algorithm has an algorithmic complexity of $\mathcal{O}(n^3)$ which is clearly inefficient (Akpan, 2006). Numerous sequential algorithms have been devised that improve upon the naive solution, for instance Strassens Algorithm that reduces the complexity to $\mathcal{O}(n^{\log 7}) \approx \mathcal{O}(n^{2.8074})$ (Strassen, 1969). Still, an enormous work has been done to develop parallel matrix multiplication algorithms.

Cannon's algorithm is based on matrix decomposition in which calculations are performed on sub-matrices and shifted a number of times between processors. Although it requires $n \times n$ cores to multiply two $n \times n$ matrices, the storage requirement is constant and independent of the number of cores (Alqadi and Aqel, 2008; Gupta and Sadayappan, 1994). *Ho-Johnsson-Edelman's algorithm* is a variation of Cannon's algorithm with the same storage requirement, but adopted for

hypercubes, otherwise called n-cubes. Ballard et al. (2012) improves upon Strassens algorithm with the *Communication-Avoiding Parallel Strassen (CAPM) algorithm* by being optimal in terms of inter-processor communication.

2.4 SeeDot

In this section we review SeeDot. First we will give a high-level overview of the SeeDot framework and its purpose. Then we will detail the required steps from obtaining a data set to deploying a ML application on a device and consider a day in the life of such a device. Last we will detail the aspects of SeeDot that enables us to deploy ML models on ULP devices.

There are two sides of SeeDot. First, SeeDot is a domain-specific language (DSL) that can be used to succinctly write ML inference algorithms that in general-purpose programming languages would otherwise be more complex. Such an algorithm could be a convolutional neural network (CNN) performing some classification task. Second, SeeDot is a compiler that compiles SeeDot programs into efficient C++ source code that can be deployed to ULP devices with as little as 2 KiB of memory. In addition there are tools implemented for generating ML models with several different ML algorithms (Gopinath et al., 2019).

The required steps involved when uploading an application to a device is presented in Figure 2.2. First one must obtain a data set and use one of the provided SeeDot tools to generate a trained ML model with one of the included ML algorithms. Then a SeeDot program must be written that achieves the objective of the application that is to be run on the ULP device. The SeeDot compiler will then use the generated ML model, the SeeDot program and the data set to generate source code for different platforms like an Arduino or X86. This source code could then be uploaded to the device.

After deployment the ULP device will have a life cycle similar to what is shown in Figure 1.1. We assume we have deployed a ML inference application that takes as input an image in the form of a set of pixels that is read from a camera that is connected to the device. This is represented as the *Read* stage in Figure 1.1. The *Process* stage will run the compiled SeeDot inference algorithm on this image in order to classify what kind of digit the image represents. The *Act* stage displays the result of the classification to the interested parties. In the end the device goes back to sleep in anticipation of a new classification task.

We will now briefly summarize the aspects of SeeDot that enables ML models to be deployed to ULP devices for efficient program execution. For more details, see the specialization project Aase (2019) preceding this masters thesis. First, SeeDot internally uses ML algorithms (like ProtoNN) that produce ML models that can fit in as little as 2 KiB of memory and hence can be deployed to ULP devices. Second, they overcome the lack of floating-point support on many ULP devices by converting the ML model to use fixed-point instead of floating-point numbers. Third, they improve the efficiency of the exponential function as this traditionally

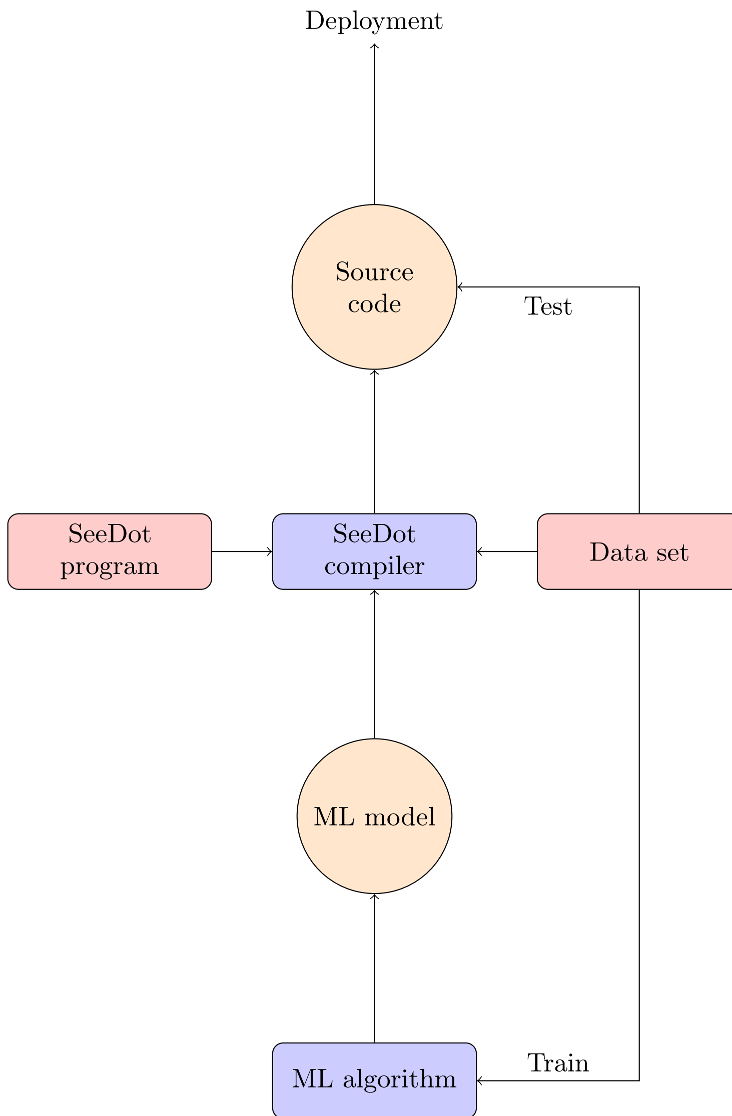


Figure 2.2: High-level overview of the steps involved in the SeeDot pipeline when generating source code that can be deployed to ULP devices. *Deployment* leads to the typical life cycle of ULP devices as presented in Figure 1.1. Red boxes are parameters. Orange boxes are generated content. Blue boxes are programs that generate other content.

incurs performance loss and high memory consumption (Gopinath et al., 2019; Aase, 2019).

Chapter 3

Applications on the Nordic Semiconductor Prototype

This chapter will describe the three applications that have been implemented in this work, specifically the ported SeeDot example application and the two matrix multiplication applications.

3.1 SeeDot

3.1.1 Single-Core

In this subsection we briefly review the steps that were taken to port the SeeDot example application to the prototype and run it on a single processing core. For more details, see Aase (2019).

We mention in Section 2.4 that SeeDot is both a DSL that can express ML inference algorithms and a compiler that compiles such programs to efficient C++ code. The compiler supports in theory generating source code for the X86 and Arduino platforms, but was hard-coded to generate code for the latter, so we rewrote parts of the compiler to generate X86 source code, i.e. regular C++ code, as this was syntactically closer to the C code that was allowed to run on the prototype. After this process we ended up with a set of files: The main entry point, `main.c`, a C++ representation of the neural network, `seedot_fixed.cpp` and a library for performing linear algebra operations, `library.h` and `library.c`. We then proceeded to translate these C++ files to regular C source code. We only needed to significantly rewrite the `main.c` file as it assumed data sets were read in locally on the executing device. In our case, storing data sets of several MiB on the prototype was impossible due to its limited memory. We therefore rewrote it to wait for data to be transferred from a host computer to RAM. This simplified the original `main.cpp` file to what can be seen in Algorithm 1.

Algorithm 1 `main` function of the SeeDot single-core application

```
1:  b ← true, F ← [], L ← 0
2:
3:  procedure MAIN
4:      while true do
5:          while b = true do
6:          end while
7:          r ← CLASSIFY(F)
8:          b ← true
9:      end while
10: end procedure
```

The single-core application works as follows: The core waits for the features and label to be sent from the host computer to shared RAM. When the transmission is done, it will read this data in from RAM and execute the classification function. The classification function is the main entry point to the SeeDot inference algorithm, i.e. the `seedot_fixed.c` file. The original inference algorithm performs a set of operations corresponding to the layers in the neural network, and this file does the same by calling functions that hook into the linear algebra library, i.e. `library.h` and `library.c`. In the end the classification function produces the classification result, i.e. which digit it thinks is depicted in the image, and goes back to waiting for the next pair of features and label.

3.1.2 Multi-Core

We made several changes to the ported SeeDot single-core application to make it execute on and take advantage of multiple cores. The relevant algorithms and source code are given in Algorithm 2, Algorithm 3 and Appendix C.

We first review the `main.c` file given in Algorithm 2, which is the main entry point for the SeeDot multi-core application. It is similar to the main entry point of the SeeDot single-core application as given in Algorithm 1. The main difference is that we now read in the identification number of the core that the program is currently executing on. We can do this because the same application is uploaded to all cores that are intended to cooperate on the task. Based on this identification number we define a different code path for the two cores. The core deemed as *master* will call into the SeeDot neural network file as was done in the single-core application in order to start the classification task. The other core, called the *slave*, calls into its own waiting flow.

We will now look at the `seedot_fixed.c` file, which represents the neural network. The source code in the multi-core case can be seen in Appendix C. Basically the master core will perform the same operations as the core in the single-core application. At the same time there are certain operations that the two cores are intended to cooperate on, i.e. matrix multiplication and matrix subtraction. We

Algorithm 2 `main` function of the SeeDot multi-core application

```
1: b ← true, F ← [], L ← 0
2:
3: procedure MAIN
4:     if READID = masterID then
5:         while true do
6:             while b = true do
7:             end while
8:             r ← CLASSIFY(F)
9:             b ← true
10:        end while
11:    else if READID = slaveID then
12:        SLAVE
13:    end if
14: end procedure
```

replace the original calls to these operations in `seedot_fixed.c` with calls to a specific wrapper function that the master core executes. In this function it will make available the input data for which both cores will use to compute the operations and assign an even workload to each core. The way in which we make the master and slave cooperate correctly on a task is through the use of a shared variable declared in shared RAM that signals if the slave is ready or done computing its assigned task. When the master has copied the relevant data, it signals through the shared variable that both can start to compute. At this point the slave has been busy-waiting on the shared variable since the program started executing. Both cores will at this point call into the original linear algebra functions with their assigned workload to compute their respective result. Upon returning from the function the master will wait to collect the result from the slave, and the slave will resume waiting for the next operation for which it should cooperate on with the master core.

To actually divide the work between the master and slave we make use of the fact that even though SeeDot has been implemented by Dennis et al. (2019) to handle matrices of a higher dimension than one, in practice we only deal with arrays in the matrix operation functions, which we noted in Section 2.2. Therefore we give a balanced number of iterations of the matrix multiplication and subtraction to the master and slave. For each iteration and core we write the result to a unique position in the output array that resides in shared memory.

The library functions for the matrix multiplication and subtraction are both functions in the linear algebra library provided by SeeDot. The source code for these functions can be found in the Appendix C and is based on the source code in the SeeDot framework implemented by Dennis et al. (2019). In short we changed the two library functions given in `library.h` and `library.c` to accept a lower and upper bound corresponding to the set of iterations that was given to each of the cores. The rest of the relevant library functions are identical to what was implemented in

Algorithm 3 Excerpt from `seedot_fixed.c` of the SeeDot multi-core application

```
1:  a ← [], b ← [], c ← [], d ← []
2:
3:  procedure MatSubWait
4:      while d = false do
5:      end while
6:      MatSub(a, b, c, 13, 25)
7:      d = false
8:  end procedure
9:
10: procedure MatMulCNWait
11:     while d = false do
12:     end while
13:     MatMulCN(a, b, c, 5, 10)
14:     d = false
15: end procedure
16:
17: procedure Slave
18:     while true do
19:         MatSubWait
20:         MatMulCNWait
21:     end while
22: end procedure
23:
24: procedure optimizeFunction(A, B, C, f)
25:     Copy(a, A), Copy(b, B)
26:     d = true
27:     if f = 0 then
28:         MatSub(a, b, c, 0, 13)
29:     else if f = 1 then
30:         MatMulCN(a, b, c, 0, 5)
31:     end if
32:     while d = true do
33:     end while
34:     Copy(C, c)
35: end procedure
```

Dennis et al. (2019).

3.2 Distributed Matrix Multiplication

We experienced and observed both application- and architecture specific challenges when we implemented and measured the SeeDot multi-core application. We discuss the former in Section 5.1 and the latter in Section 5.4, but we summarize them

Algorithm 4 Matrix multiplication applications

```
1:  a ← [], b ← [], c ← [], d ← false
2:
3:  procedure COMPUTE(A, B, C, l, u)
4:      for j ← l, j < u do
5:          for i ← 0, i < 500 do
6:              C[j] = A[j] · B[j] + i
7:          end for
8:      end for
9:  end procedure
10:
11: procedure MASTER(A, B, C, l, u)
12:     COPY(a, A), COPY(b, B)
13:     d ← true
14:     COMPUTE(a, b, c, l, u)
15:     while d = true do
16:     end while
17:     COPY(C, c)
18: end procedure
19:
20: procedure SLAVE(l, u)
21:     while true do
22:         while d = false do
23:         end while
24:         COMPUTE(a, b, c, l, u)
25:         d ← false
26:     end while
27: end procedure
28:
29: procedure MAIN
30:     if READID = masterID then
31:         A ← [], B ← [], C ← []
32:         INITIALIZE(A, B, C)
33:         MASTER(A, B, C, 50, 100)
34:     else if READID = slaveID then
35:         SLAVE(0, 50)
36:     end if
37: end procedure
```

here briefly. First, the lack of a functioning data cache in the compute domain resulted in the cores constantly reading data from and writing data to shared RAM, which caused memory contention. Second, the amount of computations done in the matrix multiplication and subtraction functions in the SeeDot application was not significant enough, such that the overhead of using multiple cores outweighed any

benefit of dividing the work between the cores. Because of this low computational load we decided to extract the matrix multiplication operation inside the SeeDot neural network into its own application. We could then regulate the computational load as needed. The pseudo code for the matrix application can be found in Algorithm 4 and the source code can be found in Appendix C. We note that the term *distributed* in this context means that it uses shared RAM. This is in contrast to the *ideal* version that only performs computations locally without sharing the result with other cores through shared RAM. Also note that the source code for the distributed single-core and multi-core matrix multiplication versions are identical except for two differences: First, in the multi-core version the master will start the slave core with a special instruction (in the single-core version the code path for the slave is never used, as the core is never started). Last, the master is assigned either all of half of the iterations depending on the number of cores.

The distributed matrix multiplication application is implemented in a single file. We will first detail the program flow when using a single core, and then using multiple cores. First, when only using a single core the master will assign the entire workload to itself. The objective is to multiply two matrices and write the result to a third matrix. Since it is the only core in this case, it will only copy the data to and from memory to mimic the original SeeDot application. Also, since the prototype has limited memory, there is a limit as to how big the matrices can be declared. We increase the computational load by performing each matrix multiplication a fixed number of times. When using two cores, the slave will immediately call into a busy-wait state with half of the workload assigned to itself. Here it will wait for the master to copy the input arrays to shared memory and signal that it can start computing the result. The master will wait for the slave to finish and then collect the result.

3.3 Ideal Matrix Multiplication

We implement an ideal matrix multiplication application because of the significant memory contention when using shared memory. In order to reason about the effects of memory contention, we modify the matrix multiplication application to only use a little piece of memory that is local and only seen by each of the cores. In this application there is no way to make the cores cooperate, as shared RAM is not used in any way. We therefore make both cores perform the same computations without exchanging the result with shared RAM. We will therefore call this the *ideal* version of the matrix multiplication application. The source code for this application can be seen in Appendix C.

The ideal single-core application behaves similarly to the distributed single-core application, so we do not provide an explanation of this application. On the other hand, the ideal multi-core application is different from the distributed multi-core application, which warrants some explanations.

Since each core performs computations in its own local memory, and no data is

transferred to shared RAM, there was no way for the cores to cooperate on the task. Therefore the two cores perform the same workload as is done in the ideal single core application. The effect is therefore that the ideal multi-core application performs twice the amount of work that the ideal single-core application does. We also note that the occurrences of `memcpy` in the source code in Appendix C does not mean that data is actually shared with RAM. It still copies data to and from its own local memory. Other than this the ideal single-core and multi-core versions are similar.

Chapter 4

Experimental Setup

In this chapter we detail the prototype provided by Nordic Semiconductor, the way in which we perform our experiments and the measurements we collect.

4.1 Prototype

We use a prototype provided by Nordic Semiconductor in this work. This prototype is configured with three modules that each is equipped with one or more processing cores, cache and RAM. One of the modules is called the *compute* domain, and this is the domain we execute our applications on in this work. A high-level overview of the compute domain can be seen in Figure 4.1 and some key numbers for it can be seen in Table 4.1

The clock frequency of a core is determined by an oscillator that can produce waveforms like the square signal. Such an oscillator can either be analog or digital. The advantage of a digital oscillator (DOSC) is that it is more accurate and can produce a larger variety of waveforms (Chamberlin, 1985). In Figure 4.1 we see such a DOSC. In this case it is a high-speed clock that we can use to produce clock frequencies that reach about 700 MHz. We see in Figure 4.1 that it provides the clock signal to all four cores.

All cores share the same RAM with a single data bus that can be accessed by one

Number of cores	4
RAM	64 KiB
Instruction cache (per CPU)	16 KiB
Data cache (per CPU)	8 KiB

Table 4.1: Key numbers for the compute domain

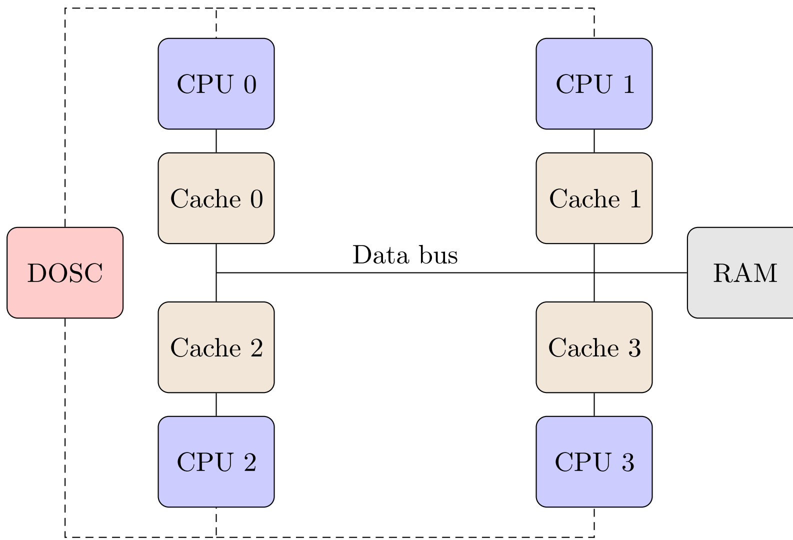


Figure 4.1: Block diagram of the relevant components in the compute domain. The DOSC clocks the four cores. Each core has a corresponding cache. A shared data bus is used by all cores to exchange data with shared RAM.

core at a time. All cores also have a unique identification number that can be programmatically read to check which core the application is currently executing on. All cores also share the same supply voltage that we can scale. This is in contrast to the cache and RAM that each have their own supply voltage that are in both cases configured to be 0.8 V and are not scaled.

The way in which we debug applications running on the prototype is unchanged from Aase (2019) and we summarize them here briefly. One option is to print to standard output and capture this on the host computer. Another is to assert GPIO pins that can be observed through an oscilloscope or sampled with a Logic Pro Logic Analyzer and inspected on the host computer.

4.2 Scenario

The scenario that we model in the experimental setup is unchanged from Aase (2019) and shown in Figure 4.2. To summarize, we model a ULP device that operates under a performance requirement like a soft real-time deadline. The device executes a ML application that is tasked with classifying images captured from a connected camera.

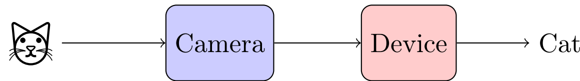


Figure 4.2: A typical system configuration. A camera captures images that are transmitted to a device for classification.

4.3 Transferring Images to Prototype for Classification

Although the way in which we transfer images from the host computer to the prototype is similar as what was done in Aase (2019), we add additional details below. The updated algorithm is given in Algorithm 5.

First, for this work we updated the implementation to be able to send the full data set of 2007 samples or a randomized subset of 100 samples. Second, to find out which destination memory addresses to send the features and label to, the streaming program parses a `map` file resulting from compiling the application source code for information about the starting memory addresses of the variables that we intend to populate. The SeeDot single-core and multi-core applications use several global variables that for instance represent the features and label that we use for a particular measurement. This information in addition to hard-coded information about the length of the features array is used to send the features and label to the correct memory addresses so that they can be used in the classification task.

4.4 Configuring Supply Voltage and Clock Frequency

On the prototype there are two ways of controlling the clock frequency, both with their advantages and disadvantages. We refer to these two approaches as *external* and *internal clocking* depending on where the clock signal is provided from. The supply voltage is configured in the same way for both approaches. We will first review how the supply voltage is configured, then the clock frequency.

In order to configure the supply voltage on the prototype we use a device called the N6705B DC Power Analyzer. This device has four supply voltage outputs that can be wired onto specific pins on the prototype. There are specific pins for different circuitry, and the power analyzer can be used to provide a supply voltage to these pins. We use one output port on the power analyzer and wire it onto the pin that supplies voltage to the cores, cache, data bus and RAM control signals. There is a separate supply voltage at 0.8 V provided to the cache and RAM, but we do not scale these in our experiments.

We clock the clock frequency internally by setting the supply voltage in the range of $[0.5, 0.9]$ V. We then use this supply voltage as an input parameter to the DOSC to set the clock frequency. The DOSC can be configured to run in an open or closed loop. The main difference between these two modes is that the open loop is a free

Algorithm 5 Transferring features and label to prototype

```
 1: procedure MAIN(a)
 2:     F ← READFEATURES
 3:     L ← READLABELS
 4:     b ← READBUSYWAITADDRESS
 5:     if a = subset then
 6:         i ← RANDOMIZEDSUBSET(100, 2007)
 7:         F ← F(i)
 8:         L ← L(i)
 9:     end if
10:     for all i ∈ S do
11:         while READMEM(b) ≠ true do
12:         end while
13:         WRITEMEM(F_i)
14:         WRITEMEM(L_i)
15:         WRITEMEM(b, false)
16:     end for
17: end procedure
```

running clock in which the clock frequency can drift. This is in contrast to the closed loop in which it will lock to a specific clock frequency multiplier with respect to a reference clock (which by default is configured to be 16 MHz), a period and a supply voltage. Since we needed to know an accurate value for the clock frequency we decided to use the closed loop mode. Since we also wanted the widest range of clock frequencies, we chose a period of $p = 2$ as that would yield the highest clock frequency possible for a given supply voltage. A larger period would yield a lower clock frequency.

The main advantage of internal clocking is that it is accurate. After configuring the supply voltage and the period one can be sure that the clock frequency that results from configuring the DOSC is actually what is used on the prototype. This makes the measurements as accurate as possible. Another advantage is that you get a much larger range of clock frequencies to run on the prototype, above 700 MHz at the most. The main disadvantage is that it is not as easy to use, since one may have to recompile and re-upload the application if there is a specific clock frequency one wants configure the prototype with.

External clocking relies on an external Python application written by Nordic Semiconductor for setting the clock frequency from the host computer instead of through code on the prototype. This was the way we configured the clock frequency in Aase (2019). The application allows a numerical value like 16000000 to be written to the memory address that stores the numerical value of the reference clock.

The main advantages of external clocking are its ease of use and flexibility. We only need to input some new numerical value and click a button to have it written to memory instead of recompiling and uploading the application and have it changed

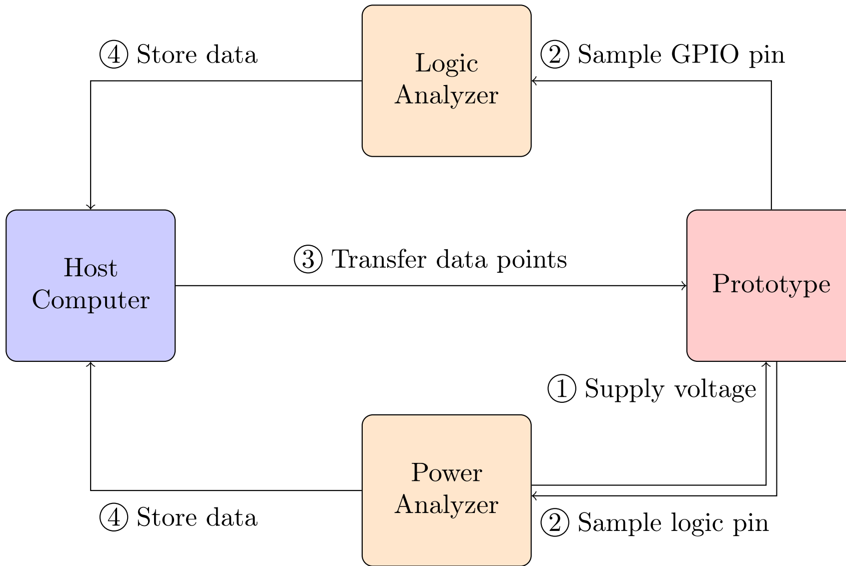


Figure 4.3: Block diagram of the experimental setup. The flow of conducting an experiment is as follows: Step 1 provides a supply voltage to the prototype. Step 2 starts sampling the GPIO and logic pins for execution time and current consumption, respectively. We start transferring data points to the prototype in step 3. When the measurement is completed we store the data in step 4. The relevant internal components of the *Prototype* box (specifically the compute domain) can be seen in Figure 4.1. We omit power supply connections.

immediately. It is also flexible because you can set the clock frequency independent of the supply voltage. The main disadvantage is that it limits the range of clock frequencies you can configure the prototype with, which is in the area of up to 64 MHz. Another disadvantage is that you do not know which supply voltage is needed to configure the prototype with a given clock frequency, even though the logic circuitry in the compute domain is supplied with a certain supply voltage level. This discrepancy might therefore affect the energy consumption measurements.

Given the need for a wide range of clock frequencies and accurate current measurements we opted to clock through internal clocking for our experiments.

4.5 Operating Points

We define an operating point as a pair of supply voltage and clock frequency, $o_i = (v_i, f_i)$. We define the order of operating points based on the value of the supply voltage since this will determine the clock frequency. When referring to operating points, instead of writing out the supply voltage and clock frequency values, we will simply write o_x, where x is a value in the range of $[0, 8]$. The set of operating points can be seen in Table 4.2.

Name	Voltage [V]	Frequency [MHz]
o_8	0.90	708
o_7	0.85	627
o_6	0.80	543
o_5	0.75	457
o_4	0.70	370
o_3	0.65	286
o_2	0.60	206
o_1	0.55	136
o_0	0.50	79

Table 4.2: Operating points, i.e. combinations of supply voltage and clock frequency, used for configuring the prototype for program execution.

We selected the set of operating points based on the need to observe the behavior of the prototype for a wide range of supply voltages and clock frequencies while maintaining program correctness and avoiding damage to the prototype. The clock frequency is configured with the DOSC closed-loop mode with a period of $p = 2$. This means we get the highest clock frequency for a given supply voltage. To determine the maximum supply voltage we looked at previous measurements conducted by Nordic Semiconductor. These measurements suggested that it is possible to configure the prototype to run at a clock frequency approaching 1 GHz, but in order to achieve this we would have to supply a voltage level above 0.9 V. Since such a supply voltage could damage the chip, we decided that this supply voltage would be our upper limit. The generated clock frequency for this supply voltage was 708 MHz.

To determine the lower bound for the supply voltage we conducted initial experiments where we repeatedly decreased the supply voltage and verified that our programs were still executing correctly. In reality this is not a perfect solution, as different applications can have different code paths that fail on different supply voltage levels. In the end we were not able to decrease the supply voltage below 0.5 V for the SeeDot application and the distributed matrix application, and 0.55 V for the ideal matrix application. The generated clock frequencies for these supply voltages was 79 MHz and 136 MHz, respectively.

Given the upper and lower bound for the supply voltage we define a linear function for the supply voltage as $v_i = 0.05i + 0.5, i \in \{0, \ldots, 8\}$.

4.6 Measurements

4.6.1 Execution Time

Execution time is measured with the Logic Pro Logic Analyzer device that is connected to the host computer and prototype as we show in Figure 4.3. In our applications we assert a high signal to the GPIO pin immediately before we start executing the code segment for which we want to measure the execution time for and then a low signal to the GPIO pin immediately after the code segment is done. The data collected can be inspected and exported for further analysis. Our data is stored as two columns where the first is a timestamp x_i and the second the state of the GPIO pin. We therefore measure execution time as the difference in time between the time of the rising edge of a GPIO signal assertion x_i and the corresponding falling edge x_{i+1}, i.e. the positive signal width between the GPIO pin assertions $t_i = x_{i+1} - x_i$.

We also present execution time in terms of what is known as The Iron Law of Performance as stated in Hennessy and Patterson (2017). If we define the instruction count as i, the clock cycles per instruction as `cpi` and the clock cycle time as c, the execution time of a program is calculated as

$$t = i \times \texttt{cpi} \times c \tag{4.1}$$

Clock cycle time can be expressed as $c = 1/f$, where f is the clock frequency. This gives us the following formula for the execution time of a program

$$t = \frac{i \times \texttt{cpi}}{f} \tag{4.2}$$

We see that doubling the clock frequency from f to $2f$ also doubles the performance, i.e. halves the execution time, given that `cpi` is constant in this work. We will assume that $\texttt{cpi} = 1$. Given Equation 4.2 we will expect performance to grow linearly with the clock frequency.

4.6.2 Current Consumption

The N6705B DC Power Analyzer is used to measure the current, and is connected to the prototype in the same way as was detailed earlier in the chapter when describing how to control the supply voltage. This can be seen in Figure 4.3. When connecting the power analyzer to the pin in this manner we are able to capture the current usage for the logic part of the compute domain. The power analyzer samples the current usage at a fixed interval and this data is then exported for further analysis.

To measure the current in the compute domain we connect one of the supply voltage outputs from the power analyzer to the pin on the prototype that provides the supply voltage to the logic circuitry. The logic circuitry is comprised of the four

cores, cache, data bus and RAM control signals. With this setup we are able to measure the total current for the logic circuitry. This is different than what we did in Aase (2019). In that work we also considered two other pins, namely those that provide the supply voltage for the cache and RAM. We believed that the pin we now know to represent the logic circuitry actually was for the cores alone, and that the two other pins measured the current through the cache and RAM. This is partially correct, but we are not interested in measuring on the two other pins as these measure the current inside the memory blocks of the cache and RAM themselves. This current is not the focus of our experiments, only the current through the logic circuitry.

Idle current is measured through turning off all clocks in all modules so that all cores are clock gated. Practically we do this by routing the pin signal for the clock that controls the cores on the chip directly to ground. This is significantly different from the way idle current was measured in Aase (2019), as only a few clocks were clock gated resulting in the idle current measurement taking into account several components that should not have contributed to the idle current at all.

The post-measurement analysis that handles the current for the SeeDot application will analyze it in terms of active periods. These active periods are found by looking for peaks in the current. A peak will correspond to the start of program execution, i.e. classifying a sample. When we find a peak, the active period is defined as starting from the time of the peak and lasting as long as the corresponding program execution time. The average current usage for a measurement is then the average current usage of all active periods.

Finding the average current for the matrix applications is a bit different. In these applications the current consumption when performing the matrix multiplication is actually marginally lower compared to executing the surrounding code. This might be explained by multiplication instructions causing stalls in the pipeline. Therefore we look for local minima in the current measurement. We also perform the same computations a number of times so that we can average over all of them — so we look for as many local minima as the number of times we ran the program, and then we average these values for the final average current consumption.

4.6.3 Power Consumption

We base our definition of static power consumption on the version found in Hennessy and Patterson (2017), which can be stated as

$$P_{\text{static}} = V \times I_{\text{static}} \tag{4.3}$$

Based on Equation 4.3 we see that static power consumption grows linearly with respect to the supply voltage. When it comes to dynamic power consumption we base our definition on the version found in Hennessy and Patterson (2017), although we simplify it by folding the constants 1/2 and the capacitive load into the constant k

$$
\begin{aligned}
P_{\text{dynamic}} &\propto 1/2 \times \text{Capacitive load} \times V^2 \times f \\
&= k \times V^2 \times f
\end{aligned} \tag{4.4}
$$

Equation 4.4 tells us that dynamic power consumption is quadratic with respect to supply voltage and linear with respect to clock frequency.

4.6.4 Energy Consumption

Energy consumption is defined by Nilsson and Riedel (2015) as

$$
E_{\text{static}} = \int_0^t P_s(t)\,dt \tag{4.5}
$$

Since the static power consumption is constant, Equation 4.5 simplifies to

$$
E_{\text{static}} = P_{\text{static}} \times t \tag{4.6}
$$

When it comes to dynamic energy consumption, the definition in Hennessy and Patterson (2017) is unsatisfactory for our case as it does not consider execution time. We therefore define dynamic energy consumption as the product of dynamic power consumption and time

$$
\begin{aligned}
E_{\text{dynamic}} &= P_{\text{dynamic}} \times t \\
&= k \times V^2 \times f \times t
\end{aligned} \tag{4.7}
$$

Chapter 5

Results

In this chapter we characterize the performance, power and energy consumption aspects of the Nordic Semiconductor prototype based on how we model execution time, power and energy consumption in Section 4.6 for a single and multiple cores. We seek to establish if the prototype is behaving as we would expect from the models in Section 4.6.

5.1 Execution Time

A bar plot of execution time versus clock frequency is shown for all operating points, core configurations and applications in Figure 5.1. Measurements for the SeeDot single-core and multi-core applications are represented by the top plot and measurements for the two matrix applications in the bottom plot. The lower x axis positions the clock frequencies of our operating points in Table 4.2 and the upper x axis positions a set of reference clock frequencies in steps of 100 MHz.

We immediately see that the execution times for the multi-core versions of the applications are consistently and significantly higher than the associated single-core execution times. There are both architecture and application-specific reasons for this, of which the former is dealt with in Section 5.4, but we summarize them here briefly. First, two cores accessing shared RAM with a data bus that does not support parallel access to memory creates memory contention. Second, the compute domain lacks a data cache which again leads to memory contention and degraded performance as data must be fetched from memory. There is also a bug in the implementation of processing core arbitration which limits the number of active cores to two, but given the aforementioned problems it seems unlikely that using additional cores would solve these issues. The high SeeDot multi-core execution times might also indicate that there is not a sufficiently high workload in the SeeDot application to benefit from parallelization. This observation is strengthened by the fact that the arrays that are used in computations in the SeeDot application are usually only ten or 25 elements wide.

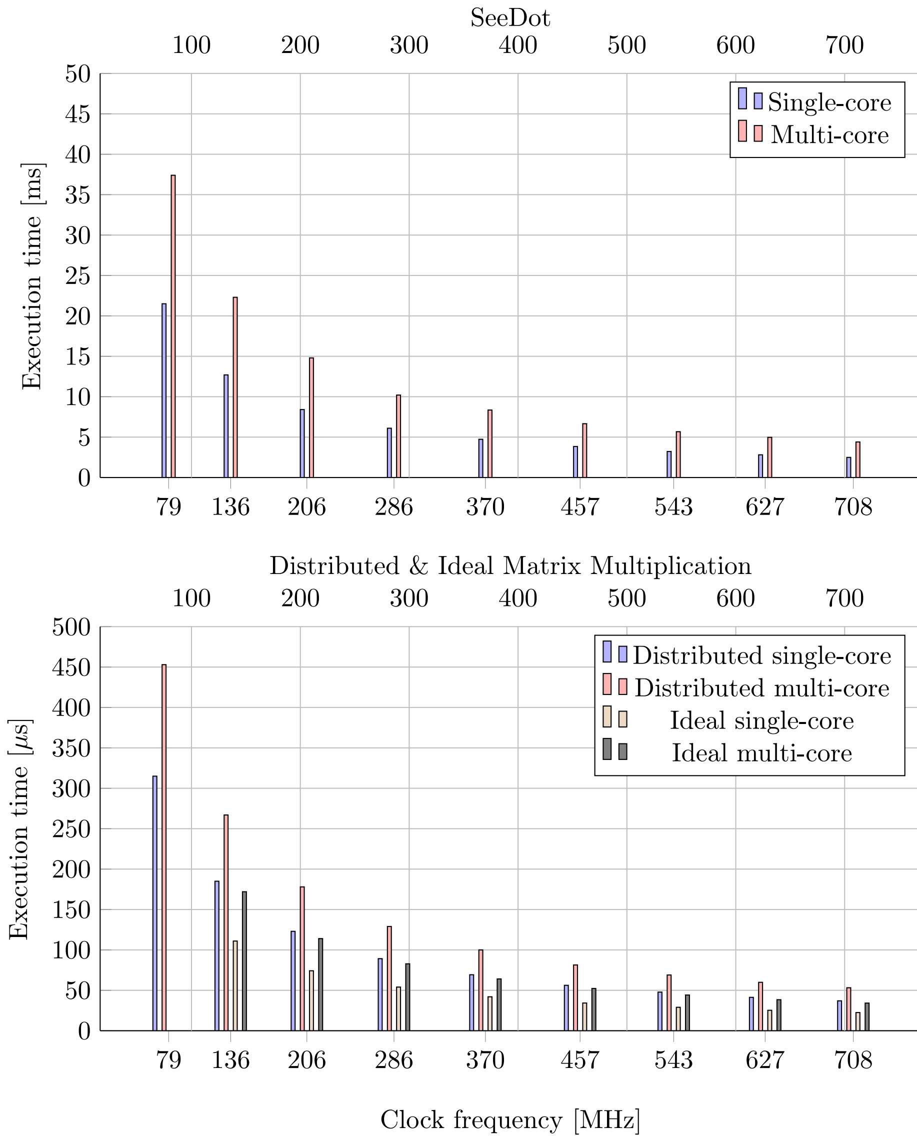


Figure 5.1: Execution time versus clock frequency for the single-core and multi-core versions of the SeeDot application and the two matrix multiplication applications.

We also observe that the single-core execution times suggest that performance in this case is not exactly linear with respect to clock frequency. Especially when the clock frequency grows large we do not see the expected performance improvement. One explanation for this is that we scale the supply voltage to the cores, but not the memory system, which is fixed at 0.8 V. Therefore instructions that compute

results are executed faster, but the memory system works just as fast as before as it is not affected by the increase in clock frequency and becomes a limiting factor in terms of performance. These observations might also suggest that the `cpi` is not one as we assumed, but higher. Multiple cores waiting for each other and stalls in the pipeline from memory requests might contribute to this.

It is also interesting to note that the single-core version of the ideal matrix application executes significantly faster than the distributed single-core matrix application. The difference here is due to regularly copying data to and from shared RAM. Figure 4.1 shows that all cores share the same RAM and uses a shared data bus with only one core allowed to use the data bus at a time, hence the degraded performance. Keeping more of the data local to the cores instead of storing it in shared RAM suggest that we could see improvements to performance and energy consumption.

5.2 Power Consumption

Figure 5.2 presents the total power consumption versus supply voltage for the SeeDot single-core and multi-core applications on the left, and the single-core and multi-core matrix applications on the right. We see that the shape of the total power consumption for all applications resemble the shape of a quadratic function, as we would expect from Equation 4.4.

We note that the difference between the total power consumption for the single-core and multi-core SeeDot applications are quite small, and that this difference is decreasing going from the highest to the lowest supply voltage. This is interesting because it associates a small power consumption expense by adding a second core. On the other hand, the differences between the total power consumption for the single-core and multi-core versions of the two matrix applications are larger than was the case for SeeDot. At the same time we still see the same trend as with SeeDot, that enabling a second core has a higher total power consumption, which we would expect.

Another thing we note about the right figure is that the total power consumption of the ideal single-core matrix application is about the same compared to the distributed single-core matrix application. This is not the case for the two multi-core applications. We noted in Chapter 3 that the two cores in the ideal multi-core matrix application each perform the work equivalent of the corresponding single-core application. This explains the higher total power consumption in the case of the ideal multi-core matrix application compared to the distributed multi-core matrix application, even though the latter share data with RAM.

5.3 Energy Consumption

Figure 5.3 shows the total energy consumption versus supply voltage for all operating points, core configurations and applications. Generally we see that the total energy consumption for the multi-core version of the three applications are significantly

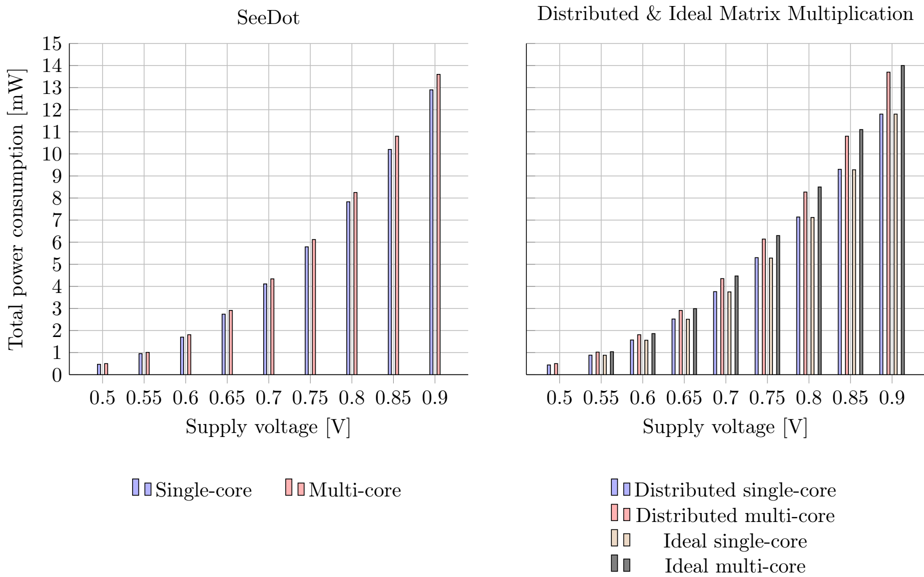


Figure 5.2: Total power consumption versus supply voltage for the single-core and multi-core versions of the SeeDot and the two matrix multiplication applications.

higher than that of the single-core versions. Due to the non-functional data caches, the cores are forced to constantly communicate with shared RAM on a data bus that only supports a single core at a time. Since the multi-core applications actually execute slower than the corresponding single-core applications, and because the total power consumption is higher for the former than the latter, the total energy consumption is also higher for the multi-core applications.

Figure 5.3 also shows that the distributed matrix versions consume significantly more energy than their ideal counterparts. This is expected and tells us that there is significant latency when moving data to and from shared RAM, especially for the multi-core applications that have to deal with significant memory contention. At the same time, the energy consumption of the distributed single-core application is only marginally lower than the energy consumption of the ideal multi-core application. This tells us again that even though using two cores increase the power consumption, the effect of sharing data with RAM is significant.

The data in Figure 5.3 also might suggest that there is an approximately linear trend between energy consumption and supply voltage. We can see this by comparing Figure 5.1 and Figure 5.2. At least a few of the execution time data points seem to follow an approximately linear trend, and the power consumption values seem to follow a quadratic trend. Since energy consumption depends on both execution time and power consumption, Figure 5.3 might then suggest that these contributions to a certain degree cancel for our prototype and implemented applications.

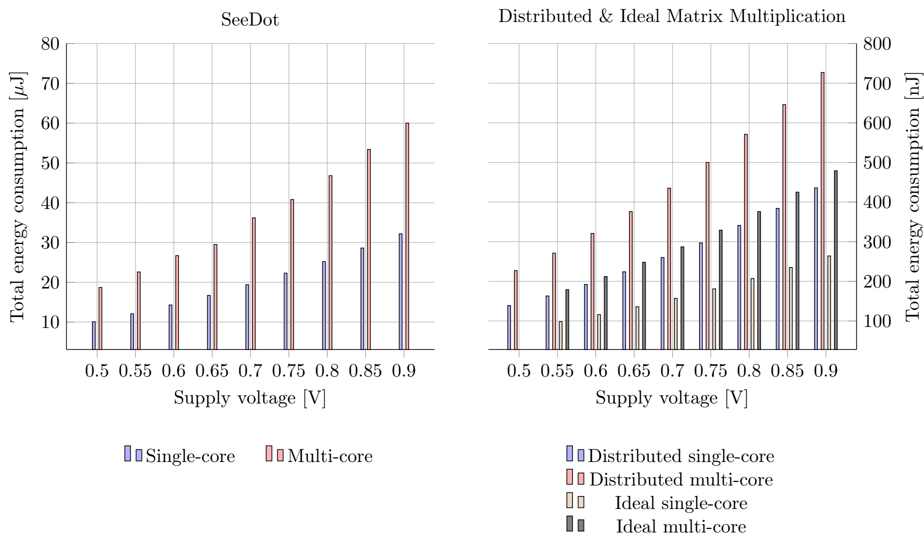


Figure 5.3: Total energy consumption versus supply voltage for the single-core and multi-core versions of the SeeDot and the matrix multiplication applications

5.4 Architecture-Specific Challenges

We found several architecture-specific challenges when implementing the multi-core version of the SeeDot application that had major effects on the performance and energy consumption of the application.

The first issue is related to a bug in the implementation of the arbitration between the four cores in the compute domain. Obviously a single core works fine, as that core will always be granted permission to run. We also found that two cores works fine. When three or four cores were active at the same time, the arbitration mechanism would favor a select few of the cores every time, starving others, making running three or four cores in parallel impossible. We were therefore limited to running only two cores, which in theory would limit the performance and energy benefits. Since we use a prototype in this work, such bugs in various implementations are bound to exist.

The second issue is due to the discovery that the prototype had only partial support for caching in the various modules. Specifically, the data cache for the compute domain were not implemented. A consequence of this is that the cores could not cache the data used when executing our applications, i.e. executing a single instruction would result in fetching the associated data from the shared RAM for all cores. The memory hierarchy serves an important function in the performance of computing systems, and the partial absence of the upper layers have a significant impact on performance (Hennessy and Patterson, 2017).

The absence of a data cache in the compute domain significantly degrades the

performance because the cores share the same RAM data bus. The two cores will make use of a shared variable that indicate whether the slave core can start or is done computing. This was the only reasonable implementation of parallelism on the chip as other traditional approaches for notifying cores like interrupts was not an option. The consequence of this is that when both cores read and write values to the shared RAM, this will create significant memory contention that degrades performance. To remedy this one could increase the bit width of the data bus or implement solutions that would allow multiple cores to access memory at the same time. To improve the scheduling solution one could also look into hardware semaphores that would enable a single core at a time to access a particular memory location. Ideally the other core would then enter a sleep mode to reduce the energy consumption. This might not be the most efficient solution if the waiting period is short, as then a simple busy-wait mechanism could be sufficient. This is because there can be significant overhead in waking up from a sleep mode compared to repeatedly accessing a value in a busy-wait, which might improve performance (Wolf, 2008).

The last architecture-based challenge with achieving energy savings on the prototype that would have helped us in the question of slowdown versus race-to-halt was the lack of a working sleep mode. Traditionally sleep mode can be initiated through a specific sleep assembly instruction. Although executing such an instruction would make the prototype unresponsive, indicating that it was indeed in sleep mode, this had no effect on the current consumption, and as such we could not use it to put the cores to sleep.

5.5 Sensitivity Analysis

In our measurements for the SeeDot application we only perform the classification on a randomized subset of data points taken from the full set of 2007 data points. We do this because each of our 25 measurements for the SeeDot application with the full set of 2007 data points would total $2.5\,\text{h} \times 18 = 45\,\text{h}$. We also had no way of automating the data collection, hence 45 h is the absolute minimum time we would spend. We base our decision on the sensitivity analysis performed in Aase (2019) in which we conclude that using only a subset of 100 samples was appropriate. The sensitivity analysis was conducted based on initial observations during the work on Aase (2019) that execution time and current consumption seemed to be approximately constant for each program execution.

Chapter 6

Minimizing Energy Consumption in Soft Real-Time Systems

This chapter deals with the objective of minimizing energy consumption under a performance constraint like as a soft real-time deadline, and the question of slowdown through DVFS versus race-to-halt. We will also delve into the concept of energy proportionality and see it in relation to that of Barroso and Hölzle (2007).

6.1 Single-Core Execution

6.1.1 Rate of Change for Execution Time, Power and Energy

Figure 6.1 shows the rate of change for execution time, power and energy consumption between all operating points for all applications and core configurations. The points for power and energy consumption are calculated with

$$p_{o_i} = e_{o_i} = \frac{x_{o_i}}{x_{o_{i-1}}} \tag{6.1}$$

and the points for the execution time is calculated with

$$t_{o_i} = \frac{x_{o_{i-1}}}{x_{o_i}} \tag{6.2}$$

where x can be the execution time, power consumption or energy consumption for operating point o_i and o_{i-1}. There is no improvement for the first operating point, hence we exclude those. Also the operating point of o_0 on the ideal matrix multiplication was not working, so those points are also excluded and the plots start at operating point o_2.

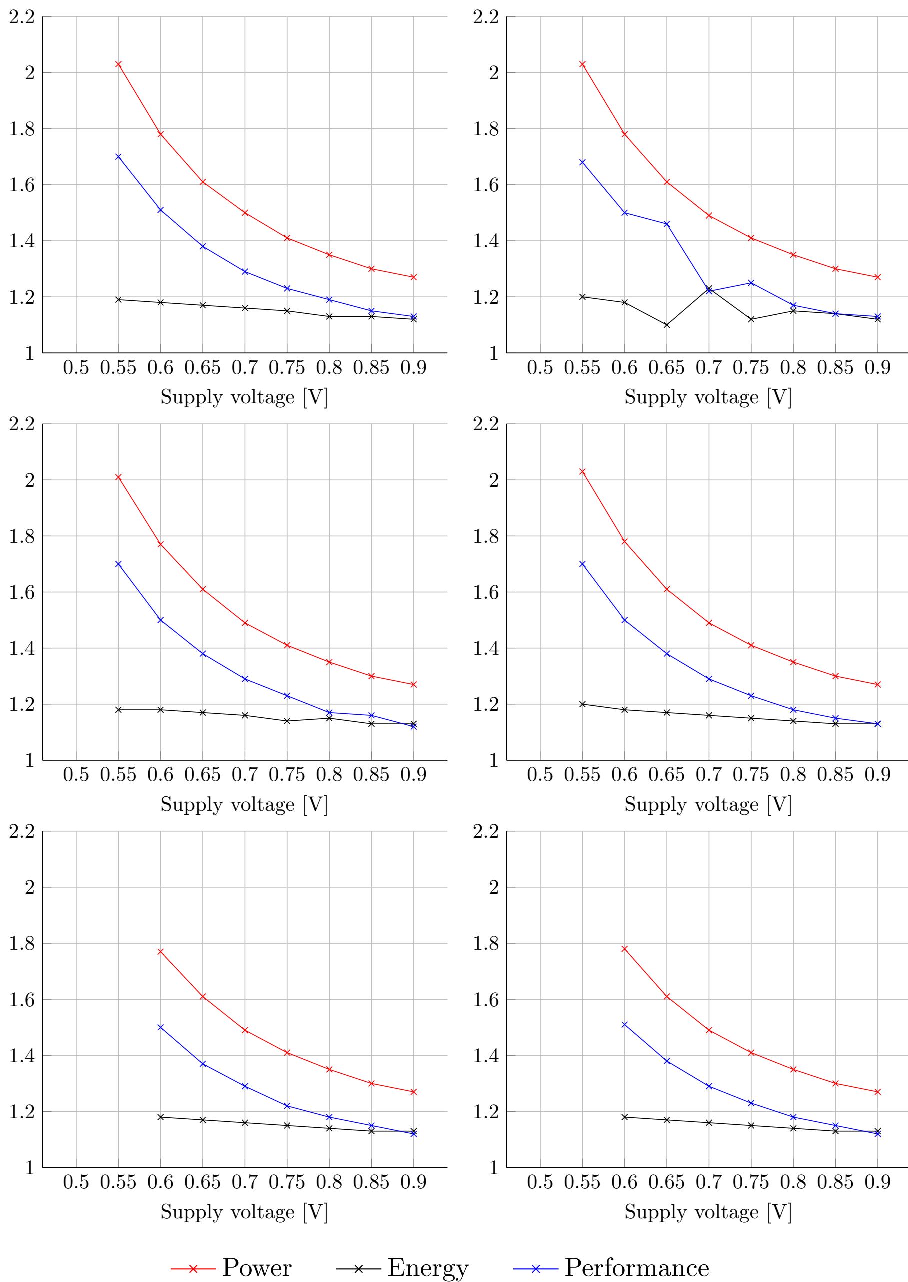


Figure 6.1: Step-wise change in performance, power and energy consumption between operating points for all applications.

We see in Figure 6.1 that both execution time and power consumption has a high step-wise increase for the lowest operating points, but that this increase is decreasing towards the higher operating points. This is not the case for the energy consumption — the general trend is that the step-wise increase in total energy consumption is about the same between all neighboring operating points, but slightly decreasing towards higher operating points. From this we can see that there is a decreasing performance benefit of scaling up the supply voltage, and the energy consumption can be significantly reduced by scaling down the supply voltage.

6.1.2 Simulated Sleep Energy with Soft Real-Time Deadline

Figure 6.2 shows the total energy consumption in the time period limited by three soft real-time deadlines (25 ms, 15 ms and 5 ms) for the SeeDot single-core application. Running one of the other applications would not change the observations based on this type of plot as the trends in execution time and power and energy consumption are the same. In this plot the total energy consumption is the sum of the total energy consumption during program execution and the total energy consumption while in sleep mode that lasts for the remainder of the time window after program execution completes. That is why the bars to the left of the soft real-time deadline has a sleep energy component while the ones to the right have not, since they do not finish before the deadline and we therefore do not model them as sleeping. The energy consumption while in sleep mode is calculated as

$$\begin{aligned} E_{\texttt{sleep}} &= P_{\texttt{sleep}} \times t_{\texttt{sleep}} \\ &= v_i \times i_{\texttt{sleep}} \times t_{\texttt{sleep}} \end{aligned} \tag{6.3}$$

where v_i is the supply voltage that is used during program execution and the sleep mode, $i_{\texttt{sleep}} = 1\,\text{mW}$ is the simulated current consumption in sleep mode and $t_{\texttt{sleep}}$ is the difference in time between the soft real-time deadline and the program execution time, i.e. how long the program sleeps. The current consumption during sleep mode is set as to make sure that the sleep mode preserves the necessary context in memory such that subsequent program executions finish correctly. $v_i \times i_{\texttt{sleep}}$ will range from 0.5 mW to 0.9 mW, which is a reasonable assumption as several sleep modes implemented for the ATmega328P microcontroller, which is similar to the devices we consider in this work, operate within such a range (Atmel, 2015).

We first note that with a soft real-time deadline of 25 ms all operating points complete program execution within the deadline. If we reduce the soft real-time deadline to 15 ms then the slowest operating point, i.e. o_0, no longer completes program execution in time. Reducing the soft real-time deadline even more, to 5 ms, means that only the five fastest operating points are able to complete execution in time.

We also see that sleep energy makes a significant contribution to the total energy consumption with a deadline of 25 ms. In this case the fastest operating points will

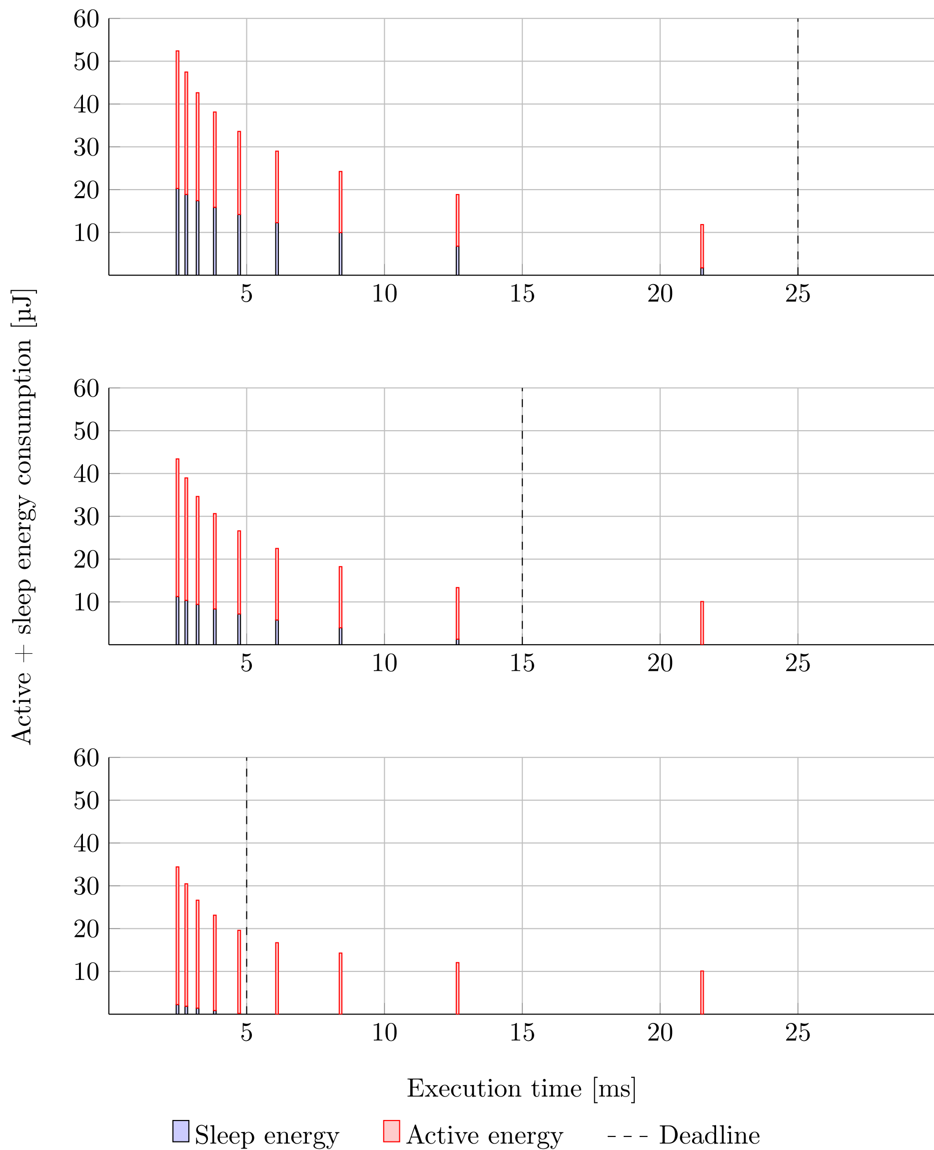


Figure 6.2: Total energy consumption during varying time windows limited by a soft real-time deadline for the SeeDot single-core application. Sleep energy is the bottom part of bars and program execution energy is the top part of bars. Dashed lines represent deadlines. Bars to the right are not feasible as they miss the performance requirement.

have to wait significantly longer than the operating points that finish closer to the deadline. There is therefore a significant difference in total energy consumption between the fastest and the slowest operating points, and this is also the case for

the two other deadlines of 15 ms and 5 ms.

6.1.3 Slowdown versus Race-to-halt

For the devices we consider in this work the primary objective should be minimizing the total energy consumption while at the same time performing well enough to complete execution before the soft real-time deadline. As we mentioned in Chapter 1, there are mainly two ways to do this: Slowdown through DVFS or race-to-halt. Slowdown means scaling down the supply voltage and clock frequency while still performing within the deadline. Race-to-halt means using all available resources to finish the work as fast as possible and then enter a sleep mode. One of our objectives in this work is to decide whether a slowdown or race-to-halt strategy is most appropriate given the applications we have implemented and the prototype architecture. Therefore we will now consider both of these alternatives.

Race-to-halt is represented as the bars for the highest operating point in Figure 6.2. We see that the energy consumption in sleep mode accounts for nearly 40 % of the total energy consumption with a deadline of 25 ms. This operating point has the highest total energy consumption of all operating points in Figure 6.2. This is *also* the case in Figure 5.3 where we assume that the energy consumption in sleep mode is zero. This operating point only achieves half of the objective we have defined — it finishes before the deadline, but in doing so it consumes the most energy of all operating points.

Slowdown through DVFS is represented by the bars for all operating points below the fastest operating point. It is important to note that in our definition of slowdown through DVFS we assume that also these programs, and not just the fastest operating point, enters sleep mode if the program finishes before the deadline. Since the program has completed its task, there is nothing else to do, i.e. it is meaningless to stay awake and waste energy. For these operating points we scale down the supply voltage and clock frequency. Each operating point o_i spends less time in sleep mode compared to the higher operating point o_{i+1} because the former finishes later than the latter. Each time we scale down we see an improvement in energy consumption while still performing equally good, where equally good means completing before the deadline. Scaling down to the slowest operating point yields the program execution that consumes the least amount of total energy, and this is the case for all deadlines. In other words, the slowest operating point o_0 is the configuration that minimizes the energy consumption the most while still making the deadline. Based on our objective this operating point is therefore preferred, hence slowdown through DVFS (where we use the lowest configuration) is the preferred strategy compared to a race-to-halt strategy given our applications and prototype. The slowdown strategy will compared to race-to-halt prolong the battery life of the device, hence add value for the end-user without compromising on the performance requirement.

We have observed that slowdown through DVFS (using the lowest configuration) is the preferred strategy compared to a race-to-halt strategy for the SeeDot single-

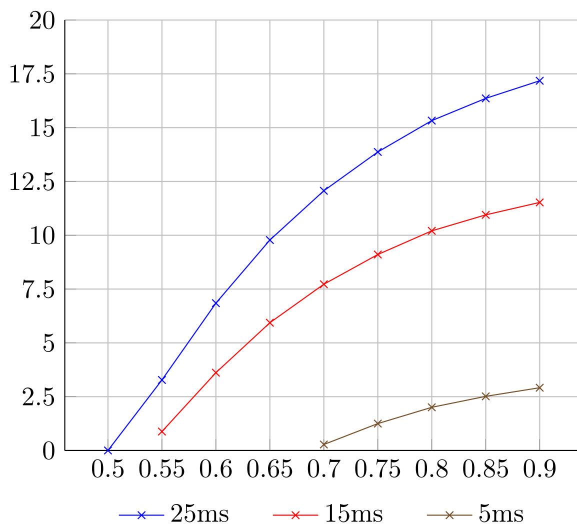


Figure 6.3: Energy savings in percentage versus supply voltage for the three deadlines we considered in Figure 6.2.

core application. We will now delve into the architecture-specific aspects of the prototype that contribute to this observation. Figure 5.3 shows that the distributed matrix multiplication application consumes about 1.65× as much energy as the corresponding ideal matrix multiplication application. This suggests that there is a significant cost of exchanging data with shared RAM given Figure 4.1 (in which we see that all communication with shared RAM happens over the shared data bus). Because of the lack of a working data cache in the compute domain and the fact that all memory requests goes to shared RAM, memory requests are expensive on this prototype. Also consider the fact that we only scale the clock frequency of the cores and not the memory system, which has a fixed supply voltage. This means that when scaling up the clock frequency, the cores might execute computational instructions faster, but the memory instructions do not see any improvement. Therefore the cores will wait increasingly longer for the memory system to complete memory requests, and the memory system will end up as a limiting factor in the performance of the applications, and also waste energy.

In addition, a working sleep mechanism could reduce the energy consumption, but Figure 5.3 tells us that even with an assumed sleep energy consumption of zero the lowest operating point still consumes the least amount of energy. Implementing interrupts that would wake the device up could also be beneficial, but only if the workload of the SeeDot application was higher, as the wake-up time from sleep modes can be significant and could potentially outweigh the program execution time.

Still, DVFS implies that the application can dynamically scale the supply voltage

and clock frequency in order to save energy. In the sensitivity analysis in Aase (2019) we establish that execution times can be considered constant for each program execution in a measurement. Such low or practically nonexistent variations in execution time suggests that supporting DVFS is not necessary as one would just find the lowest operating point for which the program finishes before the deadline and use this configuration instead of spending engineering hours on implementing DVFS. On the other hand, there might be scenarios in which the full force of DVFS can prove beneficial. If there is sufficient variation in execution time one cannot rely on a single operating point to be sufficient for all program executions. Opting for a high operating point, say o_8 in Figure 6.2, clears short deadlines but proves wasteful in terms of energy consumption for long deadlines. On the other hand, using a slow configuration like o_0 wastes little energy with slow deadlines, but might potentially miss several shorter deadlines, which will degrade performance. Having the ability to dynamically scale the voltage and frequency might then prove beneficial because with a short deadline you can ideally switch to a faster operating point to make the deadline, and with a long deadline you can switch to the slowest configuration to potentially save energy. Das et al. (2015) shows that this is indeed possible, although in the case of multi-threaded workloads.

In Figure 6.2 we use a sleep mode with a supply voltage equal to the supply voltage during program execution. An alternative is to use a sleep mode with a different supply voltage than during program execution. In Figure 6.3 we plot how much energy is saved through using the lowest supply voltage when sleeping instead of the one used during program execution versus the supply voltage for the three deadlines in Figure 6.2. We see that the savings are larger with the longer deadlines and towards the higher operating points. This makes sense as these operating points finish faster, therefore sleep longer and use higher supply voltage levels during program execution. With shorter deadlines the savings are slimmer, with just above 2.5 % at the most for a deadline of 5 ms and a supply voltage of 0.9 V. We therefore see that there is definitely some potential for saving energy when supporting several supply voltage levels, but that these energy savings occur at the higher operating points and that the required engineer work might outweigh the energy savings benefit.

6.2 Multi-Core Execution

The results in Chapter 5 show that for our applications and prototype the multi-core versions of the applications increase execution time, power consumption and energy consumption. There we also discuss the reasons as to why that is. In this section we will analyze the theoretical benefit of parallelization. We will consider the ideal matrix multiplication as it does not exchange data with shared RAM and hence do not include any overhead. *Single-core* and *multi-core application* in this section will refer to the ideal matrix multiplication unless explicitly mentioned. We note in Chapter 3 that the two cores in the multi-core application each perform the work equivalent of the single-core application. This means that the total work

Table 6.1: Equivalent multi-core performance for a given single-core configuration. f_s is the performance requirement, i.e. the single-core performance. The three cf_s columns states that if a multi-core has c cores where each is configured with a f_s clock frequency, the performance is given by cf_s. The cf_s column provides equivalent or better performance if $cf_s \geq f_s$. For instance, four cores configured with $f_s = 136$ MHz provides about equivalent performance to a single core with $f_s = 543$ MHz because each only does 1/4 of the work, hence we can compare program executions with these two configurations. If $cf_s < f_s$ then the multi-core application does not perform well enough. The o_0 operating point does not work on the ideal matrix multiplication application.

o_i	v	f_s	Multi-Core		
			$2f_s$	$3f_s$	$4f_s$
o_8	0.90	708	1416	2124	2832
o_7	0.85	627	1254	1881	2508
o_6	0.80	543	1086	1629	2172
o_5	0.75	457	914	1371	1828
o_4	0.70	370	750	1110	1480
o_3	0.65	286	572	858	1144
o_2	0.60	206	412	618	824
o_1	0.55	136	272	408	544
o_0	0.50	N/A			

performed in the multi-core application is twice that of the single-core application. Also the energy consumption in the multi-core case is approximately twice that of the single-core case, so we assume that 98 nJ is the energy consumption for a single matrix multiplication operation no matter how many cores are used. We will assume that the application is fully parallelizable such that increasing the number of cores will have a corresponding increase in performance. We will consider the energy savings for using two, three and four cores.

The three left-most columns in Table 6.1 shows our existing operating points. The three right-most columns under the *Multi-core* header shows that if c cores are configured with the operating point in the same row (v, f_s), then an equivalent single-core execution would have to be the value under cf_s. For instance, running four cores on o_1 would yield slightly better performance than a single core running on o_6. Since the energy consumption of the former is significantly lower than the latter, we would prefer using multiple cores.

Figure 6.4 plots E_m/E_s versus speedup for different combinations of single-core and multi-core operating points. Points are annotated with (c, v_m) where c is the number of cores and v_m is the supply voltage that each core runs on. Points that are lower in the plot saves more energy compared to the single-core configuration

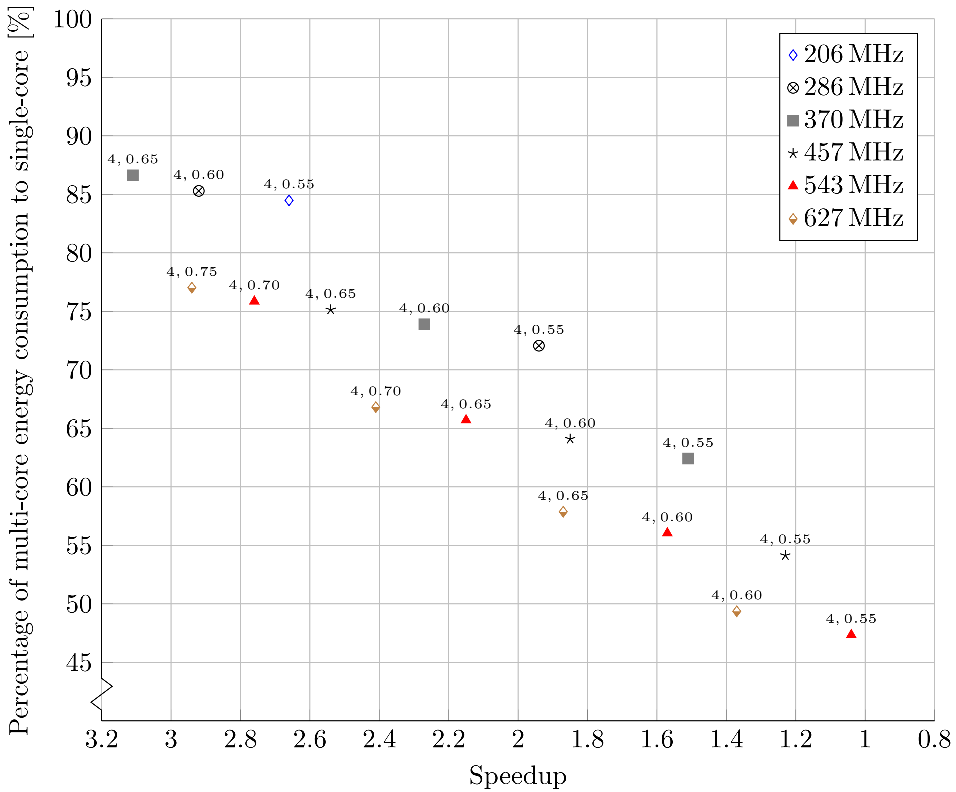


Figure 6.4: Energy savings for a scaled-down multi-core configuration compared to a scaled-down single-core configuration versus multi-core speedup. Texts represent (c, v_m) where c is the number of cores and v_m is the supply voltage that each of the c cores are configured with. Example: The blue ◊ means that four cores executing in parallel each configured with 0.55 V uses 84 % of the energy consumption that a single-core running on 206 MHz, in addition to a performance speedup of 2.66×.

and points more to the left has a higher speedup. Each point has a symbol that corresponds to a single-core clock frequency. For instance, the red △ compares the energy consumption of using four cores each on 0.55 V to using a single core running on 543 MHz. In this case we see that running four cores on o_1 only uses 47 % of the single-core consumption and achieves a speedup of 1.04 compared to the single-core configuration. This tells us again that a slowdown strategy where each of the four cores is configured with the lowest configuration saves the most energy and is therefore preferred over a race-to-halt strategy. This also increases the performance, which will allow for even lower end-user latencies.

The reason that we only plot multi-core configurations with four cores instead of two or three is that the total energy consumption for the ideal matrix multiplication is 98 nJ no matter how many cores are used. Four cores will perform better than two

and three with the same energy consumption, hence the largest energy savings can be observed when using four cores since four cores will perform better than two or three. In this case we see that there is possible to reduce the energy consumption to 88 % and all the way down to 47 % depending on the supply voltage of the multiple cores and the single-core operating we compare them to.

In order to realize this potential we assume several things. First, the work is fully parallelizable. Second, data can be moved in the background without the involvement of the cores. This can be achieved with direct memory access (DMA) to the shared memory, but would require having twice the local memory so that a single core can work uninterrupted on one part of the local memory and notify the DMA when data should be swapped with the other part that the DMA has already provided. This requires that the DMA can provide data faster than the core needs it. Third, data is fully cacheable. A single core must be able to have everything it needs for the entire work. This might either be costly or limit what external code can be used, as calling such code might evict cache lines needed by the core. Last, multiple parallel access to the shared memory is required to avoid memory contention. This can be achieved through increasing the width of the bus or using several workers that can handle memory requests simultaneously.

6.3 Energy Proportionality

We briefly summarized the idea of energy proportionality as defined by Barroso and Hölzle (2007) in Chapter 1. We will here provide an analysis in terms of comparing the scenario this work models versus what was modeled by Barroso and Hölzle.

The main difference between this work and Barroso and Hölzle (2007) is that Barroso and Hölzle focuses on server applications that experience varying loads. This means the resources of a particular server can be lightly or heavily used. This is not how our applications and prototype works — we only have applications for which the prototype experiences a fixed load at each program execution. This fixed load can be executed fast or slow depending on the operating point at which the prototype is configured to use, but the prototype must still face the same load for each classification.

There are also differences in the way the two classes of computing systems work. Barroso and Hölzle note that even though a data center might experience a low load, the individual servers might not be able to go into a sleep mode as there might be stored important data on that particular server that must be accessed from other servers or background tasks that make it impossible for that server to sleep (Barroso and Hölzle, 2007). This is in contrast to ULP devices that are able and expected to sleep when there is nothing to do. Therefore Barroso and Hölzle argue that computing systems, especially servers, should strive to be energy proportional, i.e. consume little power when idle and proportionally more power with increased utilization levels. We agree with Barroso and Hölzle on this point.

Based on these observations we find that the energy proportionality concept as defined in Barroso and Hölzle (2007) cannot be used for ULP devices in soft real-time contexts.

Chapter 7

Related Work

This chapter describes the state of the art solutions found in earlier works concerning running ML applications on ULP devices, task scheduling in soft real-time systems and the question of slowdown versus race-to-halt.

7.1 ML Applications on ULP Devices

Garofalo et al. (2019) develops “PULP-NN, an optimized computing library for a parallel ultra-low-power tightly coupled cluster of RISC-V processors”. They present several contributions. First, they develop a library containing optimized kernels for use with RISC-V-based processors. Second, they optimize this library for a cluster of parallel ultra-low-power RISC-V cores. We see similarities between the work done by Garofalo et al. and our work, primarily that both run ML applications that utilizes a quantized neural network (QNN) on ULP devices with one or more cores. We note that Garofalo et al. see almost eight times the performance improvement when using eight cores, meaning performance increases almost linearly with the increase in cores. When analyzing lost clock cycles, almost 20 % of them are caused by memory contention as all eight cores share a single L1 memory, which also is the case for the Nordic Semiconductor prototype. They also use an architecture for which memory requests to different banks can be served in parallel. We see significantly more memory contention (and lack of performance improvement when parallelizing the applications) compared to Garofalo et al., although this can be attributed to a lack of a working data cache and a shared memory that can only be accessed by a single core at a time.

Sliwa et al. (2020) has the same objective has Gopinath et al. — they develop an open source framework called LIMITS that seeks to deploy pre-trained ML models on ULP devices with as little as 16 KiB of RAM through source code generation. What differentiates Sliwa et al. is their work on automating the deployment of the source code through taking the actual deployment target into consideration.

They are then able to generate platform-specific source code and automate the deployment of to allow rapid development and deployment.

Brunelli et al. (2019) develops a ML agriculture application that is deployed on a Raspberry Pi3 (which is more powerful than the types of devices we consider in our work) for detecting codling moth in apple orchards and notifying the client of any anomalies. The system is powered by a regular battery and a solar panel, and is designed to work unsupervised. The system uses a deployed pre-trained deep neural network (DNN) model and a specialized visual processing unit (VPU) for performing the ML operations, which except for the VPU is similar to us. They see that with a 9000 mA h battery and a solar panel on 0.5 W of a few hundred square centimeters the device could operate indefinitely, which they claim "represents a breakthrough for agricultural activities (. . .)" (Brunelli et al., 2019).

Kartsch et al. (2019) develops a "smart sensor node for IoT and HMI based on a programmable Parallel Ultra-Low-Power (PULP) platform". They implement a gesture-recognition based application that performs ML operations in a real-time setting. This scenario is similar to what we model in our work. The difference is that their hardware platform supports native floating-point operations, which we do not need by generating a QNN that are deployed to the platform with SeeDot. They are able to operate the application with a power envelope of 11.84 mW for up to 35 h in active mode and 1000 h in standby. When they distribute the workload across eight different cores, they see a performance improvement of 20.4× and a 11.4× reduction in energy consumption.

7.2 Task Scheduling in Soft Real-Time Systems

Mohseni et al. (2019) develops the Hard Disk Drive and CPU Scheduling (HCS) scheduling algorithm for devices with multiple cores and hard disks with an objective to find a trade-off between energy consumption and execution time and still minimize the amount of missed tasks. They consider the scheduling of multiple parallel tasks that have their own deadlines. The algorithm consists of multiple stages that execute sorted tasks based on their ready time, execution time and CPU finish time. They also change execution times of tasks through DVFS in order to reduce waiting times. They compare the performance of their algorithm to the Hard Disk Drive and CPU Scheduling Unchanged Execution Time (HCS_UE) scheduling algorithm that does not change execution times and see higher CPU utilization and shorter execution times. They do not present data on energy consumption. Their scenario is different than what we consider as we only consider ULP scenarios with a single task and deadline. We also do not consider multiple hard disk drives (HDDs), but rather a single shared RAM. A HDD is slower than RAM, but the idea of having multiple cores accessing memory in parallel would be beneficial in our case.

Muhuri et al. (2020) seek to solve the energy efficient real-time scheduling multi-objective problem by minimizing the energy consumption and maximizing the earliness of tasks. This is similar to our objective. They consider quite similar

ranges of clock frequencies and supply voltages as is done in our work — the clock frequencies range from 13 MHz to 624 MHz and the supply voltage from 0.8 V to 1.6 V. They define a penalty function for energy efficiency and an earliness function that they try to solve simultaneously with an algorithm called the ϵ-constraint coupled energy efficient genetic algorithm (ϵ-EEGA). They find that the implemented solution improves energy efficiency and performance in terms of computational time compared to other solutions.

7.3 Slowdown versus Race-to-halt

Das et al. (2015) poses the question of whether slowdown through DVFS or race-to-halt is more appropriate for multi-threaded workloads. They find that the answer to this question depends on three factors. The first factor is the workload of the application that is executed, i.e. if it is mainly CPU bound or memory bound (in which the latter means the application often communicates with the memory system). The second factor is variations between the produced chip components, i.e. the CPU. The last factor is whether or not the chip has support for simultaneous multi-threading (SMT). Our definitions of slowdown and race-to-halt are similar except they do not model that the slowdown strategy involves having the device enter sleep mode when the program execution is completed. They implement an algorithm that tries to discover what they call the *break-even point* that represents the energy ratio. They state that having an energy ratio higher than 1 means that the race-to-halt strategy is preferred and vice versa for the slowdown strategy. The algorithm will then automatically switch to the other strategy when this break-even point is met. They find that different frequencies on different applications have different break-even points, and that their solution improves energy consumption by 13 % while maintaining the same performance. We do not consider a break-even point because the execution times for our applications do not vary, hence there is a single configuration that reduces the energy consumption the most while still completing program execution within the deadline.

Imes and Hoffmann (2015) considers the same objective as we do in our work, namely minimizing energy consumption for embedded devices that operate with imposed performance requirements like a soft real-time deadline. They consider the question of *never-idle* versus race-to-idle, where the former is defined as the configuration that finishes a task right before the deadline, which can be considered as the slowdown strategy, and the latter is the same as what we call race-to-halt. The difference between our slowdown strategies is that they only consider a single configuration, while we consider several that also sleep when they complete program execution. They find that different embedded systems require different strategies to achieve minimal energy consumption and that choosing the wrong strategy can significantly impact energy consumption. We note that Imes and Hoffmann uses devices that are more powerful than the ones we consider, namely a Sony VAIO tablet computer with a 600 MHz to 1.5 GHz dual-core processor and eleven DVFS configurations, and an ARM big.LITTLE board with two 500 MHz and 800 MHz to

1.2 GHz and 1.6 GHz quad-core processors.

Chapter 8

Conclusion and Future Work

In this work we implemented several applications performing machine learning (ML) operations in a single-core and multi-core context on the Nordic Semiconductor prototype, specifically the SeeDot application and two matrix multiplication applications. We measured their performance and energy consumption in terms of clock frequency, supply voltage and the number of processing cores. We analyzed our data with the focus of minimizing energy consumption under a performance constraint like a soft real-time deadline and asked the question of whether a slowdown strategy through dynamic voltage-frequency scaling (DVFS) or race-to-halt was preferred given the applications and prototype architecture we concerned ourselves with in this work.

In the single-core case we found that the lowest operating point that completed within the deadline had the lowest energy consumption. In the multi-core case we analyzed the theoretical potential for reducing energy consumption. We compared scaled-down multi-core configurations with scaled-down single-core configurations and found that the energy consumption could be reduced to between 47 and 88 % of the original single-core energy consumption over a clock frequency range of 206 to 543 MHz when using multiple scaled-down cores. We found that using multiple cores on the lowest configuration minimized the energy consumption the most. Hence in both cases a slowdown strategy configured with the lowest configuration that met the deadline had the minimal energy consumption.

Future work entails implementing an effective scheduling mechanism that allows for effective communication with memory and cooperation between cores, the primary goal being to reduce the observed memory contention. It will also be important to perform our experiments on a prototype with an architecture that does not suffer from the same limitations as the one we have used in order to reveal the practical benefits of utilizing multiple cores. This might entail being able to parallelize an application further with additional cores, enabling multiple cores to access shared memory simultaneously and offering effective sleep modes and sufficient data cache memory.

Bibliography

Aase, E.V., 2019. Exploring Energy-Performance Trade-offs for ML Applications on ULP Devices. Project report in TDT4501. Department of Computer Science, NTNU – Norwegian University of Science and Technology.

Akpan, O., 2006. Efficient parallel implementation of the fox algorithm., pp. 600–607.

Alqadi, Z., Aqel, M., 2008. Performance analysis and evaluation of parallel matrix multiplication algorithms .

Atmel, 2015. 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash Datasheet.

Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O., 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication, in: Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures, Association for Computing Machinery, New York, NY, USA. p. 193–204. URL: https://doi.org/10.1145/2312005.2312044, doi:10.1145/2312005.2312044.

Barroso, L.A., Hölzle, U., 2007. The case for energy-proportional computing. Computer 40, 33–37. doi:10.1109/MC.2007.443.

Brunelli, D., Albanese, A., d'Acunto, D., Nardello, M., 2019. Energy neutral machine learning based iot device for pest detection in precision agriculture. IEEE Internet of Things Magazine 2, 10–13. doi:10.1109/IOTM.0001.1900037.

Chamberlin, H., 1985. Musical Applications of Microprocessor. Sams, USA.

Das, A., Merrett, G.V., Al-Hashimi, B.M., 2015. The slowdown or race-to-idle question: Workload-aware energy optimization of smt multicore platforms under process variation, in: Conference on Design, Automation and Test in Europe 2016. URL: https://eprints.soton.ac.uk/384892/.

Dennis, D.K., Gaurkar, Y., Gopinath, S., Gupta, C., Jain, M., Kumar, A., Kusupati, A., Lovett, C., Patil, S.G., Simhadri, H.V., 2019. EdgeML: Machine Learning for resource-constrained edge devices. URL: https://github.com/Microsoft/EdgeML.

Din, I.U., Guizani, M., Rodrigues, J.J., Hassan, S., Korotaev, V.V., 2019. Machine learning in the internet of things: Designed techniques for smart cities. Future Generation Computer Systems 100, 826 – 843. URL: http://www.sciencedirect.com/science/article/pii/S0167739X19304030, doi:https://doi.org/10.1016/j.future.2019.04.017.

Garofalo, A., Rusci, M., Conti, F., Rossi, D., Benini, L., 2019. Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 378, 20190155. URL: http://dx.doi.org/10.1098/rsta.2019.0155, doi:10.1098/rsta.2019.0155.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press. http://www.deeplearningbook.org.

Gopinath, S., Ghanathe, N., Seshadri, V., Sharma, R., 2019. Compiling kb-sized machine learning models to tiny iot devices, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 79–95. URL: https://doi.org/10.1145/3314221.3314597, doi:10.1145/3314221.3314597.

Gupta, H., Sadayappan, P., 1994. Communication Efficient Matrix-Multiplication on Hypercubes. Technical Report 1994-25. Stanford Infolab. URL: http://ilpubs.stanford.edu:8090/59/.

Hennessy, J.L., Patterson, D.A., 2017. Computer Architecture, Sixth Edition: A Quantitative Approach. 6th ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Imes, C., Hoffmann, H., 2015. Minimizing energy under performance constraints on embedded platforms: Resource allocation heuristics for homogeneous and single-isa heterogeneous multi-cores. SIGBED Rev. 11, 49–54. URL: https://doi.org/10.1145/2724942.2724950, doi:10.1145/2724942.2724950.

Kartsch, V., Guermandi, M., Benatti, S., Montagna, F., Benini, L., 2019. An energy-efficient iot node for hmi applications based on an ultra-low power multicore processor, in: 2019 IEEE Sensors Applications Symposium (SAS), pp. 1–6. doi:10.1109/SAS.2019.8705984.

Khan, W., Rehman, M., Zangoti, H., Afzal, M., Armi, N., Salah, K., 2020. Industrial internet of things: Recent advances, enabling technologies and open challenges. Computers & Electrical Engineering 81, 106522. URL: http://www.sciencedirect.com/science/article/pii/S0045790618329550, doi:https://doi.org/10.1016/j.compeleceng.2019.106522.

Mohseni, Z., Kiani, V., Rahmani, A., 2019. A task scheduling model for multi-cpu and multi-hard disk drive in soft real-time systems. International Journal of Information Technology and Computer Science 11, 1–13. doi:10.5815/ijitcs.2019.01.01.

Muhuri, P.K., Nath, R., Shukla, A.K., 2020. Energy efficient task scheduling for real-time embedded systems in a fuzzy uncertain environment. IEEE Transactions on Fuzzy Systems , 1–1.

Murphy, K.P., 2012. Machine learning: a probabilistic perspective. Cambridge, MA.

Nilsson, J.W., Riedel, S.A., 2015. Electric Circuits, Second Edition. 2nd ed., Pearson Education Limited.

Sethi, P., Sarangi, S., 2017. Internet of things: Architectures, protocols, and applications. Journal of Electrical and Computer Engineering 2017, 1–25. doi:10.1155/2017/9324035.

Sliwa, B., Piatkowski, N., Wietfeld, C., 2020. LIMITS: Lightweight machine learning for IoT systems with resource limitations, in: 2020 IEEE International Conference on Communications (ICC), Dublin, Ireland.

Strassen, V., 1969. Gaussian elimination is not optimal. Numer. Math. 13, 354–356. URL: https://doi.org/10.1007/BF02165411, doi:10.1007/BF02165411.

Wolf, W., 2008. Computers as Components, Second Edition: Principles of Embedded Computing System Design. 2nd ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Appendix A

Effect of the COVID-19 Pandemic

This work was carried out during the COVID-19 pandemic. As we spent our time primarily in the offices of Nordic Semiconductor conducting experiments with a physical prototype, when the Norwegian government declared nation-wide regulations on the 12$^{\text{th}}$ of March, we were as a direct consequence of this no longer able to conduct experiments.

At this point we had only collected about 15 to 20 % of the data that we planned to use in this work. Being forced to stay at home, we continued the work on other parts of the thesis to remedy the situation as much as possible. In this time we made progress on the chapters of this work that did not directly relate to the results of the experiments and at the same time we prepared for the possibility that we would be allowed back to conduct the remaining experiments.

At the start of April the Norwegian government communicated that they would allow students at the end of their university degree access to universities such that they could finish their work. At this point we started a discussion with Nordic Semiconductor and on the 22$^{\text{nd}}$ of April we were allowed to borrow the necessary equipment from the office for about two weeks to continue the experiments. This enabled us to collect the remainder of the data.

At the same time, one of the major challenges when implementing the multi-core applications was the fact that we had no built-in mechanism for scheduling memory accesses between multiple processing cores. This contributed greatly to the memory contention we observed. When we lost access to the equipment on the 12$^{\text{th}}$ of March, we did not have any results for multi-core program executions, and since we spent all our time conducting experiments in the two weeks we regained access to the equipment, we had no time to implement any mechanisms that would remedy this problem.

In the end, even though we did not implement any scheduling mechanisms that could have remedied the memory contention problem, because we were able to work on other parts of this thesis when we lacked access to the device, and because Nordic

Semiconductor allowed us to borrow the equipment for two weeks, we consider ourselves lucky that the COVID-19 pandemic did not have a stronger impact on our work.

Appendix B

Multi-Core Analysis Table

Table B.1: Scaled-down multi-core energy savings versus scaled-down single-core

Single-Core			Multi-Core				E_m/E_s
f_s[MHz]	v_s[V]	E_s[nJ]	c_m	v_m[V]	E_m[nJ]	s	
136	0.55	98		N/A			100 %
627	0.85	235	4	0.80	207	3.47	88.09
543	0.80	207	4	0.75	181	3.37	87.44
457	0.75	181	4	0.70	157	3.27	86.74
370	0.70	157	4	0.65	136	3.11	86.62
286	0.65	136	4	0.60	116	2.92	85.29
206	0.60	116	4	0.55	98	2.66	84.48
627	0.85	235	4	0.75	181	2.94	77.02
543	0.80	207	4	0.70	157	2.76	75.85
457	0.75	181	4	0.65	136	2.54	75.14
370	0.70	157	4	0.60	116	2.27	73.89
286	0.65	136	4	0.55	98	1.94	72.06
627	0.85	235	4	0.70	157	2.41	66.81
543	0.80	207	4	0.65	136	2.15	65.70
457	0.75	181	4	0.60	116	1.85	64.09
370	0.70	157	4	0.55	98	1.51	62.42
627	0.85	235	4	0.65	136	1.87	57.87
543	0.80	207	4	0.60	116	1.57	56.04
457	0.75	181	4	0.55	98	1.23	54.14
627	0.85	235	4	0.60	116	1.37	49.36
543	0.80	207	4	0.55	98	1.04	47.34

Appendix C

Source Code

SeeDot

Multi-Core

main.c

```
1  int main(void) {
2      uint32_t cpu_id = read_csr(mhartid);
3
4      if (cpu_id == MASTER) {
5          while(1) {
6              while(busy_wait == 0xaabbccdd) {}
7
8              int res = seedotFixed(features);
9
10             // ‘res‘ can be compared with ‘label‘
11
12             busy_wait = 0xaabbccdd;
13         }
14     } else if (cpu_id == SLAVE) {
15         slave_cpu_wait_work_loop();
16     }
17
18     return 0;
19 }
```

seedot_fixed.c

```
1  void wait_for_mat_sub() {
2      while (!slave_can_go) {}
3
4      MatSub(a, b, c, 13, 25);
5
6      slave_can_go = false;
7  }
8
```

```
 9 void wait_for_mat_mul_cn() {
10     while (!slave_can_go) {}
11
12     MatMulCN(a, b, c, 5, 10);
13
14     slave_can_go = false;
15 }
16
17 void slave_cpu_wait_work_loop() {
18     while (1) {
19         wait_for_mat_sub();
20         wait_for_mat_mul_cn();
21     }
22 }
23
24 void OptimizeFunction(const MYINT *A, MYINT *B, MYINT *C, uint8_t f) {
25     size_t s = sizeof(MYINT) * (f == 0 ? 25 : 10);
26
27     memcpy(a, A, s);
28     memcpy(b, B, s);
29
30     slave_can_go = true;
31
32     if (f == 0) MatSub(a, b, &c[0], 0, 13);
33     if (f == 1) MatMulCN(a, b, &c[0], 0, 5);
34
35     // Slave will set slave_can_compute to false itself
36     while (slave_can_go) {}
37
38     memcpy(C, c, s);
39 }
40
41 int seedotFixed(MYINT *X) {
42     MYINT tmp4;
43     MYINT tmp5[25];
44     MYINT i;
45     MYINT tmp6[25];
46     MYINT tmp7;
47     MYINT tmp8[1][25];
48     MYINT tmp10;
49     MYINT tmp9[25];
50     MYINT tmp11;
51     MYINT tmp15;
52     MYINT tmp12;
53     MYINT tmp13;
54     MYINT tmp14;
55     MYINT tmp17[10];
56     MYINT tmp16[1];
57     MYINT tmp18[10];
58     MYINT tmp19;
59
60     tmp4 = 1055531162L;
61
62
63     // W |*| X
64     memset(tmp5, 0, sizeof(MYINT) * 25);
65     SparseMatMul(&Widx[0], &Wval[0], X, &tmp5[0], 256, 32768L, 32768L,
```

```
        256);
66
67      memset(tmp18, 0, sizeof(MYINT) * 10);
68      i = 0;
69      for (MYINT i0 = 0; (i0 < 55); i0++) {
70
71          // WX - B
72          OptimizeFunction(&tmp5[0], &B[i][0][0], &tmp6[0], 0);
73
74          tmp7 = (-tmp4);
75
76          // del^T
77          Transpose(&tmp6[0], &tmp8[0][0]);
78
79
80          // tmp8 * del
81          MatMulNN(&tmp8[0][0], &tmp6[0], &tmp10, &tmp9[0]);
82
83
84          // tmp7 * tmp10
85          ScalarMul(&tmp7, &tmp10, &tmp11);
86
87
88          // exp(tmp11)
89          if (((-tmp11) < 30669)) {
90              tmp13 = 0;
91              tmp14 = 0;
92          } else {
93              tmp12 = (((-tmp11) - 30669) << 10);
94              tmp13 = ((tmp12 >> 26) & 63);
95              tmp14 = ((tmp12 >> 20) & 63);
96          }
97          tmp15 = ((EXP20A[tmp13] >> 15) * (EXP20B[tmp14] >> 15));
98
99          // Z * tmp15
100         OptimizeFunction(&Z[i][0][0], &tmp15, &tmp17[0], 1);
101
102         for (MYINT i1 = 0; (i1 < 10); i1++) {
103             for (MYINT i2 = 0; (i2 < 1); i2++) {
104                 tmp18[i1] = (tmp18[i1] + (tmp17[i1] / 64));
105             }
106         }
107         i = (i + 1);
108     }
109
110     // argmax(res)
111     return ArgMax(&tmp18[0]);
112 }
```

Linear Algebra Library Functions

Matrix Multiplication (Multi-Core)

```
1       void MatMulCN(const MYINT *A, MYINT *B, MYINT *C, MYINT lower,
        MYINT upper) {
2       MYINT tmp[1];
3       MYINT shrA = 32768L, shrB = 32768L, I = 10, K = 1, J = 1, H1 = 0,
        H2 = 0;
4
5       for (MYINT i = lower; i < upper; i++) {
6           for (MYINT j = 0; j < J; j++) {
7               for (MYINT k = 0; k < K; k++) {
8                   MYINT a = A[i * K + k];
9                   MYINT b = B[k * J + j];
10
11                  a = a / shrA;
12                  b = b / shrB;
13
14                  tmp[k] = a * b;
15              }
16
17              MYINT count = K, depth = 0;
18              bool shr = true;
19
20              while (depth < (H1 + H2)) {
21                  if (depth >= H1)
22                      shr = false;
23
24                  for (MYINT p = 0; p < (K / 2 + 1); p++) {
25                      MYINT sum;
26                      if (p < (count >> 1))
27                          sum = tmp[2 * p] + tmp[(2 * p) + 1];
28                      else if ((p == (count >> 1)) && ((count & 1) == 1)
        )
29                          sum = tmp[2 * p];
30                      else
31                          sum = 0;
32
33                      if (shr)
34                          tmp[p] = sum / 2;
35                      else
36                          tmp[p] = sum;
37                  }
38                  count = (count + 1) >> 1;
39
40                  depth++;
41              }
42
43              C[i * J + j] = tmp[0];
44          }
45      }
46  }
```

Matrix Subtraction (Multi-Core)

```
1 void MatSub(MYINT *A, const MYINT *B, MYINT *C, MYINT lower, MYINT
```

```
    upper) {
2       MYINT I = 25, J = 1, shrA = 1, shrB = 256, shrC = 1;
3       for (MYINT i = lower; i < upper; i++) {
4           for (MYINT j = 0; j < J; j++) {
5               MYINT a = A[i * J + j];
6               MYINT b = B[i * J + j];
7
8               a = a / shrA;
9               b = b / shrB;
10
11              MYINT c = a - b;
12              c = c / shrC;
13
14              C[i * J + j] = c;
15          }
16      }
17      return;
18  }
```

Ideal Matrix Multiplication

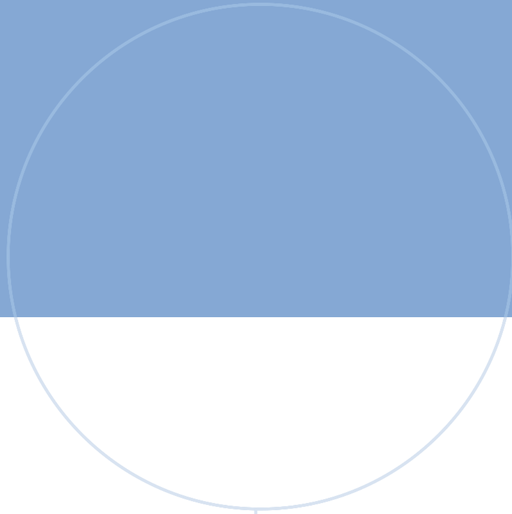
Single-Core

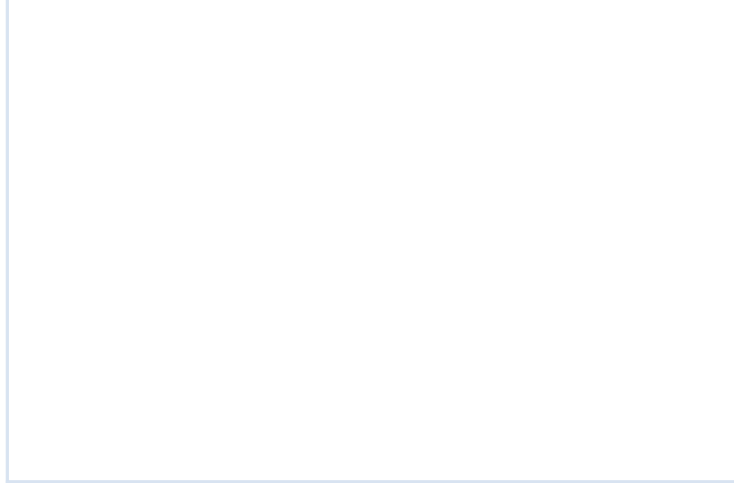
```
1 void compute(uint32_t *A, uint32_t *B, uint32_t *C, uint32_t lower,
      uint32_t upper) {
2     for (uint32_t j = lower; j < upper; j++) {
3         for (int i = 0; i < 500; i++) {
4             C[j] = A[j] * B[j];
5         }
6     }
7 }
8
9 int main(void) {
10     uint32_t aa[100], bb[100], cc[100];
11     for (int i = 0; i < SIZE; i++) {
12         aa[i] = i;
13         bb[i] = SIZE - i;
14         cc[i] = 0;
15     }
16
17     compute(aa, bb, cc, 0, SIZE);
18
19     return 0;
20 }
```

Multi-Core

```
1 void compute(uint32_t *A, uint32_t *B, uint32_t *C, uint32_t lower,
      uint32_t upper) {
2     for (uint32_t j = lower; j < upper; j++) {
3         for (int i = 0; i < 500; i++) {
4             C[j] = A[j] * B[j];
5         }
6     }
7 }
8
9 void perform_computation(uint32_t *A, uint32_t *B, uint32_t *C,
      uint32_t lower,  uint32_t upper) {
10     size_t max_s = sizeof(uint32_t) * SIZE;
11
12     memcpy(a, A, max_s);
13     memcpy(b, B, max_s);
14
15     compute(A, B, c, lower, upper);
16
17     memcpy(C, c, max_s);
18 }
19
20 uint32_t perform_all_computations(bool is_master) {
21     uint32_t aa[100], bb[100], cc[100];
22     for (int i = 0; i < SIZE; i++) {
23         aa[i] = i;
24         bb[i] = SIZE - i;
25         cc[i] = 0;
26     }
```

```
27
28     perform_computation(aa, bb, cc, 0, SIZE);
29
30     return 0;
31 }
32
33 int main(void) {
34     uint32_t cpu_id = read_csr(mhartid);
35
36     if (cpu_id == MASTER) {
37         return perform_all_computations(true);
38
39     } else if (cpu_id == SLAVE) {
40         return perform_all_computations(false);
41     }
42 }
```





NTNU
Norwegian University of
Science and Technology